Einsteiger-Workshop
Eclipse Rich Client Platform (RCP)

Volker Wegert
rcp@volker-wegert.de

13. September 2008

Inhaltsverzeichnis

1 Vorwort **3**
2 Erste Schritte **5**
3 Komponentenstruktur **9**
4 Dialogprogrammierung **13**
5 Oberflächengestaltung **17**
6 Anwendung **21**
7 Paketierung und Export **25**
8 Menüs und Aktionen **28**
9 Erweiterbarkeit (I) **31**
10 Erweiterbarkeit (II) **36**
Literatur **40**

Die aktuelle Version dieser Unterlagen finden Sie jeweils im Internet unter `http://www.volker-wegert.de/rcp-einsteiger-workshop`.

Zusammenfassung

Die Eclipse Rich Client Platform (RCP) ist eine stabile und bewährte Grundlage zur Entwicklung client-seitiger Anwendungen. Der Mächtigkeit dieser Plattform steht allerdings eine Komplexität gegenüber, die viele Einsteiger abschreckt. Dieser Workshop führt praxisorientiert durch die ersten Schritte der Erstellung einer RCP-Anwendung und vermittelt die wichtigsten Grundbegriffe zur weiteren Einarbeitung.

Lizenz

Dieses Werk ist unter einem Creative Commons Namensnennung-Weitergabe unter gleichen Bedingungen 2.0 Deutschland Lizenzvertrag lizenziert. Um die Lizenz anzusehen, gehen Sie bitte zu `http://creativecommons.org/licenses/by-sa/2.0/de/` oder schicken Sie einen Brief an Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

1 Vorwort

Vorkenntnisse

RCP-Kenntnisse sind für die Teilnahme an diesem Workshop nicht erforderlich, es handelt sich um eine ausgesprochene Einsteiger-Veranstaltung. Wer bereits Erfahrungen mit der RCP gesammelt hat, wird in diesem Workshop vermutlich nicht viel Neues erfahren. Erfahrungen im Umgang mit der Eclipse-Entwicklungsumgebung sind von Vorteil, Grundkenntnisse in Java sind Voraussetzung.

Benötigte Programme

Um den Beispielen folgen zu können, benötigen Sie folgende Programme:

- Java 5 - frühere Versionen sind auch möglich, allerdings müssen Sie dann einige der Beispiele etwas umständlicher schreiben, weil *generics* entfallen und Sie an einigen Stellen Iteratoren einziehen müssen.
- Das Eclipse SDK in der Version 3.4 – zum Beispiel als *Eclipse Classic*[1] oder als *Eclipse for RCP/Plug-In Developers*[2]. Frühere Versionen funktionieren zwar prinzipiell ebenfalls, können aber an verschiedenen Stellen Abweichungen in der Vorgehensweise erfordern.
- Die Eclipse SDK Examples, die Sie entweder von der Eclipse-Internetseite[3] oder über die integrierte Download-Funktion installieren können. Dieses Paket ist optional, hilft aber beim Einstieg sehr.

Vorbereitung

Neben den o. a. Programmen benötigen Sie noch einen Arbeitsbereich auf Ihrem Rechner. Dabei sollten Sie einplanen, dass wir nicht nur den *workspace* der Eclipse-Entwicklungsumgebung, sondern auch noch ein separates Exportverzeichnis benötigen werden – am Besten legen Sie sich ein eigenes Verzeichnis an, das den *workspace* und alle anderen Artefakte aufnimmt.

Wenn Sie bereits Erfahrungen mit der Eclipse-Entwicklungsumgebung gemacht haben, sollten Sie sich Ihren Arbeitsbereich so einrichten, wie Sie es gewohnt sind. Wenn Sie noch nicht mit der Eclipse gearbeitet haben, sind folgende Einstellungen empfehlenswert:

[1] http://www.eclipse.org/downloads/packages/eclipse-classic-34/ganymeder

[2] http://www.eclipse.org/downloads/packages/eclipse-rcpplug-developers/ganymeder

[3] http://download.eclipse.org/eclipse/downloads/drops/R-3.4-200806172000/index.php#ExamplePlugins

- *Run/Debug* → *Launching*: *Launch Operation* auf *Always launch the previously launched application* umstellen
- *Java* → *Editor* → *Typing*: Unter *Automatically insert at correct positions* sowohl *Semicolons* als auch *Braces* aktivieren

Typographische Konventionen

In diesem Text finden sich naturgemäß viele englische Fachbegriffe. Zugunsten eines höheren Wiedererkennungswerts in der größtenteils englischen Literatur und Dokumentation wurde auf jegliche Übersetzung verzichtet. Englische Begriffe sind im Text *kursiv* gesetzt, um sie von technischen Namen abzugrenzen, die in `Nichtproportionalschrift` gesetzt sind. Beispiel: *view* bezeichnet das Konzept, `View` eine Klasse.

Der Text enthält Beschreibungen der im Rahmen des Workshops druchgeführten Entwicklungsaktivitäten – eine Art „Anleitung zum Mitspielen." Diese Anweisungen sind mit folgendem Symbol gekennzeichnet:

☞ Starten Sie jetzt Ihre Eclipse-Entwicklungsumgebung.

Gelegentlich sind längere Code-Strecken abgedruckt, in denen nur wenige gezielte Änderungen notwendig sind; die Änderungen sind in diesem Fall durch Fettdruck hervorgehoben.

2 Erste Schritte

Eine RCP-Anwendung von Grund auf neu anzulegen, erfordert normalerweise einiges an Vorwissen, über das Sie als Teilnehmer dieses Einsteiger-Workshops vermutlich nicht verfügen werden. Es gibt allerdings eine Reihe von Vorlagen, die den Einstieg deutlich vereinfachen. Eine dieser Vorlagen, das *RCP Mail Template*, werden wir jetzt nutzen:

☞ Wählen Sie *File* → *New* → *Project...*.

☞ Wählen Sie aus der Liste *Plug-in Project* und fahren Sie mit *Next* fort.

☞ Geben Sie als Projektnamen `org.example.myapp` ein. Belassen Sie die Standardwerte für die restlichen Einstellungen und fahren Sie mit *Next* fort.

☞ Ändern Sie ggf. den Namen des *plug-in* und achten Sie darauf, dass die Frage *Would you like to create a rich client application?* mit *Yes* beantwortet wird. Fahren Sie anschließend mit *Next* fort.

☞ Wählen Sie das *RCP Mail Template* aus der Liste aus und schliessen Sie die Erstellung mit *Finish* ab.

Wenn Sie an dieser Stelle gefragt werden, ob Sie die *Plug-in Development perspective* öffnen möchten, bestätigen Sie die Frage mit *Yes*. Im linken Bereich des Eclipse-Fensters sehen Sie jetzt das neu angelegte Projekt im *Package Explorer*. Außerdem wurde automatisch ein Editor geöffnet, in dem Sie einige Eigenschaften des soeben angelegten *plug-in*-Projekts wiederfinden. In der rechten Hälfte dieses Editors finden Sie die Aktion *Launch an Eclipse application* (Abbildung 1).

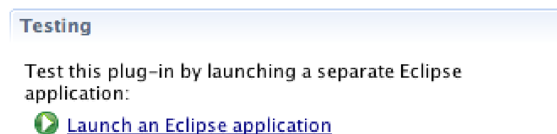

Abbildung 1: Aktion *Launch an Eclipse application*

☞ Klicken Sie auf diese Aktion, um die neu angelegte RCP-Anwendung zu starten.

Nach wenigen Sekunden erscheint ein *splash screen*, der kurz darauf vom Anwendungsfenster (Abbildung 2) abgelöst wird.

Mit Hilfe dieser einfachen Anwendung werden wir jetzt diesem Workshop die Grundlagen der Rich Client Platform erkunden. Tatsächlich haben wir jetzt schon ein grundlegendes Konzept kennengelernt: RCP-Anwendungen bestehen aus *plug-ins*, daher haben wir ein neues *plug-in*-Projekt angelegt. *plug-ins* werden analog zu Java-Paketen benannt. An vielen Stellen werden Sie den aus dem OSGi-Umfeld stammenden Begriff *Bundle* finden, der seit Eclipse 3.0 synonym zu *plug-in* verwendet werden kann. *plug-ins* sind die wesentliche Strukturierungseinheit einer RCP-Anwendung, so dass wir uns im Folgenden noch näher mit diesem Konstrukt auseinandersetzen werden. Um die Bedeutung der *plug-ins* zu verdeutlichen, führen Sie einmal folgende Schritte aus:

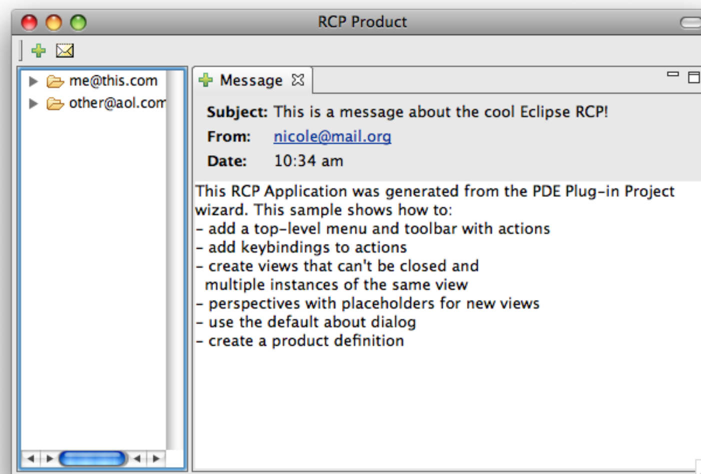

Abbildung 2: Anwendungsfenster nach Erstellung der Anwendung

☞ Schließen Sie Ihre Beispielanwendung, wenn sie noch läuft.

☞ Wählen Sie *Run* → *Run Configurations...* bzw. den entsprechenden Eintrag aus der *toolbar*.

☞ In dem nun erscheinenden Dialogfenster *Run Configurations* sollte bereits der Eintrag Ihrer Anwendung ausgewählt sein. Wechseln Sie auf die Registerkarte *Arguments*.

☞ Ergänzen Sie die Kommandozeilen-Optionen (*program arguments*) Ihrer Anwendung um `-console` und `-consoleLog` (vgl. Abbildung 3).

☞ Starten Sie Ihre Anwendung mit *Run*.

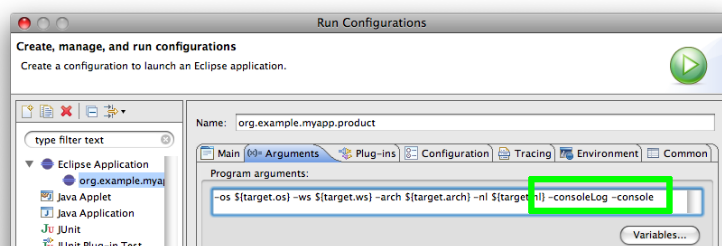

Abbildung 3: Kommandozeilenparameter zur Verwendung der Konsole

An Ihrer Anwendung hat sich dadurch nichts geändert, allerdings sollte in Ihrem Eclipse-Fenster jetzt eine neue Sicht mit dem Titel *Console* und dem Inhalt `osgi>` erschienen sein.

☞ Geben Sie in die *Console View* das Kommando `ss` (Abkürzung für *short status*) ein und bestätigen Sie mit der Eingabetaste.

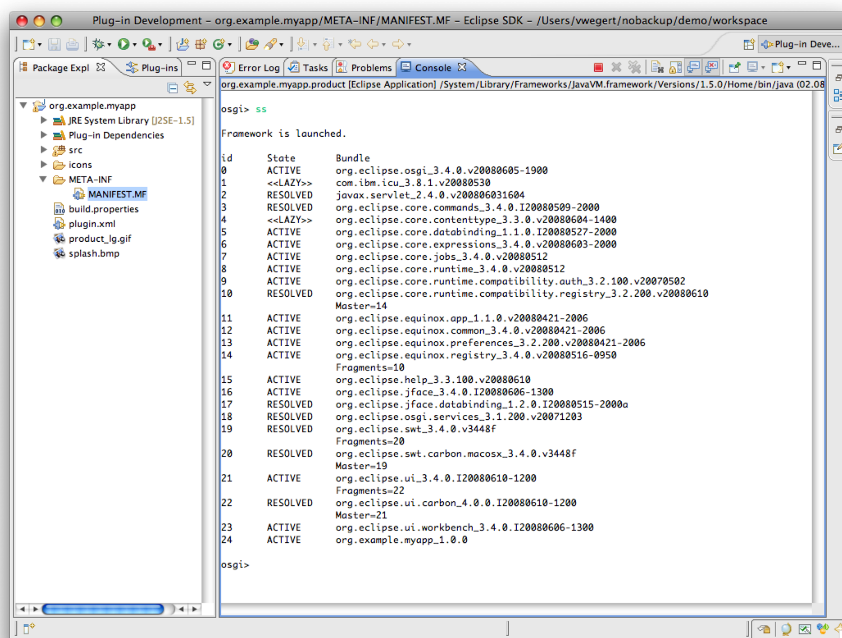

Abbildung 4: Liste der *bundles* der Anwendung

Durch diesen Befehl erhalten Sie eine kurze Übersicht aller *bundles* mit ihrem jeweiligen Status[4] wie in Abbildung 4 gezeigt. In dieser Liste werden Sie – in der Regel als letzten Eintrag – das *bundle* Ihrer RCP-Anwendung erkennen, das in Zusammenarbeit mit den anderen in der Liste aufgeführten *bundles* die RCP-Anwendung bildet. Bevor wir uns weiter mit der Definition und dem Zusammenspiel der einzelnen *bundles* beschäftigen, runden wir diesen ersten Abschnitt mit einem kurzen Überblick über den Inhalt des generierten *plug-in*-Projekts ab (vgl. Abbildung 5).

Die ersten beiden Einträge des Projekts stellen Verknüpfungen zu den eingebundenen Programmbibliotheken dar – wenn Sie mit der Eclipse bereits Java-Projekte bearbeitet haben, werden Sie diese Einträge kennen; wenn nicht, können Sie sie für den Augenblick ignorieren. Der Ordner `src` enthält das Paket mit den generierten Java-Quelltexten, die einen wesentlichen Teil der Anwendung ausmachen und die wir in den folgenden Abschnitten im Einzelnen besprechen werden. Ebenfalls zur Anwendung gehören die Symbole und Grafikdateien im Verzeichnis `icons` sowie im Hauptverzeichnis des Projekts, die Ihnen an verschiedenen Stellen in der Anwendung begegnen werden. Die Datei `build.properties` steuert den automatisierten Übersetzungs- und Paketierungsvorgang, mit dem wir uns im Rahmen dieses Workshops nicht beschäftigen werden. Die Dateien `META-INF/MANIFEST.MF` und `plugin.xml` schließlich stellen die zentralen *plug-in*-Deklarationen dar, die Gegenstand des nächsten Abschnitts sind.

[4]Mit dem Befehl `help` können Sie eine Liste der weiteren verfügbaren Befehle abrufen.

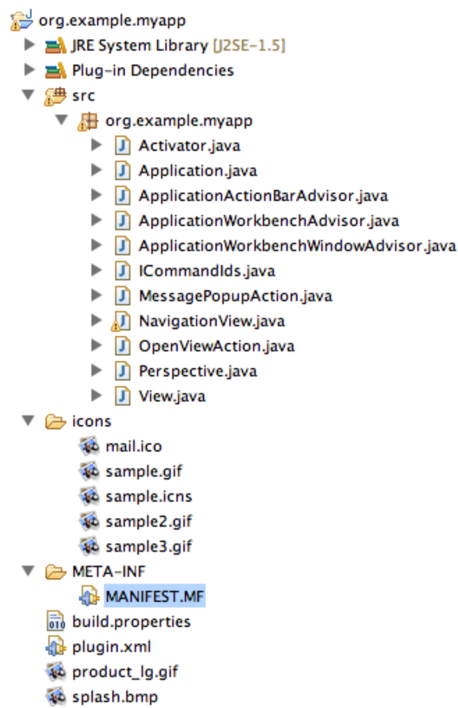

Abbildung 5: Inhalt des Anwendungs-Projekts

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *Simple plug-in example*
- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Reference* → *Other reference information* → *Runtime options*
- [Dau07], Kapitel 2.2 - Eine minimale RCP-Anwendung
- [SJB08], Kapitel 1.2 – In fünf Minuten zur ersten Rich-Client-Anwendung
- [SJB08], Kapitel 36.2 – OSGi-Konsole

3 Komponentenstruktur

Wie bereits angedeutet verteilen sich die Deklarationen und Eigenschaften eines *plug-in* unter Umständen auf mehrere Dateien. Bei neu erstellten *plug-ins* gibt es immer die Datei `META-INF/MANIFEST.MF`, zu der je nach Inhalt des *plug-in* noch eine Datei namens `plugin.xml` im Hauptverzeichnis kommt[5]. Die genaue Verteilung der Eigenschaften auf diese Dateien ist im Rahmen dieses Workshops nicht weiter von Belang, zumal das *Plug-in Development Environment* (PDE) einen sehr komfortablen Editor bereitstellt, der die Bearbeitung de Eigenschaften eines *plug-in* weitestgehend unabhängig vom eigentlichen Speicherort erlaubt. Dieser Editor lässt sich übrigens bei einem *plug-in*-Projekt auch über das Kontext-Menü im Package Explorer → *PDE Tools* → *Open Manifest* öffnen.

Nachdem wir uns im vorherigen Abschnitt schon kurz mit den Eigenschaften und Aktionen beschäftigt haben, die auf der Seite *Overview* angeboten werden, werden wir uns jetzt mit der Seite *Dependencies* beschäftigen. Ausgangspunkt hierfür ist die Frage, wie die in Abbildung 4 gezeigte Liste der geladenen *bundles* zustandekommt – immerhin haben wir ja nur ein *bundle* erstellt.

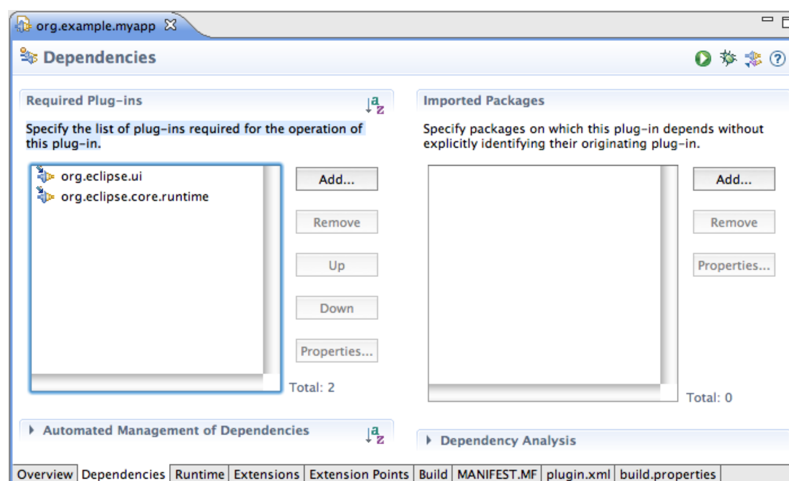

Abbildung 6: Abhängigkeiten des Anwendungs-*plug-in*

Ein Blick auf die Abhängigkeiten (Editor-Seite *Dependencies*, Abbildung 6) schafft hier Klarheit: In der generierten Anwendung wurde offensichtlich bereits festgelegt, dass das generierte *plug-in* zur Ausführung zwei weitere *plug-ins* benötigt, nämlich `org.eclipse.ui` und `org.eclipse.core.runtime`. Diese *plug-ins* finden sich in der Liste der geladenen plug-ins dann auch wieder. Die verbleibenden *plug-ins* kommen dadurch zusammen, dass die beiden Eclipse-*plug-ins* ihrerseits wieder Abhängigkeiten besitzen.

☞ Doppelklicken Sie auf eines der *Required plug-ins* in der Liste, um die zugehörige Definition zu öffnen.

☞ Versuchen Sie einmal, die Abhängigkeitspfade der tatsächlich geladenen *plug-ins*

[5] Es existieren allerdings auch noch *plug-ins* mit einem alten Definitionsformat, die keine `MANIFEST.MF` besitzen.

nachzuvollziehen.

- ☞ Kehren Sie zum Plug-In `org.example.myapp` zurück.
- ☞ Expandieren Sie den Bereich *Dependency Analysis* und wählen Sie *Show the plug-in dependency hierarchy*, um eine Übersicht über die Abhängigkeiten zu erhalten.

Die Auflösung dieser Abhängigkeiten ist Aufgabe des Laufzeitsystems und braucht uns an dieser Stelle nicht weiter zu interessieren – als angehende RCP-Entwickler geben wir lediglich die benötigten *plug-ins* an, und die Plattform erledigt den Rest. Bei der Deklaration der Abhängigkeiten können auch weitergehende Angaben wie die benötigten Versionen oder optionale Abhängigkeiten angegeben werden, die wir allerdings im Rahmen dieses Workshops nicht weiter behandeln werden. Stattdessen wenden wir uns einem einfachen Beispiel zu:

- ☞ Wählen Sie *File → New → Project. . .* .
- ☞ Wählen Sie aus der Liste *Plug-in Project* und fahren Sie mit *Next* fort.
- ☞ Geben Sie als Projektnamen `org.example.mybundle` ein. Belassen Sie die Standardwerte für die restlichen Einstellungen und fahren Sie mit *Next* fort.
- ☞ Ändern Sie ggf. den Namen des plug-ins und achten Sie darauf, dass die Frage *Would you like to create a rich client application?* mit ***No*** beantwortet wird.
- ☞ Schliessen Sie die Erstellung mit *Finish* ab[6].

Jetzt befindet sich in Ihrem Workspace ein weiteres *plug-in*-Projekt, das dem bereits erstellten Projekt sehr ähnelt, aber deutlich weniger Inhalt zeigt. Auch dieses neue *plug-in* besitzt eine `MANIFEST.MF`, allerdings kommt es offensichtlich[7] ohne `plugin.xml` aus. Die einzige enthaltene Klasse ist der sogenannte *Activator*, mit dem wir uns in Kürze noch beschäftigen werden.

- ☞ Wechseln Sie wieder in die Eigenschaften des *plug-in* `org.example.myapp` auf die Editor-Seite *Dependencies*.
- ☞ Fügen Sie eine Abhängigkeit zu dem gerade neu erstellten *plug-in* `org.example.mybundle` hinzu.
- ☞ Versuchen Sie, Ihre Anwendung zu starten.

An dieser Stelle wird der Start der Anwendung in aller Regel misslingen. Ein Blick in die *Console View* oder das Fehlerprotokoll zeigt die Ursache: *Missing required bundle org.example.mybundle_1.0.0.* Hintergrund dieser Fehlermeldung ist ein Problem in der Startkonfiguration, denn wir haben Eclipse noch nicht mitgeteilt, dass das neu erstellte *plug-in* zusammen mit der Anwendung gestartet werden soll.

- ☞ Wählen Sie *Run → Run Configurations. . .* bzw. den entsprechenden Eintrag aus der *toolbar*.

[6] Wenn Sie an dieser Stelle versehentlich *Next* gewählt haben, achten Sie darauf, keine der auf der letzten Seite angebotenen Vorlagen zu verwenden.

[7] noch

☞ In dem nun erscheinenden Dialogfenster *Run Configurations* sollte bereits der Eintrag Ihrer Anwendung ausgewählt sein. Wechseln Sie auf die Registerkarte *Plug-ins*.

☞ Wählen Sie das neu erstellt *plug-in* `org.example.mybundle` in der Liste aus **oder** wählen Sie *Add Required plug-ins*, um die Abhängigkeiten innerhalb der Startkonfiguration automatisch auflösen zu lassen.

☞ Setzen Sie die Option *Add new workspace plug-ins to this launch configuration automatically*, um diesen Fehler in den nächsten Schritten zu vermeiden.

☞ Starten Sie Ihre Anwendung mit *Run*.

☞ Führen Sie den Befehl `ss` auf der OSGi-Konsole aus.

In der ausgegebenen Liste werden Sie jetzt auch das neue *bundle* `org.example.mybundle` finden, allerdings ist es im Gegensatz zu `org.example.myapp` mit der Kennzeichnung «`LAZY`» versehen. Der Grund dafür ist, dass bisher lediglich eine Abhängigkeit zu diesem *bundle* definiert wurde, aber noch keine echte Verwendung des *bundle* stattgefunden hat. Die Kennzeichnung in der Liste ist also aus Sicht der Laufzeitumgebung die Angabe, dass das *plug-in* gefunden wurde und bei Bedarf geladen werden kann, aber eben noch kein Bedarf bestand (sog. *lazy loading*).

Wenden wir uns nun der einzigen Klasse des *plug-in* `org.example.mybundle` zu, dem bereits angesprochenen *Activator*. Zu jedem *plug-in* kann es maximal einen aktiven *Activator* geben, es gibt aber auch *plug-ins* ohne *Activator*. Die Verknüpfung zwischen den *plug-in*-Definitionsdaten und der Klasse findet im *plug-in* auf der Editor-Seite *Overview* statt. Im Anlage-Dialog für das *plug-in*-Projekt wird der *Activator* beschrieben als *a Java class that controls the plug-in's life cycle*; diese Aufgabe wird durch die beiden Methoden `start()` und `stop()` übernommen. Wir können diese Funktion testen, indem wir diese Methoden jeweils um eine kurze Nachrichtenausgabe erweitern:

```
public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;
    getLog().log(new Status(IStatus.OK, PLUGIN_ID, "Hello, world."));
}

public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);
    getLog().log(new Status(IStatus.OK, PLUGIN_ID, "Goodbye, world."));
}
```

Aus diesen Zeilen kann man übrigens noch einige weitere wichtige Informationen gewinnen: Das eingebaute Logging verwendet `Status`-Objekte (genauer gesagt, Instanzen einer beliebigen Klasse, die `IStatus` implementiert); die Log-Instanz erhält man über den *Activator*. Die *plug-in*-ID, die wir bisher nur in den Definitionsdaten gesehen haben, wird in der Regel als konstantes Attribut im *Activator* hinterlegt.

☞ Fügen Sie die o. a. Zeilen in den *Activator* von `org.example.mybundle` ein.

☞ Starten Sie Ihre Anwendung erneut und beobachten Sie die Konsole.

In der Regel wird die erwartete Meldung ausbleiben. Das liegt an dem bereits erläuterten *lazy loading* der *bundles* – da `org.example.mybundle` noch nicht wirklich verwendet wurde, wurde es von der Laufzeitumgebung auch noch nicht gestartet. Das können wir aber manuell veranlassen:

☞ Geben Sie auf der Konsole `start org.example.mybundle` ein.

☞ Geben Sie auf der Konsole `stop org.example.mybundle` ein.

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → Runtime overview → The runtime plug-in model → Plug-ins and bundles*
- [CR06], Kapitel 3 – Eclipse Infrastructure
- [Dau07], Kapitel 3.1 – Die Anatomie eines Plugins
- [ML05], Kapitel 26 – OSGi Essentials
- [SJB08], Kapitel 2 – Das Eclipse Plug-in-Konzept
- [WHKL08]

4 Dialogprogrammierung

Wir verlassen nun die Bundle-Definition für eine Weile und wenden uns der Oberfläche unserer Beispielanwendung zu. Die Anwendung (Abbildung 2) besteht offensichtlich aus zwei Teilbereichen, einer fixen Baumstruktur und einer aus mehreren einzelnen Elementen zusammengesetzten Nachrichtenanzeige, die mehrfach erscheinen kann und auch vom Anwendungsfenster abgekoppelt werden kann. Wie werden diese Obefflächenkomponenten verwendet, zusammengesetzt und in die Anwendung eingebracht?

Ausgangspunkt für die Oberflächen-Entwicklung ist das *Standard Widget Toolkit* (SWT), das die grundlegenden Dialogelemente zur Verfügung stellt. Das SWT ist eine standardisierte Bibliothek, die eine einheitliche und weitestgehend plattformneutrale Entwicklung grafischer Oberflächen erlaubt, dabei allerdings auf die jeweiligen plattformspezifischen Basisfunktionen zurückgreift. Damit ist gewährleistet, dass sich – rein aus Sicht der Oberflächenprogrammierung – eine SWT-basierte Anwendung unter Windows genauso aussieht und auch weitestgehend so reagiert wie eine native Windows-Anwendung, die gleiche SWT-Anwendung sich unter OS X allerdings verhält wie eine native OS X-Anwendung.

Die Entwicklung mit SWT basiert auf einer Reihe vordefinierter Klassen, die sogenannte *widgets* bereitstellen. Wenn Sie die *Eclipse SDK Examples*[8] installiert haben, steht Ihnen unter *Window* → *Show View* → *Other. . .* eine *SWT Controls View* (Abbildung 7) zur Verfügung, mit der Sie die Eigenschaften und Funktionen der einzelnen Widgets erforschen können[9].

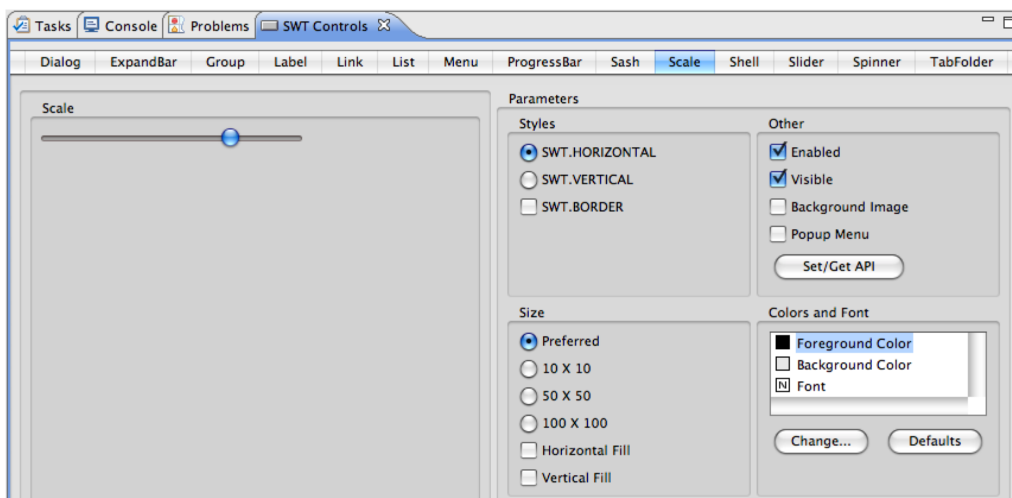

Abbildung 7: *SWT Controls View* zur Erprobung von SWT-Komponenten (Ausschnitt)

Bevor wir in die praktische Arbeit mit dem SWT einsteigen, sollten allerdings noch drei wichtige Klassen kurz erläutert werden:

[8] siehe Vorwort, Abschnitt „Benötigte Programme“

[9] OS X-Nutzer werden sich leider noch über `https://bugs.eclipse.org/bugs/show_bug.cgi?id=242163` ärgern.

`Display` ist eine der unsichtbaren Klassen des SWT und verwaltet die gesamte Anzeige (alle Fenster, Resourcen, ...).

`Shell` bezeichnet ein Fenster der Oberfläche - dazu zählt das Hauptfenster unserer Anwendung ebenso wie der äußerste Rahmen eines Dialogfensters

`Composite` ist eine der zentralen Basisklassen für andere Widgets. In der täglichen Praxis wird `Composite` aber insbesondere zur Bündelung anderer Dialogelemente verwendet

Zur Anordnung der einzelnen Komponenten verwendet SWT ein *layout management*-Verfahren, wie es auch bei anderen grafischen Oberflächen gebräuchlich ist. Einem Behälterelement wird dabei ein sogenannter *layout manager* zugewiesen, eine Instanz einer Klasse, die die Anordnung der einzelnen Elemente innerhalb des Behälters steuert. Je nach verwendetem *layout manager* können den einzelnen Elementen oder auch dem *layout manager* Parameter mitgegeben werden, die diesen Vorgang steuern. Eine vollständige Einführung würde an dieser Stelle zu weit führen, daher sei hier nur auf die Sekundärliteratur (z. B. [CR06], Kapitel 4 und 5) verwiesen.

Mit diesem Grundwissen können wir uns jetzt dem Aufbau der Nachrichtenanzeige nähern.

☞ Öffnen Sie die Klasse `View` und navigieren Sie zur Methode `createPartControl()`.

Sie werden hier einige der soeben erläuterten Konzepte wiederfinden: Zunächst werden einige `Composite`-Instanzen aufgebaut und mit Layout-Angaben versehen, dann werden in diese `Composites` Dialogelemente wie Beschriftungen, Links oder Textfelder eingefügt. Zur Demonstration können wir diese Anzeige um eine einfache Drucktaste erweitern:

```
...
l = new Label(banner, SWT.WRAP);
l.setText("Date:");
l.setFont(boldFont);
l = new Label(banner, SWT.WRAP);
l.setText("10:34 am");

Button b = new Button(banner, SWT.PUSH);
b.setLayoutData(new GridData(GridData.FILL, GridData.CENTER, true, false, 2, 1));
b.setText("Press Me");

// message contents
Text text = new Text(top, SWT.MULTI | SWT.WRAP);
...
```

☞ Fügen Sie die hervorgehobenen Programmzeilen in die Klasse `View` ein.

☞ Starten Sie die Anwendung.

Diese Drucktaste (Abbildung 8) hat jetzt natürlich noch keinerlei Funktion, aber das soll uns im Rahmen dieses Workshops nicht stören. Wenn Sie das ändern möchten, können Sie mit `addSelectionListener()` einen Ereignisbehandler registrieren, der bei Betätigung der Drucktaste benachrichtigt wird. Näheres hierzu finden Sie in der Dokumentation

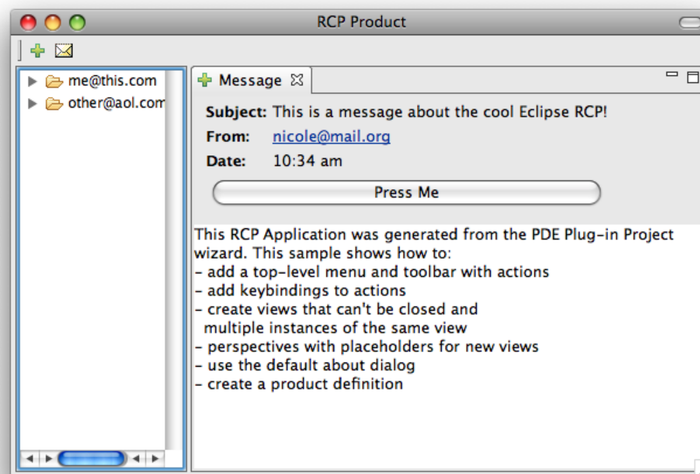

Abbildung 8: Anwendung mit eingefügter Drucktaste

dieser Methode und des verwendeten *interface*.

Bevor wir uns weiter mit der Struktur unserer Anwendung beschäftigen, soll mit JFace die zweite wichtige Bibliothek vorgestellt werden. Während SWT sich auf der Ebene des einzelnen Dialogelements bewegt, stellt JFace darauf aufbauend Klassen bereit, die häufig benötigte komplexere Dialogelemente abbilden und die Erstellung komplexer Dialoge deutlich vereinfachen. Dazu zählen Dialogfenster in verschiedenen Ausprägungen (die wir in einem späteren Abschnitt noch einmal kennenlernen werden), Wizards zum geführten Bearbeiten mehrschrittiger Aktionen und die sogenannten *JFace Forms*, eine optisch ansprechende Gestaltungsmöglichkeit für komplexere Oberflächen wie den *plug-in*-Editor. Außerdem stellt JFace sogenannte *viewer* zur Verfügung – Kapseln um einfache SWT-Elemente, die die Programmierung deutlich erleichtern. Ein Beispiel ist der `TreeViewer`, der im linken Teil der Anwendung zum Einsatz kommt. Die zugrundeliegende SWT-Klasse `Tree` stellt gewisse – rein technisch motivierte – Anforderungen an die Form der Eingabedaten, die nicht immer einfach durch die anzuzeigenden Objekte abgedeckt werden können. Das Konzept der *viewer* dreht daher das Programmiermodell um: Ein *viewer* kann alles anzeigen, solange der Verwender Hilfsklassen (sog. *provider*) bereitstellt, die bestimmte Fragestellungen zu Struktur und Anzeige der Objekte beantworten können (etwa „Mit welchem Text soll dieses Objekt angezeigt werden?“ oder „Hat dieses Objekt im Baum untergeordnete Elemente?“).

Mit einigen dieser komplexeren Element werden wir uns in späteren Abschnitten noch einmal beschäftigen. Für den Augenblick steht allerdings eine andere Frage im Vordergrund: Woher weiss die Rich Client Platform, dass die Dialogelemente an dieser Stelle des Fensters aufgebaut werden sollen, und woher kommt das offensichtlich ererbte Verhalten, das ein Verschieben, Anordnen und Abdocken der Nachrichtenanzeige erlaubt?

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *Standard Widget Toolkit*
- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *JFace UI Framework*
- [CR06], Kapitel 4 – The Standard Widget Toolkit
- [CR06], Kapitel 5 – JFace Viewers
- [Dau07], Kapitel 9 – SWT, JFace und das Forms API
- [SJB08], Kapitel 4 - Das User Interface

5 Oberflächengestaltung

Wir haben uns im vorherigen Abschnitt schon mit der Klasse `View` beschäftigt – genauer gesagt mit ihrem Innenleben, das den Aufbau der enthaltenen Dialogelemente bestimmt. Um die Integration dieser Klasse in die Anwendung zu verstehen, müssen wir ein weiteres Konzept einführen: *extensions*. Mit dieser Technologie ist es möglich, *plug-ins* untereinander zu verbinden, genauer gesagt: Sich in eine definierte Erweiterungsmöglichkeit (einen sog. *extension point*) eines anderen *plug-in* einzuklinken. Die *plug-in*-Architektur der Eclipse RCP besteht also nicht nur aus „Steckverbindungen" der einzelnen *plug-ins* in die Laufzeitumgebung, sondern bietet auch eine strukturierte und flexible Möglichkeit, die einzelnen Komponenten untereinander zu verbinden. Diese Möglichkeit wird auch verwendet, um die zwei *views* der Anwendung (die Klassen `View` und `NavigationView`) zu registrieren.

- ☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.myapp` und wechseln Sie auf die Editor-Seite *Extensions*.
- ☞ Expandieren Sie den Teilbaum unter `org.eclipse.ui.views`.
- ☞ Wählen Sie den Eintrag *Messages (view)* aus.

In der Detailanzeige auf der rechten Seite sehen Sie jetzt die Eigenschaften der *view*. Hier finden Sie den Namen und das Symbol zur Anzeige in der Titelzeile sowie den Namen der Klasse. Wichtig ist weiterhin der technische Schlüssel (ID) der *view*, der sich als öffentliches konstantes Attribut auch nochmals in der Klasse wiederfindet. Über diesen Schlüssel wird die *view* sowohl aus dem Programm als auch aus anderen *extensions* identifiziert.

Wir können unserer Anwendung nun – wiederum mit Hilfe einer Vorlage – eine weitere *view* hinzufügen.

- ☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.mybundle` und wechseln Sie auf die Editor-Seite *Extensions*.
- ☞ Rufen Sie mit *Add. . .* den Dialog zur Anlage einer *extension* auf.
- ☞ Wählen Sie die Registerkarte *Extension Wizards* und dort aus der Liste *Sample View* aus. Fahren Sie mit *Next* fort.
- ☞ Deaktivieren Sie die Optionen *Add the view to the java perspective* und *Add context help to the view*, um die Komplexität des Beispiels zu begrenzen. Belassen Sie die Vorgabewerte der anderen Optionen und schließen Sie den Vorgang mit *Finish* ab.

Anhand der Vorlage wurde jetzt eine *extension* erzeugt, die der vorhin bereits vorgestellten *extension* im *plug-in* `org.example.myapp` sehr ähnlich sieht[10].

- ☞ Kopieren Sie den Schlüssel der View (`org.example.mybundle.views.SampleView`) in die Zwischenablage.

[10] Die *Sample Category* können Sie für die Belange dieses Workshops ignorieren.

☞ Navigieren Sie zur Implementierung der neu erstellten *view*, indem Sie in den Detaildaten der *extension* auf die unterstrichene Bezeichnung *class* klicken.

Sie sehen, dass diese Klasse einen der im Abschnitt 4 angesprochenen JFace-*Viewer* verwendet, genauer gesagt einen `TableViewer`. In der Methode `createPartControl()` können Sie erkennen, dass dem Viewer mehrere der bereits erwähnten *provider* übergeben werden. Wichtig sind hier insbesondere der *content provider*, der die Eingabedaten in eine Liste anzuzeigender Objekte aufbereitet und der *label provider*, der die Texte und Symbole zur Anzeige ermittelt. Mit diesem viewer werden wir uns in einem späteren Abschnitt noch einmal beschäftigen.

Sie werden außerdem feststellen, dass diese Klasse – im Gegensatz zu den bisher betrachteten *view*-Klassen – über kein konstantes Attribut verfügt, über das sich zur Laufzeit der Schlüssel der *view* ermitteln ließe[11].

☞ Legen Sie eine neue Konstante zum Zugriff auf den Schlüssel an: `public static final String ID = "org.example.mybundle.views.SampleView";`

Wenn Sie jetzt die Anwendung starten, werden Sie noch keinerlei Veränderung bemerken. Wir haben zwar eine neue *view* angelegt, diese aber noch nicht in die Anwendung eingebunden. Dazu benötigen wir zunächst noch ein weiteres Konzept, dass Ihnen bereits aus dem Einsatz der Eclipse als Entwicklungsumgebung bekannt sein könnte. Die Anordnung der *views*[12] bezeichnet man als *perspective*. Wenn wir also unsere neue *view* in unserer Anwendung anzeigen wollen, müssen wir sie in die *perspective* einbinden.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.myapp` und wechseln Sie auf die Editor-Seite *Extensions*.

☞ Expandieren Sie den Teilbaum unter `org.eclipse.ui.perspectives`.

☞ Wählen Sie den Eintrag *RCP Perspective (perspective)* aus.

Wie Sie sehen, werden auch *perspectives* mit Hilfe von *extensions* deklariert – allerdings kommt auch hier eine Klasse zum Einsatz, die die eigentliche Arbeit zum Aufbau der Ansicht erledigt.

☞ Navigieren Sie zur Implementierung der *perspective*, indem Sie in den Detaildaten der *extension* auf die unterstrichene Bezeichnung *class* klicken.

In der generierten *perspective* werden die beiden *views* auf unterschiedliche Weisen eingebunden: die `NavigationView` soll genau einmal erscheinen und wird daher als *standalone view* eingebunden und darüberhinaus als „nicht schließbar“ markiert. für die mehrfach verwendbare Nachrichtenanzeige wird ein Anzeigebereich mit Registerkarten (*folder layout*) angelegt und neben einer initalen Instanz der *view* ein Platzhalter für weitere Instanzen angelegt. In diese *perspective* können wir jetzt auch unsere neu angelegte *view* einfügen:

[11]siehe auch `https://bugs.eclipse.org/bugs/show_bug.cgi?id=243087`

[12]sowie der Editoren und noch einiger anderer Einstellungen, die uns an dieser Stelle nicht weiter interessieren sollen

```
public void createInitialLayout(IPageLayout layout) {
    String editorArea = layout.getEditorArea();
    layout.setEditorAreaVisible(false);

    layout.addStandaloneView(NavigationView.ID, false, IPageLayout.LEFT,
                             0.25f, editorArea);
    IFolderLayout folder = layout.createFolder("messages", IPageLayout.TOP,
                                               0.5f, editorArea);
    folder.addPlaceholder(View.ID + ":*");
    folder.addView(View.ID);

    layout.getViewLayout(NavigationView.ID).setCloseable(false);

    IFolderLayout newFolder = layout.createFolder("table", IPageLayout.BOTTOM,
                                                  0.5f, editorArea);
    newFolder.addView(SampleView.ID);
}
```

☞ Fügen Sie die obigen Zeilen in die Klasse `Perspective` ein.

Sie werden feststellen, dass die Klasse jetzt einen Fehler aufweist: `SampleView` kann offensichtlich nicht aufgelöst werden. Die Ursache hierfür ist eine fehlende „Freigabe" der Klasse `SampleView`. Zwar haben wir bereits eine Abhängigkeit des *plug-in* `org.example.myapp` von `org.example.mybundle` deklariert, die semantisch als „`myapp` will etwas von `mybundle` benutzen" gelesen werden kann. Damit ist aber nicht automatisch der gesamte Inhalt des referenzierten *plug-in* verfügbar; vielmehr kann das *plug-in* selbst entscheiden, welche Pakete Verwendern zur Verfügung stehen sollen.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.mybundle` und wechseln Sie auf die Editor-Seite *Runtime*.

☞ Fügen Sie mit der Drucktaste *Add...* das Paket `org.example.mybundle.views` zur Liste der exportierten Pakete hinzu.

☞ Sichern Sie die geänderte *plug-in*-Definition und wechseln Sie zurück zur Klasse `Perspective`.

☞ Beheben Sie den Syntaxfehler mit Hilfe des *Quick Fix* oder indem Sie *Source* → *Organize Imports* ausführen.

☞ Starten Sie die Anwendung erneut.

Jetzt sehen Sie die neue *view* mit der tabellarischen Darstellung der Beispielelemente unterhalb der Nachrichtensicht (vgl. Abbildung 9). Es bleibt allerdings die Frage, wie denn die Anwendung überhaupt zu dieser *perspective* kommt, und was überhaupt die Anwendung ausmacht – damit werden wir uns im nächsten Abschnitt beschäftigen.

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *Advanced workbench concepts* → *Perspectives*
- [CR06], Kapitel 7 – Views

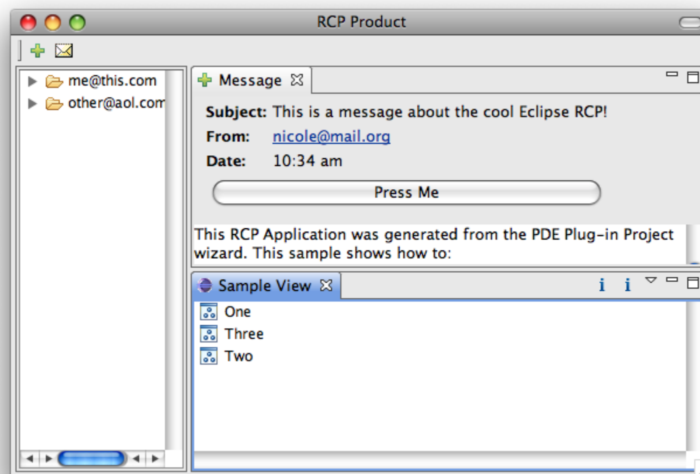

Abbildung 9: Anwendung mit neu angelegter *view*

- [CR06], Kapitel 10 – Perspectives
- [Dau07], Kapitel 3.5 – Die Benutzeroberfläche der Eclipse-Plattform
- [ML05], Kapitel 5.2 – Adding a Contacts View
- [ML05], Kapitel 16 – Perspectives, Views and Editors
- [SJB08], Kapitel 9 - ViewPart: Die Todo-Liste

6 Anwendung

Als Java-Entwickler lernt man sehr früh die Methode `public static void main()` kennen, mit der die Ausführung eines alleinstehenden Java-Programms beginnt. Für die Eclipse Rich Client Platform gibt es ein ähnliches Konzept, das allerdings naturgemäß etwas komplexer ausfällt und daher auf zwei Kapitel verteilt wird.

☞ Wählen Sie *Run* → *Run Configurations...* bzw. den entsprechenden Eintrag aus der *toolbar*.

☞ Wechseln Sie auf die Registerkarte *Main* (wenn nötig).

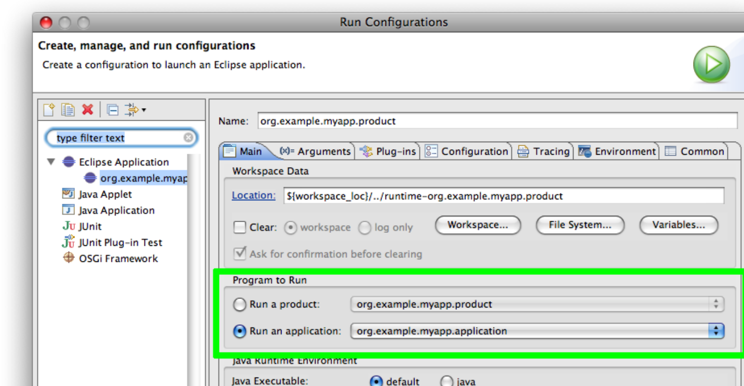

Abbildung 10: Startkonfiguration mit *product* und *application*

Etwa in der Mitte des Dialogs (Abbildung 10) sehen Sie, dass entweder ein *product* oder eine *application* ausgeführt werden kann. Durch die Verwendung des *RCP Mail Template* ist hier bereits das *product* voreingestellt.

☞ Ändern Sie die Einstellung auf *Run an application* und wählen Sie aus der Liste `org.example.myapp.application` aus.

☞ Starten Sie Ihre Anwendung.

Auf den ersten Blick werden Sie keine Änderungen bemerken, und tatsächlich sind die Unterschiede zwischen *product* und *application* für den Anwender auf den ersten Blick oft nicht zu erkennen. Wir wenden uns nun der *application* zu und werden das *product* im nächsten Abschnitt genauer betrachten.

Wie *views* und *perspective* werden auch *applications* mit Hilfe von *extensions* definiert.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.myapp` und wechseln Sie auf die Editor-Seite *Extensions*.

☞ Expandieren Sie den Teilbaum unter `org.eclipse.ui.perspectives`.

☞ Wählen Sie den Eintrag *org.example.myapp.Application (run)* aus.

Wie Sie sehen, besteht die Definition einer *application* aus wenig mehr als dem Verweis der zu verwendenden Implementierungsklasse.

☞ Navigieren Sie zur Implementierung der *application*, indem Sie in den Detaildaten der *extension* auf die unterstrichene Bezeichnung *class* klicken.

Die Klasse enthält zwei für uns relevante Methoden, `start()` und `stop()`. Letztere wird benötigt, da eine RCP-Anwendung auf vielerlei Arten beendet werden kann: Durch Schließen aller Fenster, durch eine Aktion innerhalb der Anwendung, durch das Betriebssystem oder auch durch ein Kommando auf der OSGi-Konsole. Für einige dieser Möglichkeiten muss eine Prozedur definiert werden, die eine laufende Anwendung geordnet beendet. In der Methode `start()` wird mithilfe einiger Methoden der RCP-Basisklasse `PlatformUI` eine Instanz der sog. *workbench* erstellt und gestartet. Die Behandlung des Rückgabewerts ist an dieser Stelle nicht von Interesse.

Als *workbench* bezeichnet man das äußerste Element einer RCP-Anwendung. Die *workbench* ist für die Verwaltung der einzelnen Fenster, Dialoge und anderer zentraler Elemente zuständig, ist allerding selbst nicht sichtbar. Der Anwender arbeitet mit *workbench windows*, die von der *workbench* verwaltet werden.

☞ Ändern Sie etwas an der Aufteilung Ihres Anwendungsfensters – verschieben Sie einige *view*-Grenzen oder schließen Sie eine oder mehrere *views*.

☞ Wählen Sie in Ihrer Anwendung *File* → *Open in New Window* aus.

Durch diese Aktion wird ein zweites gleichberechtigtes *workbench window* erzeugt. Die Anwendung wird erst beendet, wenn beide Fenster geschlossen werden. Das neue Fenster besitzt allerdings wieder das initiale Layout, das durch die *perspective* festgelegt wird. Damit stellt sich die Frage, wie die *perspective* mit der *application* verknüpft wird. Diese Aufgabe leistet die Klasse `ApplicationWorkbenchAdvisor`, die – wie der Name schon andeutet – der *workbench* verschiedene „Ratschläge“ zur Initialisierung und Verhaltensweise gibt. Einer dieser Ratschläge ist die zum Aufbau eines *workbench windows* zu verwendende *perspective*.

☞ Öffnen Sie die Klasse `ApplicationWorkbenchAdvisor`.

Leider befindet sich – anders als bei den *views* – der Schlüssel der *perspective* nicht in einem Attribut der Klasse `Perspective`. Das *RCP Mail Template* generiert vielmehr ein privates Attribut innerhalb der Klasse `ApplicationWorkbenchAdvisor`, was weder zur Übersichtlichkeit noch zur besseren Wartbarkeit beiträgt[13]. Die einzige Aufgabe, die zur Bereitstellung eines *workbench advisor* implementiert werden muss, ist die Benennung der initial zu verwendenden *perspective* durch Implementierung der abstrakten Methode `getInitialWindowPerspectiveId()` ist; es sind allerdings noch eine Reihe weiterer Möglichkeiten vorgesehen, auf die wir im Rahmen dieses Workshops nicht eingehen können.

[13] siehe auch `https://bugs.eclipse.org/bugs/show_bug.cgi?id=243086`

In der generierten Klasse `ApplicationWorkbenchAdvisor` findet sich allerdings noch eine weitere Methode namens `createWorkbenchWindowAdvisor()`. Über diese Methode wird ein Objekt erzeugt, das auf Ebene der *workbench windows* eine ähnliche Funktion hat wie die Klasse `ApplicationWorkbenchAdvisor` für die gesamte Anwendung.

☞ Öffnen Sie die Klasse `ApplicationWorkbenchWindowAdvisor`.

In dieser Klasse finden Sie einige Methoden, die Aussehen und Verhalten eines neu zu erstellenden *workbench window* beeinflussen, so zum Beispiel die Größe des Fensters oder die Verfügbarkeit der verschiedenen *toolbars* und Statuszeilen.

☞ Ändern Sie die Fenstergröße von 600×400 auf 800×600.

☞ Schalten Sie die *perspective bar* ein, indem Sie in die Methode `preWindowOpen()` die Anweisung `configurer.setShowPerspectiveBar(true);` einfügen.

☞ Starten Sie Ihre Anwendung erneut und prüfen Sie das Ergebnis.

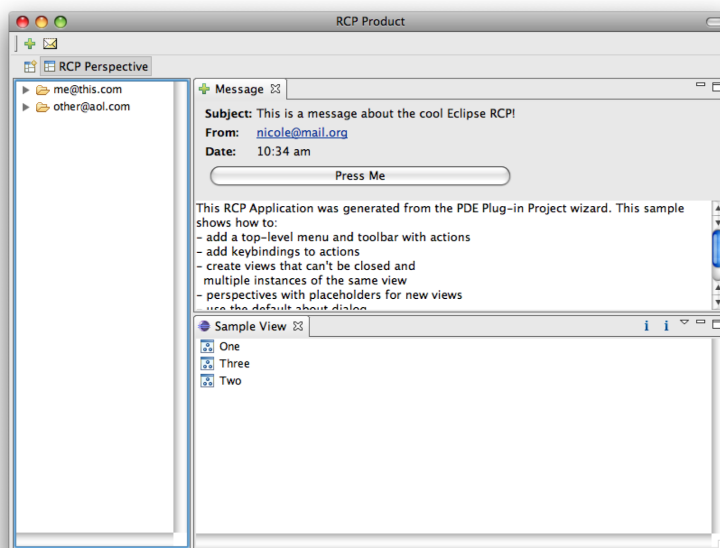

Abbildung 11: vergrößertes Anwendungsfenster mit *perspective bar*

An dieser Stelle wird auch der *action bar advisor* initialisiert, mit dem wir uns in einem späteren Abschnitt noch einmal beschäftigen werden. Zunächst wollen wir aber noch den Zusammenhang zwischen *application* und *product* näher untersuchen.

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *Building a Rich Client Platform Application* → *Defining a rich client application*
- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *Building a Rich Client Platform Application* → *Customizing the workbench*

- [ML05], Kapitel 4.2 – Tour of the Code
- [ML05], Kapitel 15 – Workbench Advisors
- [Dau07], Kapitel 4.2 – Applikationen

7 Paketierung und Export

Im vorangegangenen Abschnitt haben Sie zur Ausführung Ihrer Anwendung statt des vorher verwendeten *product* eine ebenfalls vorhandene *application* verwendet; vermutlich ohne einen nennenswerten Unterschied zu bemerken. Dennoch ist die Verwendung eines *product* kein überflüssiger Ballast, wie folgendes einfache Beispiel zeigt:

☞ Wählen Sie in Ihrer Anwendung *Help* → *About*.

Sie werden feststellen, dass das Fenster mit Ausnahme der Drucktasten am unteren Rand leer ist (Abbildung 12).

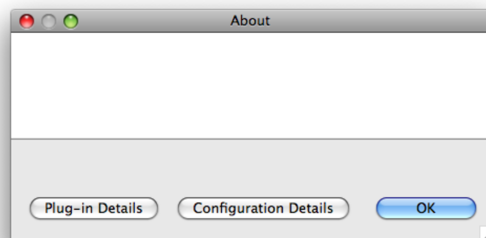

Abbildung 12: Informationsfenster ohne *product*

☞ Schließen Sie Ihre Anwendung.

☞ Wählen Sie *Run* → *Run Configurations...* bzw. den entsprechenden Eintrag aus der *toolbar*.

☞ Wechseln Sie auf die Registerkarte *Main* (wenn nötig).

☞ Ändern Sie die Einstellung auf *Run a product* und stellen Sie sicher, dass aus der Liste `org.example.myapp.product` ausgewählt ist.

☞ Starten Sie Ihre Anwendung und wählen Sie erneut *Help* → *About RCP Product*.

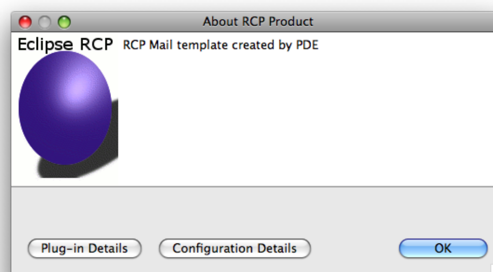

Abbildung 13: Informationsfenster mit *product*

Auf den ersten Blick fällt auf, dass sich der Name des Menüeintrags geändert hat. Darüberhinaus erscheinen in dem Dialog (Abbildung 13) jetzt ein (beispielhaftes) Produktlogo sowie ein erläuternder Text. In der Tat hat ein *product* in der Anwendung zunächst die Aufgabe, eine bereits funktionierende *application* um das sogenannte *branding* zu erweitern. Dementsprechend ist ein *product* immer eine rein deklarative Sammlung von Einstellungen, die keine eigene Programmlogik trägt, sondern immer auf eine *application* verweist. Die Definition erfolgt mit einer speziellen *extension*.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.myapp` und wechseln Sie auf die Editor-Seite *Extensions*.

☞ Expandieren Sie den Teilbaum unter `org.eclipse.core.runtime.products`.

☞ Wählen Sie den Eintrag *RCP Product (product)* aus.

An dieser Stelle erkennen Sie die Verknüpfung zwischen *product* und *application* sowie die Änderungen in den Oberflächeneigenschaften, die Sie beim Wechsel von *application* auf *product* beobachten konnten.

Die zweite wesentliche Aufgabe eines *product* neben dem *branding* ist die Steuerung des Paketierungs-Vorgangs. Um das demonstrieren zu können, benötigen wir eine weitere Steuerdatei:

☞ Wählen Sie *File* → *New* → *Product Configuration*[14].

☞ Wählen Sie das Projekt `org.example.myapp` als Zielordner und geben Sie als Dateinamen `myapp.product` ein.

☞ Achten Sie darauf, dass die Einstellung *Use an existing project* aktiviert ist und schließen Sie die Erstellung mit *Finish* ab.

Auf alle Möglichkeiten der Konfiguration einzugehen, würde den Rahmen dieses Workshops sprengen. Auf der ersten Seite allerdings werden Sie bereits einige bekannte Einstellungen wiederfinden. Wir nehmen nun eine weitere Einstellung vor, die nur bei Verwendung einer *product configuration* zur Verfügung steht.

☞ Wechseln Sie auf die Editor-Seite *Launching*.

☞ Tragen Sie in das Feld *Launcher Name* `myapp` ein.

☞ Sichern Sie alle Dateien.

Mit Hilfe dieser *product configuration* ist es jetzt möglich, den Paketierungsvorgang zu starten.

☞ Wählen Sie *File* → *Export...*.

☞ Wählen Sie aus der Liste *Plug-in Development* → *Eclipse product* und fahren Sie mit *Next* fort.

[14]Wenn Sie sich nicht in der *Plug-in Development Perspective* befinden, wählen Sie *File* → *New* → *Other...* und dann aus der Liste *Plug-in Development* → *Product Configuration*

☞ Wählen Sie die Datei `myapp.product` als *product configuration* aus.

☞ Geben Sie als *Destination Directory* ein Zielverzeichnis ausserhalb des *workspace* an.

☞ Starten Sie den Export mit *Finish*.

☞ Untersuchen Sie den Inhalt des gewählten Zielverzeichnisses.

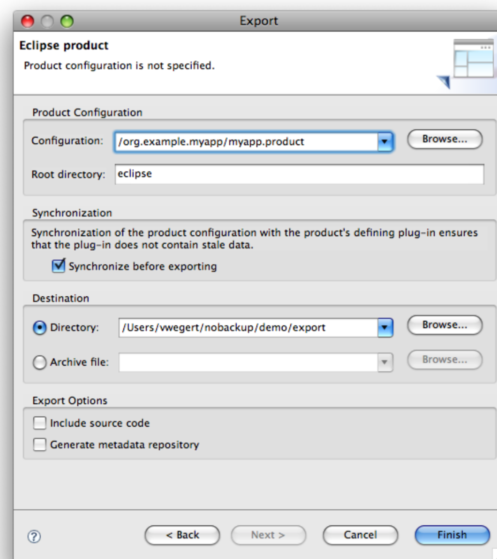

Abbildung 14: Export auf Basis der *product configuration*

Gerade zur genauen Steuerung des Paketierungsvorgangs, automatischen Aktualisierungen, zielplattformübergreifendem Export und derartigen Themen gäbe es noch einiges zu erforschen, was allerdings den Rahmen dieses Workshops definitv sprengen würde.

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide* → *Programmer's Guide* → *Packaging and delivering Eclipse based products*
- [CR06], Kapitel 18 – Features, Branding and Updates
- [Dau07], Kapitel 5 – Produktanpassung
- [ER04]
- [ML05], Kapitel 8 – Branding Hyperbola
- [ML05], Kapitel 9 – Exporting Hyperbola
- [SJB08], Kapitel 21 - Der Produkt-Editor
- [SJB08], Kapitel 25 – Branding

8 Menüs und Aktionen

Im Umgang mit der Beispielanwendungen sind Ihnen sicherlich schon die Menüeinträge und die Drucktasten der *toolbar* aufgefallen. In diesem Abschnitt wollen wir die Mechanismen betrachten, die für den Aufbau dieser Oberflächenelemente und die dadurch ausgelösten Abläufe verantwortlich sind. Wie nicht anders zu erwarten war, beginnen wir auch hier mit den generierten *extensions*.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.myapp` und wechseln Sie auf die Editor-Seite *Extensions*.

☞ Expandieren Sie die Teilbäume unter `org.eclipse.ui.commands` und *org.eclipse.ui.bindings*.

☞ Wählen Sie den Eintrag *Opens a mailbox (command)* aus.

Dieser Eintrag definiert ein sogenanntes *command*, eine Aktion, die ausgelöst und bearbeitet werden kann, allerdings weitestgehend indifferent hinsichtlich seiner Auslöser und Verwender ist. Das *command* ist offensichtlich einer Kategorie zugeordnet und – mit dem Eintrag *org.example.myapp.open (key)* einer Tastenkombination zugeordnet, allerdings finden sich in den *extensions* keine Angaben zum Aufbau des Menüs oder der *toolbar* oder gar zur auszulösenden Verarbeitung. Es ist zwar – wie wir gleich noch sehen werden – auch möglich, diese Verknüpfungen rein deklarativ über *extensions* vorzunehmen, das *RCP Mail Template* verwendet aber einen anderen Ansatz.

☞ Öffnen Sie die Klasse `ApplicationActionBarAdvisor`.

Wie der Kommentar bereits erkennen läßt, ist die Klasse für den Aufbau des Menüs und der *toolbar* (hier *coolbar* genannt) verantwortlich. In der Methode `makeActions()` werden offensichtlich Objekte aufgebaut, die mit den Menüeinträgen – also den auslösbaren Aktionen – korrespondieren und die Verarbeitung der Aktionen übernehmen sollen. Dabei wird kann auf eine Reihe vorgegebener Standardaktionen zurückgegriffen werden, die die Klasse `ActionFactory` bereitstellt. Die Methoden `fillMenuBar()` und `fillCoolBar()` verwenden diese Objekte, um die Oberflächenelemente aufzubauen und anzuordnen.

Während dieser Ansatz, die Oberfläche programmatisch zusammenzusetzen, für kleinere Anwendungen durchaus tragbar ist, wird er spätestens bei Anwendungen mit mehreren *plug-ins*, von denen einige vielleicht auch noch optional sind, sehr aufwendig in der Handhabung. Deshalb wollen wir nun einen Blick auf den bereits angesprochenen rein deklarativen Aufbau der Oberflächenelemente werfen.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.mybundle` und wechseln Sie auf die Editor-Seite *Extensions*.

☞ Rufen Sie mit *Add. . .* den Dialog zur Anlage einer *extension* auf.

☞ Wählen Sie die Registerkarte *Extension Wizards* und dort aus der Liste *“Hello, World” command contribution* aus. Fahren Sie mit *Next* fort.

☞ Behalten Sie die eingestellten Vorgabewerte bei und schließen Sie die Erstellung mit *Finish* ab.

☞ Sichern Sie alle geänderten Dateien und starten Sie Ihre Anwendung erneut.

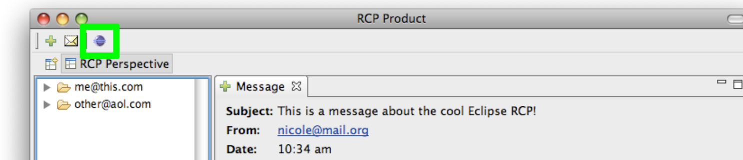

Abbildung 15: Anwendungsfenster mit neu eingefügter Aktion

Neben der in Abbildung 15 gezeigten Drucktaste werden Sie auch ein neues Menü namens *Sample Menu* mit einem Eintrag *Sample Command* bemerken. Sowohl die Drucktaste als auch der Menüeintrag und die verknüpfte Tastenkombination lösen eine einfache Nachricht aus.

☞ Untersuchen Sie die jetzt neuen *extensions* des *plug-in* `org.example.mybundle`.

Sie werden auch hier wieder die Einträge `org.eclipse.ui.commands` und *org.eclipse.ui.bindings* finden, die auch genau die gleichen Aufgaben haben wie zuvor: ein *command* zu definieren und es mit einem Tastaturkürzel zu verbinden.

☞ Wählen Sie den Eintrag unterhalb von `org.eclipse.ui.menus`, der mit `menu` beginnt und untersuchen Sie die untergeordneten Elemente.

Dieser Teilbaum definiert das neue Menü namens *Sample Menu* mit dem Eintrag *Sample Command*, der an dieser Stelle auch mit dem auszulösenden *command* verknüpft wird. Durch die Angabe `org.eclipse.ui.main.menu?after=additions` wird das neue Menü innerhalb des Hauptmenüs relativ weit rechts, aber noch vor dem Hilfe-Menü einsortiert.

☞ Wählen Sie den Eintrag unterhalb von `org.eclipse.ui.menus`, der mit `toolbar` beginnt und untersuchen Sie die untergeordneten Elemente.

Durch diesen Teilbaum wird eine neue *toolbar* angelegt – zu erkennen durch den senkrechten Trenner, an dem sich die gesamte Leiste verschieben läßt – und eine Drucktaste mit Symbol und *tooltip* eingefügt, die wiederum mit dem auszulösenden *command* verknüpft ist.

☞ Wählen Sie den Eintrag unterhalb von `org.eclipse.ui.handlers`.

Durch dieses Element wird eine Klasse benannt, die die ausgelöste Aktion ausführt. Wenn Sie einen Blick in diese Klasse werfen, werden Sie den Aufruf des `MessageDialog` vorfinden – übrigens eine JFace-Klasse.

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → Plugging into the workbench → Basic workbench extension points using actions*
- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → Plugging into the workbench → Basic workbench extension points using commands*
- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → Advanced workbench concepts → Menu and toolbar paths*
- [CR06], Kapitel 6 – Actions
- [ML05], Kapitel 6.1 – Adding to the Menus and Toolbar
- [ML05], Kapitel 12 – Adding Key Bindings
- [ML05], Kapitel 17 – Actions
- [SJB08], Kapitel 11 - Programmatische Actions
- [SJB08], Kapitel 12 - Deklarative Actions

9 Erweiterbarkeit (I)

Bisher haben wir *extensions* immer nur von der Verwender-Seite betrachtet – wir haben die Existenz passender *extension points* schlicht vorausgesetzt. In den beiden abschließenden Abschnitten des Workshops befassen wir uns mit der anderen Seite. Ausgangspunkt sei der Wunsch, in der Liste der in Abschnitt 5 angelegten *view* einen etwas dynamischeren Inhalt anzuzeigen. Dafür müssen wir uns zunächst kurz orientieren und verstehen, woher die bisherigen fixen Beispielelemente kommen, die wir ersetzen möchten.

☞ Öffnen Sie die Klasse `SampleView` im *plug-in* `org.example.mybundle`.

☞ Navigieren Sie in die innere Klasse `ViewContentProvider`.

Das Prinzip der *provider* hatten wir in Abschnitt 4 bereits kurz angesprochen. Aufgabe des *content providers* ist es, aus einer beliebigen – aus Sicht des *viewers* opaken – Eingabemenge einen Satz von Elementen zu erzeugen, die der *viewer* anzeigen soll. Die hier vorliegende Implementierung sorgt für die fixen Beispielelemente, die wir ersetzen wollen. Dazu müssen wir zunächst einen *extension point* definieren, der die Definition der anzuzeigenden Einträge erlaubt.

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.mybundle` und wechseln Sie auf die Editor-Seite *Extension Points*.

☞ Fügen Sie mit *Add...* einen neuen *extension point* hinzu.

☞ Wählen Sie als *Extension Point ID* `org.example.mybundle.entry` und als *Extension Point Name List Entry*.

☞ Schließen Sie die Erstellung mit *Finish* ab.

Jetzt wird das sogenannte Schema des *extension point* geöffnet. Wenn Sie diesen Begriff mit XML Schema assoziieren, liegen Sie damit gar nicht so verkehrt – tatsächlich weist die Schemadefinition eines *extension point* große Ähnlichkeit mit einem XML Schema auf.

☞ Wechseln Sie im Schema-Editor auf die Seite *Definition*.

☞ Wählen Sie *New Element* und geben Sie dem Element den Namen `entry`.

☞ Wählen Sie *New Attribute* und legen Sie ein neues Attribut mit dem Namen `id` an. Machen Sie die Angabe des Attributs verpflichtend, in dem Sie *Use* auf *required* umstellen.

☞ Wählen Sie *New Attribute* und legen Sie ein neues Attribut namens `description` an. Machen Sie die Angabe des Attributs verpflichtend, in dem Sie *Use* auf *required* umstellen. Kennzeichnen Sie das Attribut als übersetzbar (*translatable*).

☞ Wählen Sie den Eintrag `extension` in der Liste aus und legen Sie mit *New Sequence* eine neue Untergruppe an.

☞ Klicken Sie mit der rechten Maustaste auf den Eintrag `Sequence` und wählen Sie *New → entry*, um das zuvor angelegte Element hier zuzulassen.

☞ Ändern Sie die Multiplizität des Elements auf 1..*, indem Sie auf der rechten Seite *unbounded* aktivieren.

☞ Sichern Sie **alle** geänderten Dateien.

Zu diesem neu angelegten *extension point* können wir jetzt *extensions* erstellen. Der Übersichtlichkeit halber verwenden wir dazu ein neues *plug-in*.

☞ Wählen Sie *File* → *New* → *Project...* .

☞ Wählen Sie aus der Liste *Plug-in Project* und fahren Sie mit *Next* fort.

☞ Geben Sie als Projektnamen `org.example.myentries` ein. Belassen Sie die Standardwerte für die restlichen Einstellungen und fahren Sie mit *Next* fort.

☞ Ändern Sie ggf. den Namen des *plug-in* und achten Sie darauf, dass die Frage *Would you like to create a rich client application?* mit ***No*** beantwortet wird.

☞ Schliessen Sie die Erstellung mit *Finish* ab[15].

☞ Wechseln Sie auf die Editor-Seite *Extensions*.

☞ Wählen Sie *Add...*, um eine neue *extension* anzulegen und geben Sie als Suchbegriff *example* ein.

An dieser Stelle wird die Trefferliste der verfügbaren *extension points* leer sein, weil unterhalb der Liste die Einstellung *Show only extension points from the required plug-ins* aktiv ist. Da wir noch keine Abhängigkeitsbeziehung angelegt haben, erscheint unser neu angelegter *extension point* nicht in der Liste.

☞ Deaktivieren Sie die Option *Show only extension points from the required plug-ins*.

☞ Wählen Sie den Eintrag `org.example.mybundle.entry` aus der Liste und schließen Sie die Anlage mit *Finish* ab.

☞ Beantworten Sie die Frage, ob die o. a. Abhängigkeitsbeziehung anlegelgt werden soll, mit *Yes*.

Sie sehen, dass aufgrund der gewählten Multiplizität, die die Existenz mindestens eines *entry*-Elements vorschreibt, bereits ein Eintrag angelegt wurde.

☞ Ändern Sie die Beschreibung (*description*) des vorhandenen Eintrags.

☞ Fügen Sie einige weitere Einträge hinzu, indem Sie mit der rechten Maustaste auf den Eintrag `org.example.mybundle.entry` klicken und *New* → *entry* auswählen

☞ Sichern Sie alle geänderten Dateien.

Jetzt fehlt allerdings noch der Teil des Programms, der diese Einträge tatsächlich ausliest und in die Liste überführt. Dazu bauen wir innerhalb der `SampleView` eine Liste von Einträgen auf, die wir anschließend dem *viewer* übergeben. Die Listeneinträge werden dabei durch eine einfache lokale Klasse abgebildet:

[15] Wenn Sie an dieser Stelle versehentlich *Next* gewählt haben, achten Sie darauf, keine der auf der letzten Seite angebotenen Vorlagen zu verwenden.

```
class ListEntry {
    private String id;
    private String description;

    public ListEntry(String id, String description) {
        super();
        this.id = id;
        this.description = description;
    }

    public String getId() {
        return id;
    }

    public String getDescription() {
        return description;
    }
}
```

☞ Öffnen Sie die Klasse `SampleView` im *plug-in* `org.example.mybundle`.

☞ Fügen Sie die o. a. Klasse als lokale Klasse ein.

☞ Legen Sie ein neues Attribut `private List<ListEntry> entries` an.

Im nächsten Schritt muss die Liste mit den verfügbaren Einträgen befüllt werden. In unserem Beispiel erledigen wir das im Konstruktor der *view*:

```
public SampleView() {
    entries = new Vector<ListEntry>();
    for (final IConfigurationElement element:
         Platform.getExtensionRegistry().getConfigurationElementsFor(
                                         Activator.PLUGIN_ID, "entry")) {
        entries.add(new ListEntry(element.getAttribute("id"),
                                  element.getAttribute("description")));
    }
}
```

Hier wird vom Laufzeitsystem über die sogenannte *extension registry* eine Liste aller *extensions* zu einem bestimmten *extension point* abgerufen. Der *extension point* wird dabei aus der *plug-in*-ID und dem lokalen Namen des *extension points* zusammengesetzt, was eine gute Absicherung gegen eventuelle Umbenennungen des *plug-in* oder gar Kopieren von Quelltext darstellt. Aus diesen – kurzlebigen und recht umständlich zu verwendenden – `IConfigurationElement`-Instanzen bauen wir dann unsere eigene Liste von Einträgen auf.

Diese Liste von Einträgen müssen wir jetzt natürlich dem *viewer* übergeben. Dazu muss die Methode `createPartControl()` leicht angepasst werden:

```
public void createPartControl(Composite parent) {
    viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
    viewer.setContentProvider(new ViewContentProvider());
    viewer.setLabelProvider(new ViewLabelProvider());
    viewer.setSorter(new NameSorter());
    viewer.setInput(entries);
```

```
        makeActions();
        hookContextMenu();
        hookDoubleClickAction();
        contributeToActionBars();
    }
```

☞ Führen Sie die o. a. Änderungen an der Klasse `SampleView` durch.

Jetzt muss der *content provider* noch umgeschrieben werden, um die neuen Eingabedaten auch verarbeiten zu können:

```
class ViewContentProvider implements IStructuredContentProvider {

    private Object input;

    public void inputChanged(Viewer v, Object oldInput, Object newInput) {
        input = newInput;
    }

    public void dispose() {
        input = null;
    }

    public Object getElements(Object parent) {
        if (input instanceof List<?>) {
            return ((List<?>) input).toArray();
        }
        return null;
    }
}
```

☞ Führen Sie die o. a. Änderungen an der lokalen Klasse `ViewContentProvider` durch.

☞ Starten Sie Ihre Anwendung.

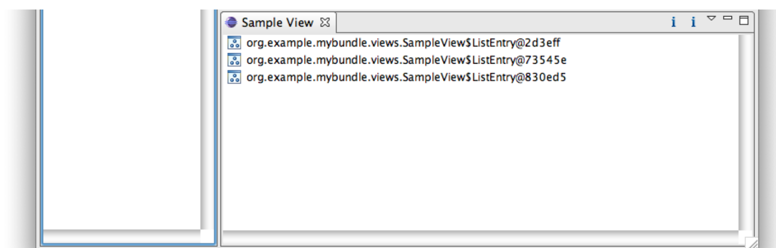

Abbildung 16: Liste mit dynamischem, aber häßlichen Inhalt

Der Inhalt der Liste (Abbildung 16) ist jetzt natürlich noch wenig ansprechend aufbereitet, weil der *label provider* noch nichts mit den neuen Eingabedaten anzufangen weiß. Das ist allerdings mit einer einfachen Redefinition möglich:

```
class ViewLabelProvider extends LabelProvider implements ITableLabelProvider {
```

```
    @Override
    public String getText(Object element) {
        if (element instanceof ListEntry) {
            return ((ListEntry) element).getDescription();
        }
        return super.getText(element);
    }

    ...
}
```

☞ Führen Sie die o. a. Änderungen an der lokalen Klasse `ViewLabelProvider` durch.

☞ Starten Sie Ihre Anwendung erneut.

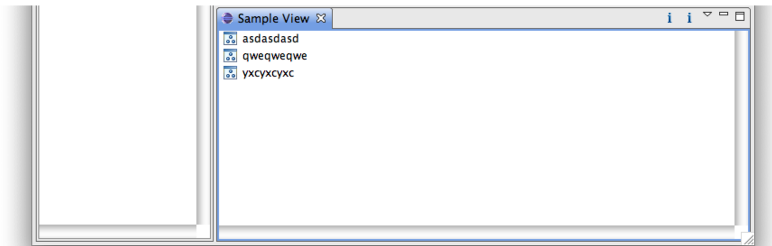

Abbildung 17: Liste mit ansprechenderem Inhalt

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → Runtime overview → The runtime plug-in model → Extension points and the registry*
- [CR06], Kapitel 17 – Creating New Extension Points
- [SJB08], Kapitel 15 – Extension Points

10 Erweiterbarkeit (II)

Mit dem im vorherigen Abschnitt angelegten *extension point* haben wir erprobt, wie Anwendungskomponenten deklarativ erweitert werden können. Aus dem Umgang mit bestehenden *extension points* wissen wir aber, dass auch eine Erweiterung durch Programmkomponenten möglich ist – so wie wir es beispielsweise bei *views* oder *perspectives* gesehen haben. Zum Abschluss des Workshops wollen wir diesen Aspekt noch einmal genauer betrachten. Da dieses Thema beliebig komplex werden kann, wählen wir zur Demonstration eine stark vereinfachte Aufgabenstellung aus: Bei Doppelklick eines Eintrags in der Liste soll durch das *plug-in*, das diesen Eintrag beigesteuert hat, eine beliebige Programmlogik (in unserem Fall: die Anzeige eines Dialogfensters) ausgeführt werden.

☞ Doppelklicken Sie in Ihrer Anwendung auf einen Eintrag der Liste.

Wie Sie sehen, wird jetzt schon eine Aktion ausgeführt – allerdings geschieht dies lokal innerhalb der Klasse `SampleView`. Verantwortlich dafür ist der *listener*, der in der Methode `hookDoubleClickAction()` registriert wird. Um dieses Verhalten jetzt erweiterbar zu gestalten, müssen wir zunächst die für die Kopplung zu verwendenden Typen anlegen:

☞ Öffnen Sie die Klasse `SampleView` im *plug-in* `org.example.mybundle`.

☞ Klicken Sie mit der rechten Maustaste auf die Klasse `ListEntry` und wählen Sie *Refactor → Extract Interface...*.

☞ Geben Sie als Namen `IListEntry` an und wählen Sie alle Methoden mit *Select All* aus. Schließen Sie den Vorgang mit *OK* ab.

☞ Öffnen Sie die neu erstellte Java-Datei und ändern Sie die Sichtbarkeit des *interface* `IListEntry` auf `public`.

☞ Klicken Sie mit der rechten Maustaste auf das Paket `org.example.mybundle.views` und wählen Sie *New → Interface*. Legen Sie ein neues *interface* `IActionListener` an.

☞ Fügen Sie dem *interface* eine neue Methode hinzu:

```
public interface IActionListener {
    public void handleDoubleClick(IListEntry entry);
}
```

☞ Sichern Sie alle geänderten Dateien.

Jetzt können wir das Schema des *extension point* erweitern:

☞ Öffnen Sie das Schema (Datei `schema/org.example.mybundle.entry.exsd` im *plug-in* `org.example.mybundle`) und wechseln Sie auf die Editor-Seite *Definition*.

☞ Wählen Sie in der Liste den Eintrag `entry` aus (nicht den Verweis unterhalb der *sequence*), sondern den Definitionseintrag am Ende der Liste.

☞ Wählen Sie *New Attribute* und legen Sie ein neues Attribut `actionListener` an. Ändern Sie den Typ auf *java* und nutzen Sie die *content assist*-Funktion, um unter *Implements* `org.example.mybundle.views.IActionListener` einzutragen (geben Sie `IActionL` ein und drücken Sie Strg+Leertaste).

☞ **Wichtig:** Sichern Sie jetzt das geänderte Schema!

☞ Öffnen Sie die *plug-in*-Definition des *plug-in* `org.example.myentries` und wechseln Sie auf die Editor-Seite *Extensions*.

☞ Wählen Sie einen der definierten Listeneinträge aus.

Im Editor sehen Sie jetzt auf der rechten Seite die neue Eigenschaft `actionListener`. Durch die Angabe der erforderlichen Schnittstelle im Schema kann der Editor jetzt die Anlage einer entsprechenden Klasse unterstützen:

☞ Klicken Sie auf den unterstrichenen Bezeichner `actionListener`.

☞ Wählen Sie das Paket `org.example.myentries` als Zielpaket aus (Eigenschaft *Package* im oberen Teil des Dialogs).

☞ Geben Sie einen Namen für die Klasse ein, etwa `MyListener` und schließen Sie die Anlage mit *Finish* ab.

☞ Implementieren Sie die Methode `handleDoubleClick()` wie folgt:

```
public void handleDoubleClick(IListEntry entry) {
    Shell shell = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell();
    MessageDialog.openInformation(shell, "MyEntries Plug-in",
            MessageFormat.format("Item {0} with ID {1} doubleclicked.",
                    entry.getDescription(), entry.getId()));
}
```

Jetzt haben wir zwar schon den Ereignisbehandler angelegt, aber noch nirgends verwendet. Dafür müssen wir in der Klasse *SampleView* noch einige Veränderungen vornehmen:

☞ Ändern Sie die interne Klasse `ListEntry` wie folgt:

```
class ListEntry implements IListEntry {
    private String id;
    private String description;
    private IActionListener listener;

    public ListEntry(String id, String description, IActionListener listener) {
        super();
        this.description = description;
        this.id = id;
        this.listener = listener;
    }

    ...

    public IActionListener getListener() {
        return listener;
```

```
        }
    }
```

☞ Führen Sie jetzt folgende Erweiterung des Konstruktors der Klasse `SampleView` durch:

```
public SampleView() {
    IActionListener listener;
    entries = new Vector<ListEntry>();
    for (final IConfigurationElement element:
         Platform.getExtensionRegistry().getConfigurationElementsFor(
                                          Activator.PLUGIN_ID, "entry")) {
        try {
            listener = (IActionListener)
                element.createExecutableExtension("actionListener");
        } catch (CoreException e) {
            listener = null;
        }
        entries.add(new ListEntry(element.getAttribute("id"),
                element.getAttribute("description"), listener));
    }
}
```

Damit wird der Ereignisbehandler von der Laufzeitumgebung instantiiert und in unserem `ListEntry`-Objekt abgelegt, allerdings noch nicht verwendet.

☞ Passen Sie die Implementierung der Methode `hookDoubleClickAction()` wie folgt an:

```
private void hookDoubleClickAction() {
    viewer.addDoubleClickListener(new IDoubleClickListener() {
        public void doubleClick(DoubleClickEvent event) {
            if (event.getSelection() instanceof IStructuredSelection) {
                final IStructuredSelection selection =
                        (IStructuredSelection) event.getSelection();
                if (selection.getFirstElement() instanceof ListEntry) {
                    final ListEntry entry = (ListEntry) selection.getFirstElement();
                    if (entry.getListener() != null) {
                        entry.getListener().handleDoubleClick(entry);
                    }
                }
            }
        }
    });
}
```

☞ Sichern Sie alle geänderten Dateien und starten Sie Ihre Anwendung erneut.

☞ Doppelklicken Sie auf den Eintrag, zu dem Sie einen Ereignisbehandler eingetragen haben.

weiterführende Literatur

- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → Runtime overview → The runtime plug-in model → Extension points and the registry*
- Eclipse Online-Hilfe, *Platform Plug-in Developer Guide → Programmer's Guide → JFace UI Framework → Viewers*
- [CR06], Kapitel 17 – Creating New Extension Points
- [SJB08], Kapitel 15 – Extension Points

Ausblick

Mit diesem Ausflug in die Welt der *extensions* beenden wir unseren Einsteiger-Workshop. Sehr viele interessante Konzepte können in einem Einsteiger-Workshop aufgrund des beschränkten Umfangs leider nicht behandelt werden, darunter so interessante Dinge wie Editoren, das Hilfesystem, Voreinstellungen (*preferences*), die asynchrone Hintergrundverarbeitung und vieles mehr. Wenn Sie neugierig geworden sind und gerne mehr über die Eclipse Rich Client Platform erfahren möchten, sollten Sie eines oder mehrere der im Literaturverzeichnis angegebenen Werke besorgen, um mehr über die Konzepte und vielfältigen Einsatzmöglichkeiten zu erfahren.

Abbildungsverzeichnis

1 Aktion *Launch an Eclipse application* 5
2 Anwendungsfenster nach Erstellung der Anwendung 6
3 Kommandozeilenparameter zur Verwendung der Konsole 6
4 Liste der *bundles* der Anwendung 7
5 Inhalt des Anwendungs-Projekts 8
6 Abhängigkeiten des Anwendungs-*plug-in* 9
7 *SWT Controls View* zur Erprobung von SWT-Komponenten (Ausschnitt) . . 13
8 Anwendung mit eingefügter Drucktaste 15
9 Anwendung mit neu angelegter *view* 20
10 Startkonfiguration mit *product* und *application* 21
11 vergrößertes Anwendungsfenster mit *perspective bar* 23
12 Informationsfenster ohne *product* 25
13 Informationsfenster mit *product* 25
14 Export auf Basis der *product configuration* 27
15 Anwendungsfenster mit neu eingefügter Aktion 29
16 Liste mit dynamischem, aber häßlichen Inhalt 34
17 Liste mit ansprechenderem Inhalt 35

Literatur

[CR06] Eric Clayberg und Dan Rubel: *Eclipse: Building Commectial-Quality Plug-ins* (Pearson Education, 2006) ISBN 0-321-42672-X

[Dau07] Berthold Daum: *Rich-Client-Entwicklung mit Eclipse 3.3* (dpunkt.verlag, 2007) ISBN 978-3-89864-503-4

[ER04] Andrew Eidsness und Pascal Rapicault: *Branding Your Application* (2004) URL `http://www.eclipse.org/articles/Article-Branding/branding-your-application.html`

[ML05] Jeff McAffer und Jean-Michel Lemieux: *Eclipse Rich Client Platform: designing, cofing and packaging Java applications* (Pearson Education, 2005) ISBN 0-321-33461-2

[SJB08] Heiko Sippel, Michael Jastram, und Jens Bendisposto: *Die Eclipse Rich Client Platform: Entwicklung von erweiterbaren Anwendungen mit RCP* (entwickler.press, 2008) ISBN 978-3-93908-491-4

[WHKL08] Gerd Wütherich, Nils Hartmann, Bernd Kolb, und Matthias Lübken: *Die OSGi Service Platform* (dpunkt.verlag, 2008) ISBN 978-3-89864-457-0