Ralf Ebert

Eclipse RCP

Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform 3.7

INHALT

KAPITEL 1

Einführung Eclipse RCP

Eclipse RCP (Rich Client Plattform) ist eine Plattform zur Entwicklung von Desktop-Anwendungen. Sie stellt ein Grundgerüst bereit, das um eigene Anwendungsfunktionalitäten erweitert werden kann.

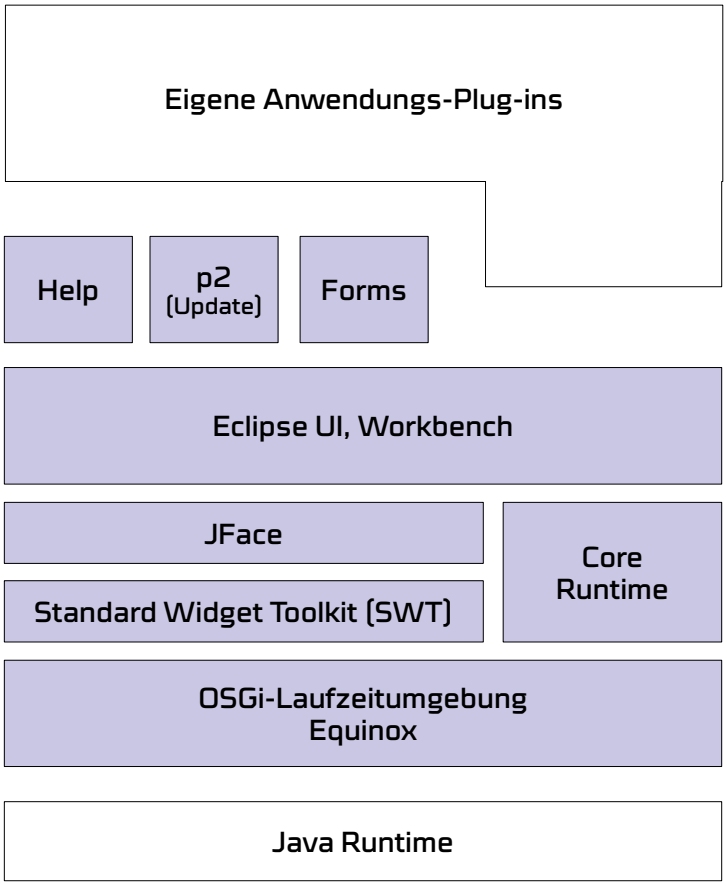
Entstanden ist Eclipse RCP aus der Eclipse IDE. Viele Aspekte und Komponenten der Eclipse IDE sind allgemeiner Natur und können auch in anderen Anwendungen Verwendung finden, z.B. der Workbench-Aufbau der Benutzeroberfläche, das erweiterbare Plug-in-System, die Hilfe-Komponente oder der Update-Manager. All diese allgemeinen Bestandteile der Eclipse IDE wurden 2004 extrahiert und werden seit Eclipse 3.0 als "Eclipse RCP" veröffentlicht. Sie dienen weiterhin als Grundlage für die Eclipse IDE, können nun jedoch auch separat verwendet werden.

Einige Stärken der RCP-Plattform sind:

- Eclipse RCP bietet ausgereifte Basiskomponenten für grafische Anwendungen, die sich in vielen Anwendungsfälle bewährt haben.
- Die Workbench stellt ein einheitliches, durchdachtes Bedienkonzept bereit. In den Grundzügen sind RCP-Anwendungen daher sehr konsistent, Endanwender müssen sich in die grundlegende Bedienung von RCP-Anwendungen nur einmal einarbeiten.
- Die Grundstrukturen für den Aufbau einer GUI-Anwendung sind durch das Framework vorgegeben und müssen nicht erst entwickelt werden.
- Die Plattform ist konsequent auf Modularität und Erweiterbarkeit hin ausgelegt. Dies ist insbesondere ein Vorteil für größere Anwendungen, die in kleinere Module aufgespalten werden müssen, um überschaubar zu bleiben.
- Erweiterungen auf der Grundlage der Eclipse RCP-Plattform können zusammen harmonieren, ohne einander zu kennen. So können Anwendungen in verschiedenen Ausstattungsvarianten ausgeliefert werden oder zunächst separat entwickelte Plug-ins später gemeinsam in einer Anwendung zusammengefasst werden.
- Benötigte Tools zum Schreiben von Erweiterungen für Eclipse RCP sind nahtlos in die Eclipse IDE integriert (*Eclipse for RCP/Plug-in Developers*).
- Die Eclipse-Projekte haben in den vergangenen Jahren eine sehr gute Stabilität und Konsistenz hinsichtlich der Veröffentlichung von neuen Versionen bewiesen. Jährlich erscheint ein neues Release (zuletzt im Juni 2010 Eclipse 3.7 mit Indigo), welches die vielen unabhängigen Eclipse-Projekte zu einem gemeinsamen Termin veröffentlichen. Mit 3.x Releases wird dabei die Rückwärtskompatibilität aller APIs gewährleistet. Das heißt, der Aufwand für das Update auf eine neue Version der Plattform ist überschaubar und gut planbar.

- Eclipse RCP steht unter der Eclipse Public License (EPL), eine Open-Source-Lizenz, die den kommerziellen Einsatz des Frameworks erlaubt. Insbesondere die Auslieferung von kommerziellen Erweiterungen zu EPL-Plug-ins ist erlaubt. Lediglich Änderungen an der Plattform selbst unterliegen einem *Copyleft*, d.h. dürfen nur unter der EPL verteilt werden.
- Die Eclipse Foundation verwaltet das *Intellectual Property (IP)* der Eclipse Code-Basis sehr strikt, d.h. es wird sichergestellt, dass sämtlicher Code von den Urhebern zu den Bedingungen der EPL bereitgestellt wird.
- Um die Eclipse-Plattform herum hat sich ein ausgeprägtes Ökosystem mit vielen Anbietern für Erweiterungen, Tools, Support und Schulungen entwickelt.

Folgende Komponenten machen Eclipse RCP aus:



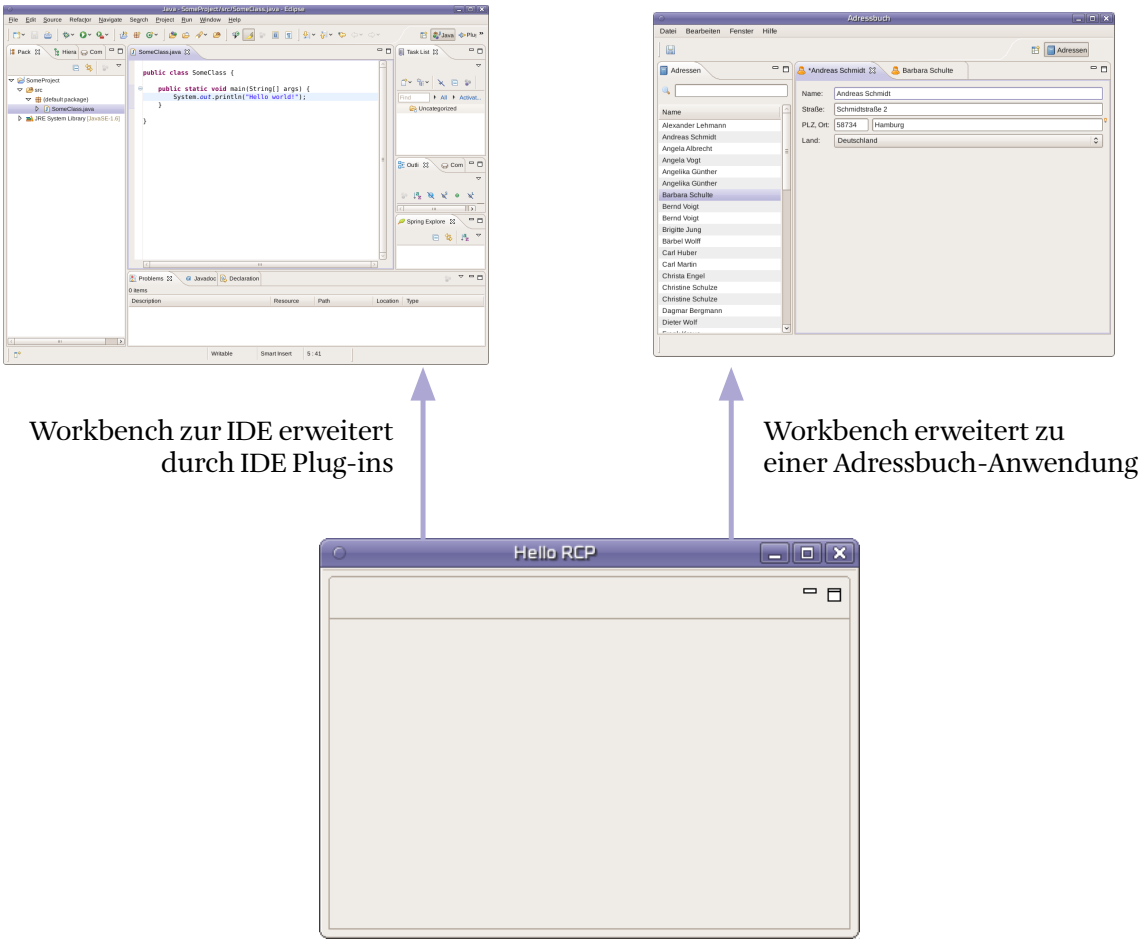
OSGi spezifiziert eine Java-Laufzeitumgebung, die die Ausführung von Modulen (sog. *Bundles* oder *Plug-ins*) ermöglicht. Eclipse implementiert diese Spezifikation mit *Eclipse Equinox*. Eclipse-Anwendungen bestehen aus einer Vielzahl von Plug-ins, die in *Equinox* ausgeführt werden.

Eclipse Core Runtime stellt allgemeine, nicht UI-bezogene Funktionalitäten für Eclipse-Anwendungen bereit. Es verwaltet u.a. den Lebenszyklus von Eclipse-Anwendungen, ist also für den Start und die Initialisierung der Anwendung verantwortlich.

Standard Widget Toolkit (**SWT**) ist das UI-Toolkit der Eclipse-Plattform, welches ein einheitliches Java für die nativen UI-Widgets des jeweiligen Betriebssystems bereitstellt. Es ist eine minimale Abstraktionsschicht für die UI-Widgets des Betriebssystems.

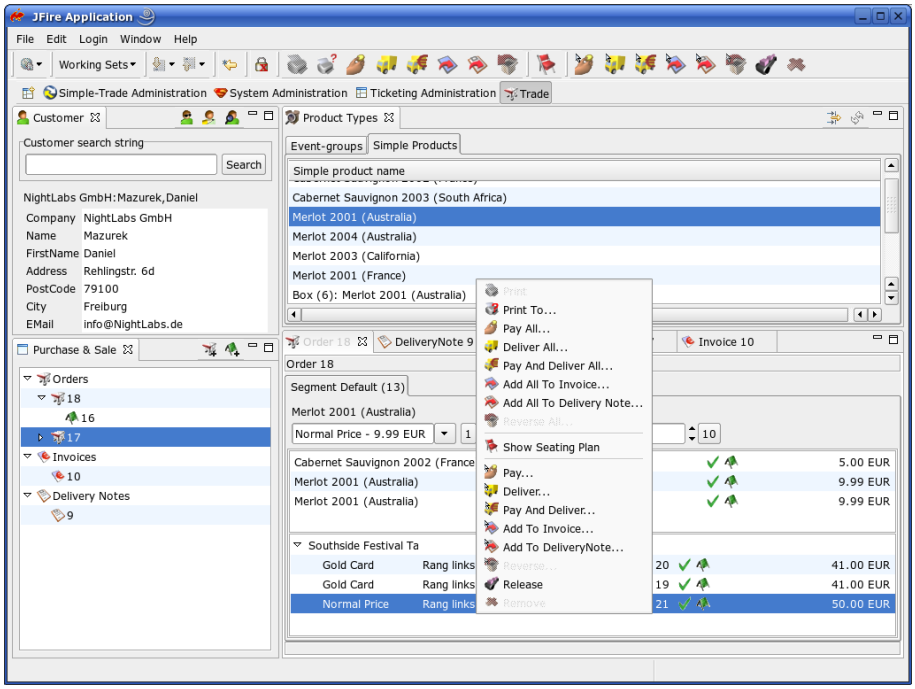
JFace stellt weitergehende Funktionalitäten wie die Befüllung von UI-Widgets mit Daten aus Java-Modellobjekten zur Verfügung.

Eclipse UI stellt die **Workbench** bereit. Darunter kann man sich die "Eclipse IDE ohne Inhalte" vorstellen – eine leere, grafische Anwendung, die die von der IDE bekannten Bedienkonzepte wie View- und Editor-Reiter, Perspektiven, Menüstrukturen etc. unterstützt. Diese leere Anwendung ist zur Befüllung durch Plug-ins bestimmt. So erweitern die Plug-ins der Eclipse IDE die Workbench um die Funktionalitäten, die für die Softwareentwicklung notwendig sind. Ihre eigenen Plug-ins auf der Grundlage von Eclipse RCP werden die Workbench hingegen mit Inhalten befüllen, die für Ihre Anwendung relevant sind:

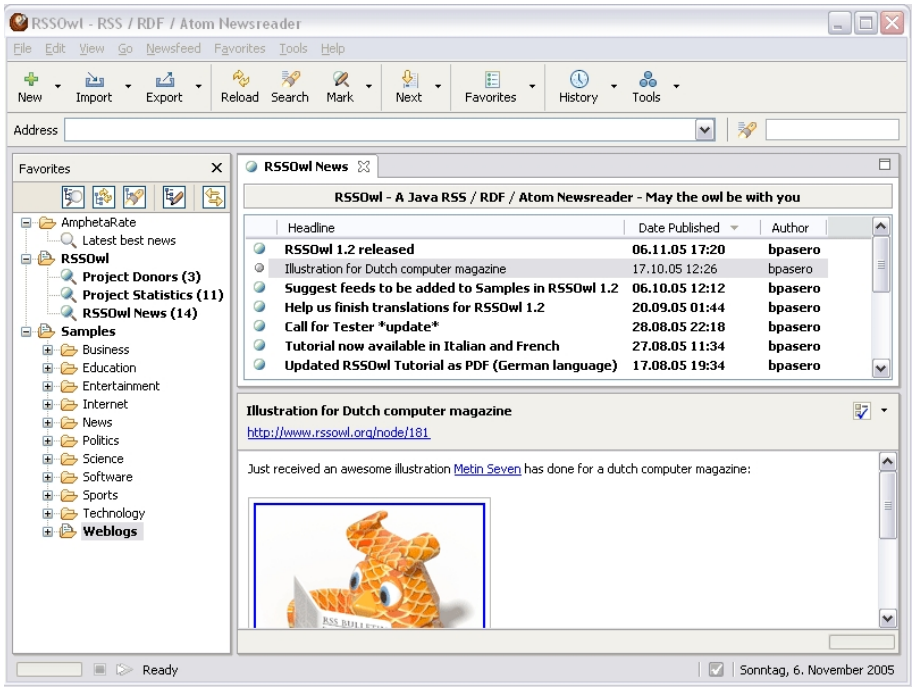


Im Folgenden einige Beispiele für Anwendungen, die mit Eclipse RCP realisiert sind.

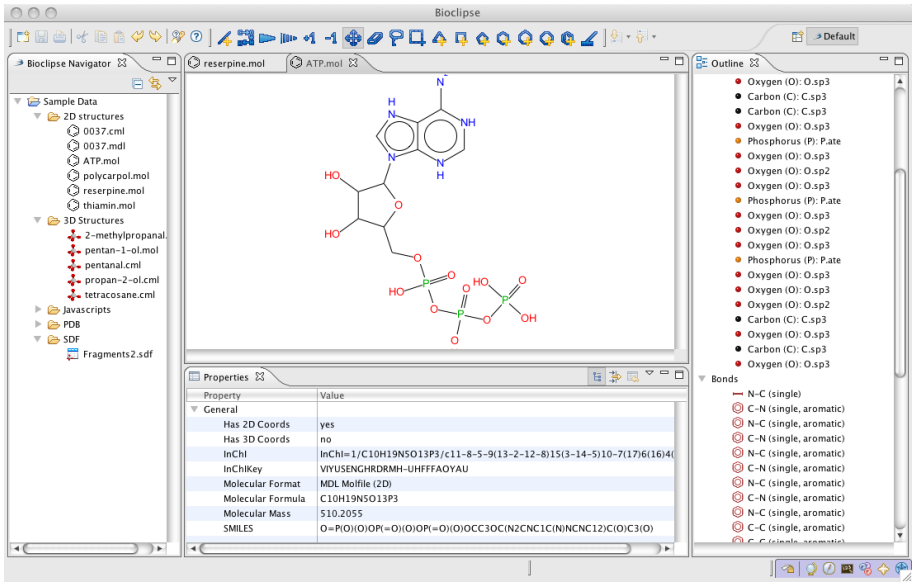
JFire, ein ERP-System:



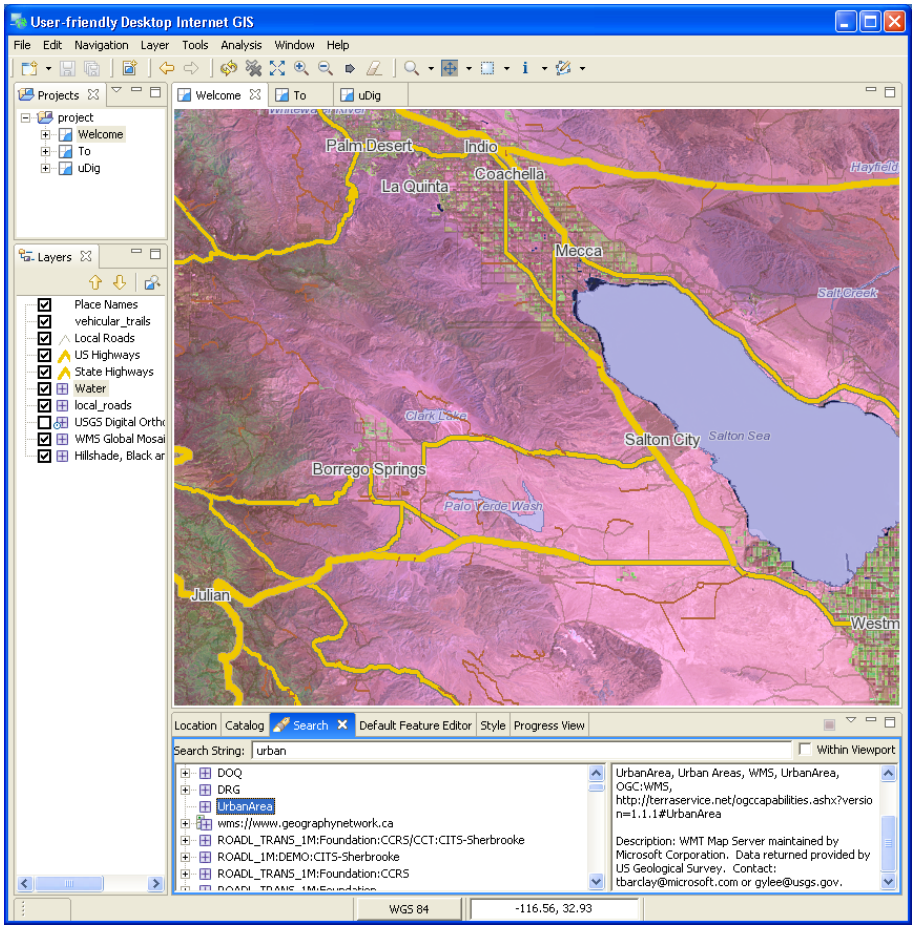
RSSOwl, ein Newsreader für RSS-Feeds:



Bioclipse, eine Software für Biowissenschaften:



uDig, ein geographisches Informationssystem:



Weitere Informationen

- Eclipse Wiki: Rich Client Platform
  http://wiki.eclipse.org/index.php/Rich_Client_Platform
- Rich Client Platform (RCP) applications
  http://www.eclipse.org/community/rcp.php
- Eclipse Bugzilla
  https://bugs.eclipse.org/
- Eclipse Project downloads
  http://download.eclipse.org/eclipse/downloads/
- Eclipse Forums
  http://www.eclipse.org/forums/
- Planet Eclipse
  http://www.planeteclipse.org/
- About the Eclipse Foundation
  http://www.eclipse.org/org/
- Eclipse Public License (EPL) Frequently Asked Questions
  http://www.eclipse.org/legal/eplfaq.php
- Eclipse RCP Buch
  http://www.rcpbuch.de/
- Eclipse RCP Tutorial
  http://www.vogella.de/articles/EclipseRCP/article.html
- Eclipse RCP examples
  http://www.ralfebert.de/blog/eclipsercp/rcp_examples/

TUTORIAL 1.1

Installation und Einrichtung der Eclipse IDE

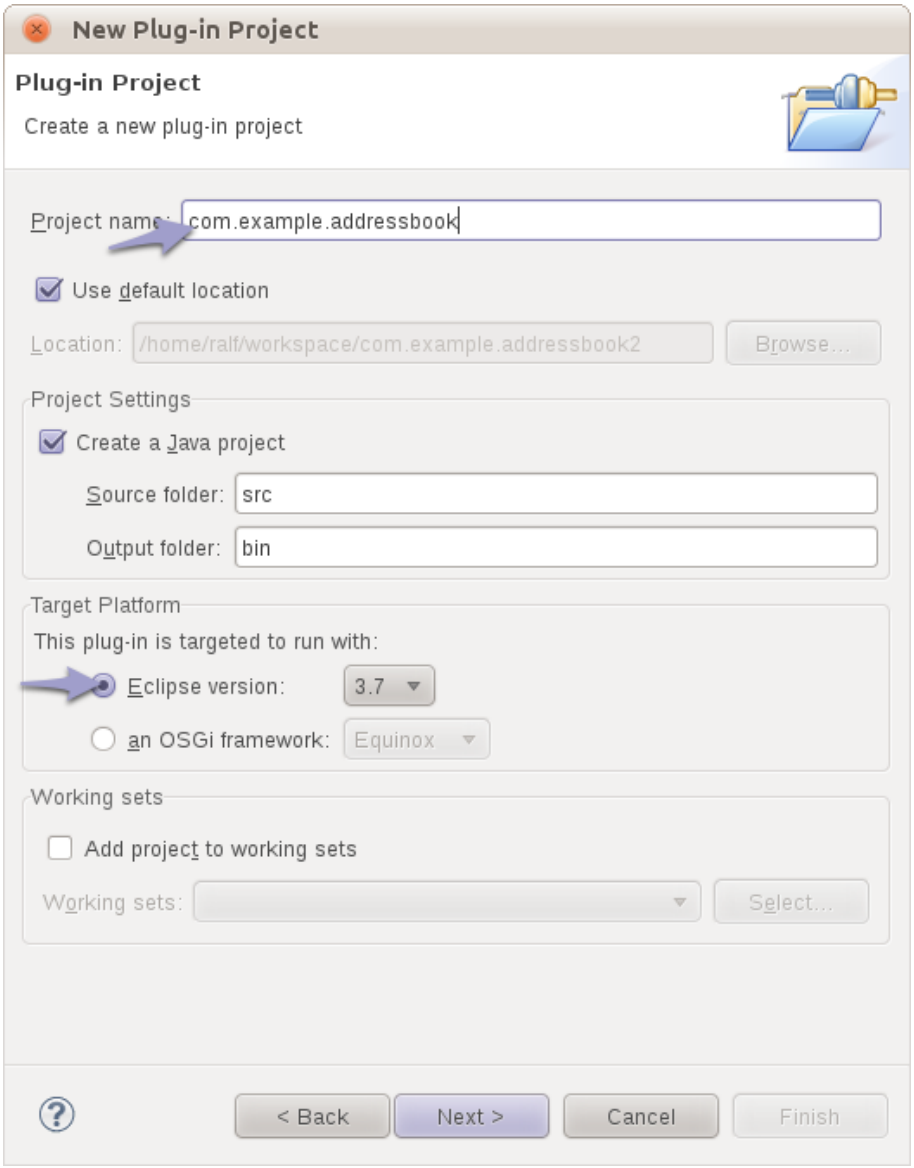
- ☐ Stellen Sie sicher, dass Sie min. Java JDK Version 1.6 installiert haben. Falls nicht, laden und installieren Sie es von http://www.oracle.com/technetwork/java/javase/downloads/.
- ☐ Laden, entpacken und starten Sie *Eclipse for RCP and RAP Developers 3.7.0* von http://www.eclipse.org/downloads/.
- ☐ Verwenden Sie für die Tutorials einen neuen, separaten Workspace-Ordner *addressbook* (im Workspace werden alle Eclipse-Einstellungen sowie standardmäßig alle Projektordner abgelegt, ein nachträglicher Wechsel des Workspaces erfolgt mit *File › Switch Workspace*).

TUTORIAL 1.2

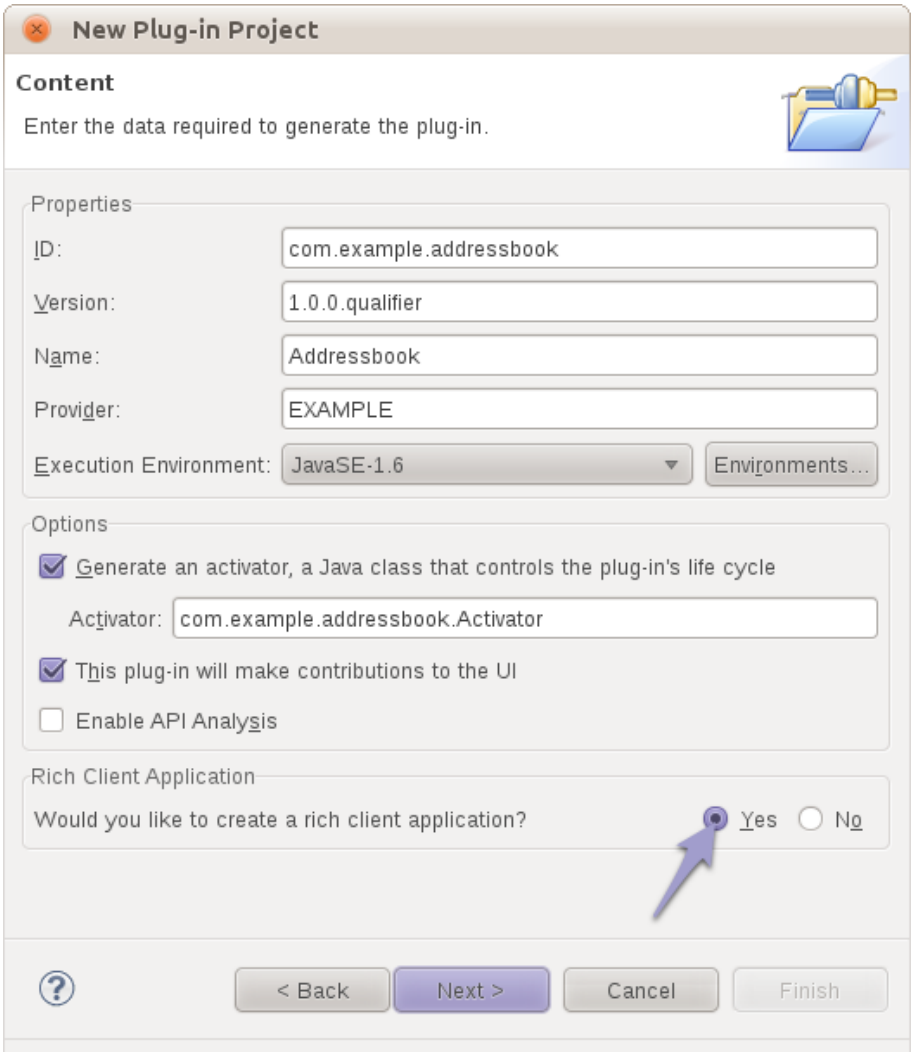
Neue RCP-Anwendung erstellen

☐ Erstellen Sie mit *File* › *New* › *Project* › *Plug-in Development* › *Plug-in project* ein neues Projekt.

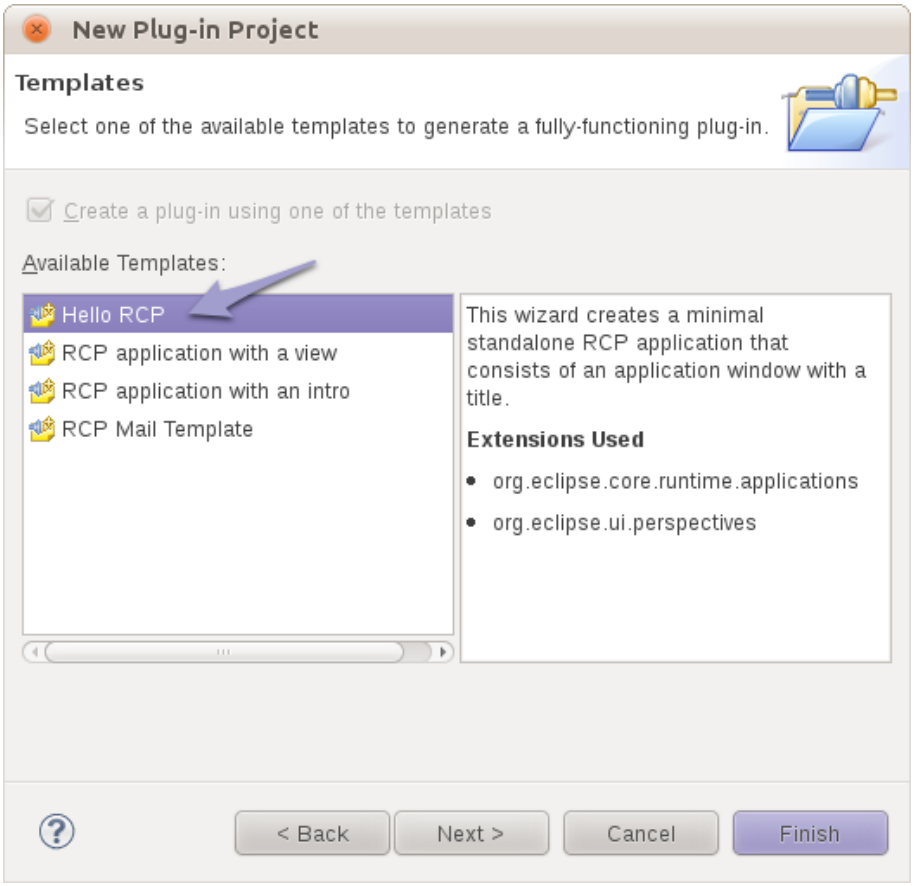
☐ Vergeben Sie *com.example.addressbook* als Projektname und stellen Sie sicher, das *Eclipse-Version* ausgewählt ist:



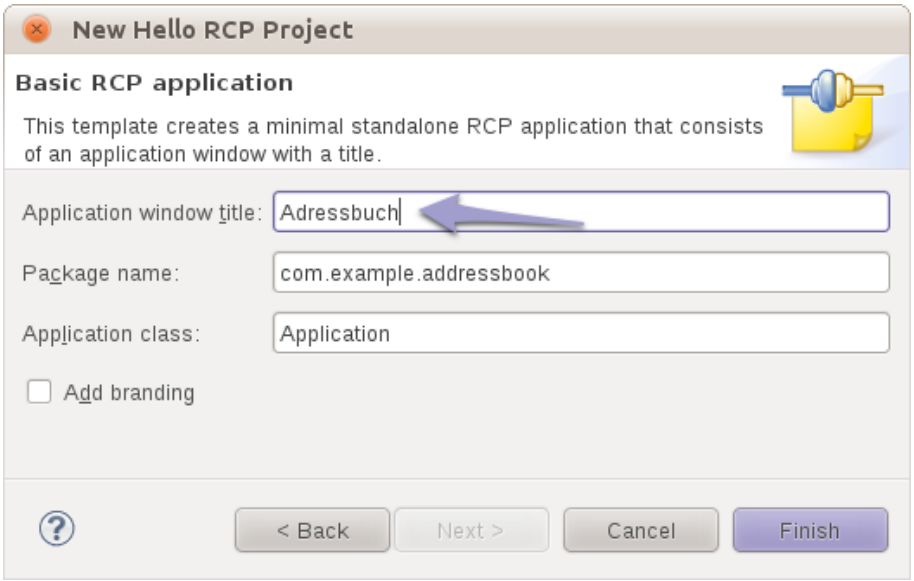
- ☐ Erstellen Sie eine Rich-Client-Anwendung (*No* würde eine Erweiterung zu einer bestehenden Anwendung erstellen):



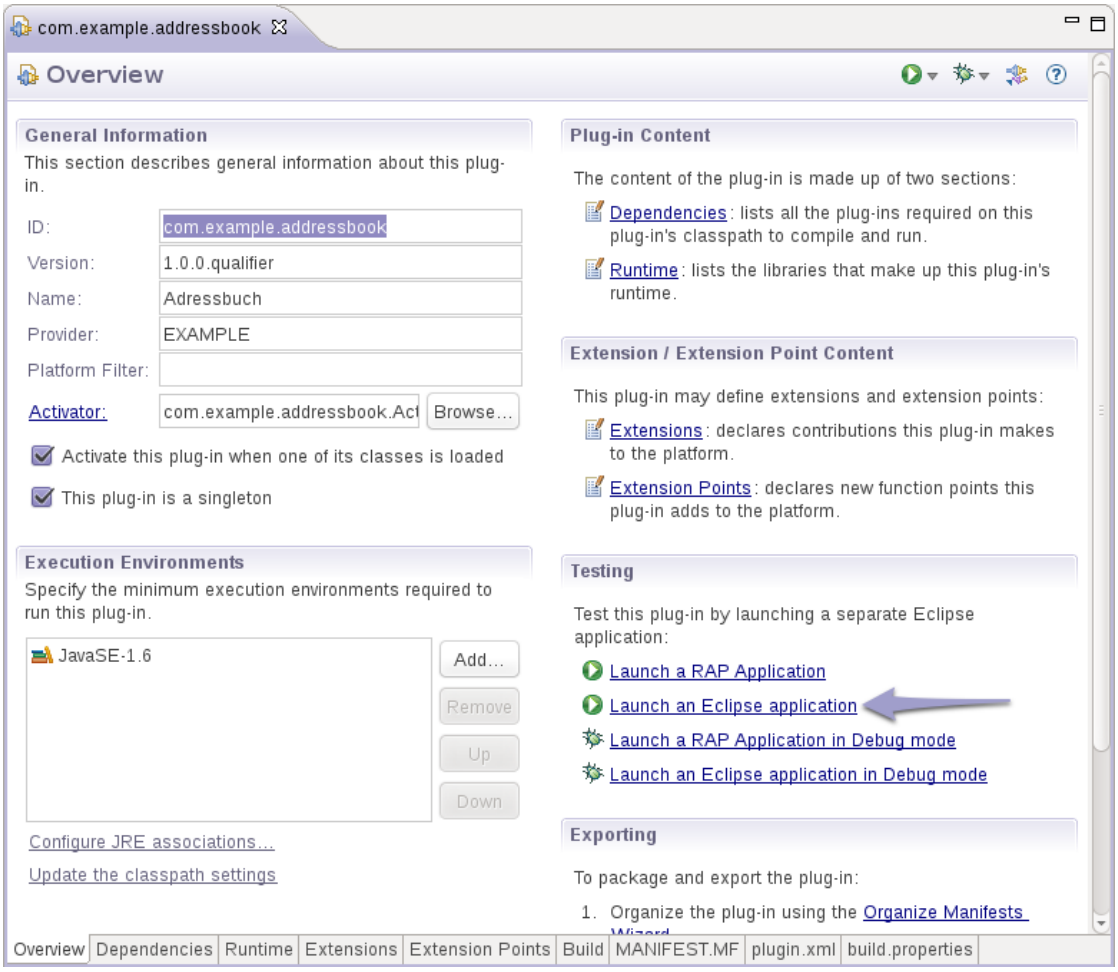
□ Wählen Sie das Template *Hello RCP*:



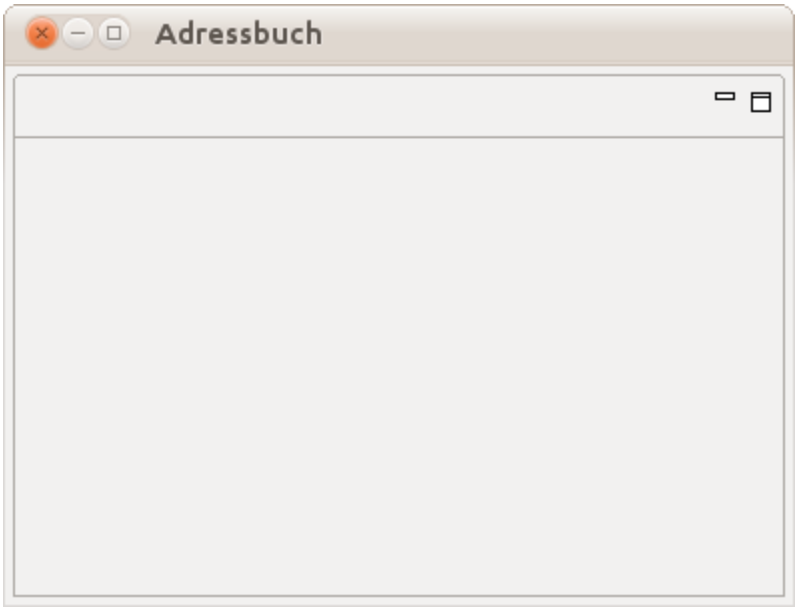
□ Vergeben Sie *Adressbuch* als Titel für das Applikationsfenster:



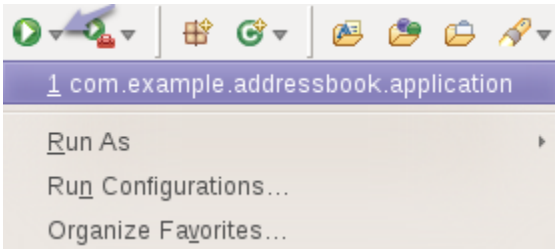
- ☐ Starten Sie die erstellte Anwendung in dem automatisch geöffneten Reiter *plugin.xml* mit *Launch an Eclipse application*:



☐ Die Anwendung sollte nun starten und ein leeres Fenster anzeigen:



☐ Durch den vorherigen Schritt wurde eine Startkonfiguration (auch *Launch/Run Configuration*) angelegt. Starten Sie die Anwendung erneut über die erstellte Startkonfiguration:



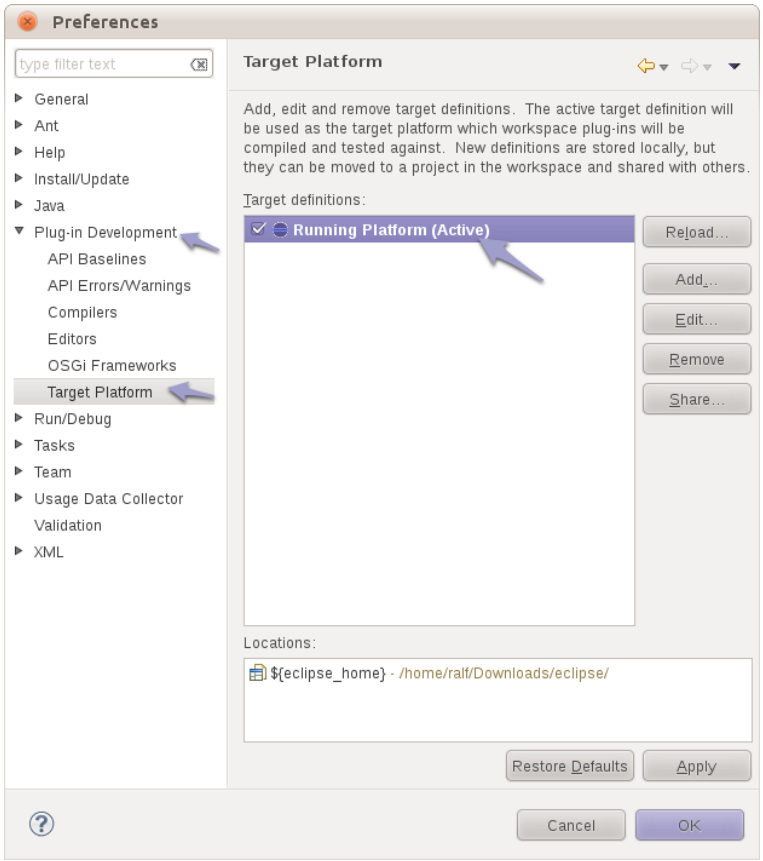
KAPITEL 2

Target Platform

Die Entwicklung von Eclipse-Anwendungen erfolgt für eine konfigurierbare Zielplattform (*Target Platform*). Alle Komponenten, die wir nicht selbst entwickeln, werden aus dieser Zielplattform geladen. Standardmäßig ist die Zielplattform die Eclipse IDE, d.h. Eclipse ist ideal für die Entwicklung von Erweiterungen für die Eclipse IDE selbst konfiguriert.

Die Eclipse IDE selbst ist keine gute Target Platform für die Entwicklung von RCP-Anwendungen. Denn so würden alle Komponenten der IDE auch in der RCP-Applikation zur Verfügung stehen. Zusätzliche Komponenten für die RCP-Anwendung müssten dann zwangsweise auch in die IDE installiert werden. Durch die Definition einer Target Platform kann explizit festgelegt werden, welche Komponenten in welcher Version für die Anwendungsentwicklung zur Verfügung stehen. Daher empfiehlt es sich, die Zielplattform explizit festzulegen.

Dies erfolgt global für den gesamten Eclipse-Workspace unter *Window* › *Preferences* › *Plug-in Development* › *Target Platform*. Standardmäßig ist die *"Running Platform"*, d.h. die IDE selbst, konfiguriert:



Eine Target-Plattform kann aus einem lokalen Ordner mit Plug-in-JARs zusammengestellt werden oder von dem Eclipse-Updatemanager *p2* heruntergeladen und bereitgestellt werden. Über *.target*-Dateien kann eine Target Plattform zwischen Entwicklern ausgetauscht werden. In einer solchen Datei wird eine Zusammenstellung von Softwarekomponenten definiert und festgelegt, wie diese bezogen werden sollen.

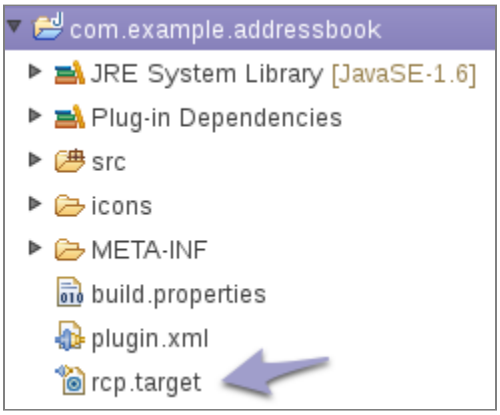
TUTORIAL 2.1

Target Plattform einrichten

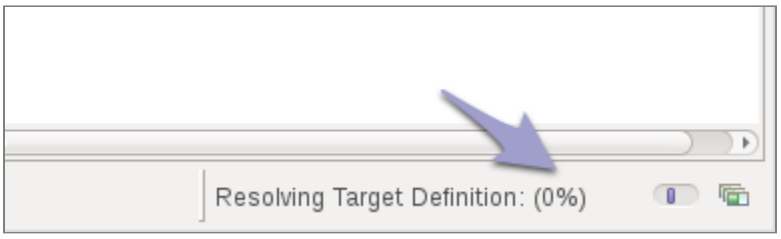
Für die folgenden Tutorials werden einige zusätzliche Komponenten benötigt, die nicht Bestandteil von Eclipse RCP sind. Zudem sollten wir als Basis für die Anwendung nicht die IDE verwenden und daher eine dedizierte Target Platform einrichten:

- ☐ Rufen Sie http://www.ralfebert.de/eclipse_rcp/target/ auf und laden Sie die *.target*-Datei für die jeweilige Eclipse-Version herunter. In dieser Target-Definition ist alles enthalten, was für die Tutorials in diesem Buch benötigt wird.

- ☐ Kopieren Sie die *.target*-Datei in das Projekt *com.example.addressbook*:



- ☐ Öffnen Sie die *rcp.target*-Datei. p2 lädt nun die Komponenten der Target Plattform. Der Download kann je nach Netzwerkverbindung 1-2 Minuten dauern - schließen Sie während des Downloads nicht den *rcp.target*-Reiter, da dies zum Abbruch des Downloads führt.



☐ Warten Sie bis der Download abgeschlossen ist, erkennbar daran, dass unter der URL die heruntergeladenen Komponenten (u.a. *Eclipse RCP SDK*) erscheinen:

☐ Aktivieren Sie die Target Plattform mit *Set as Target Platform*:

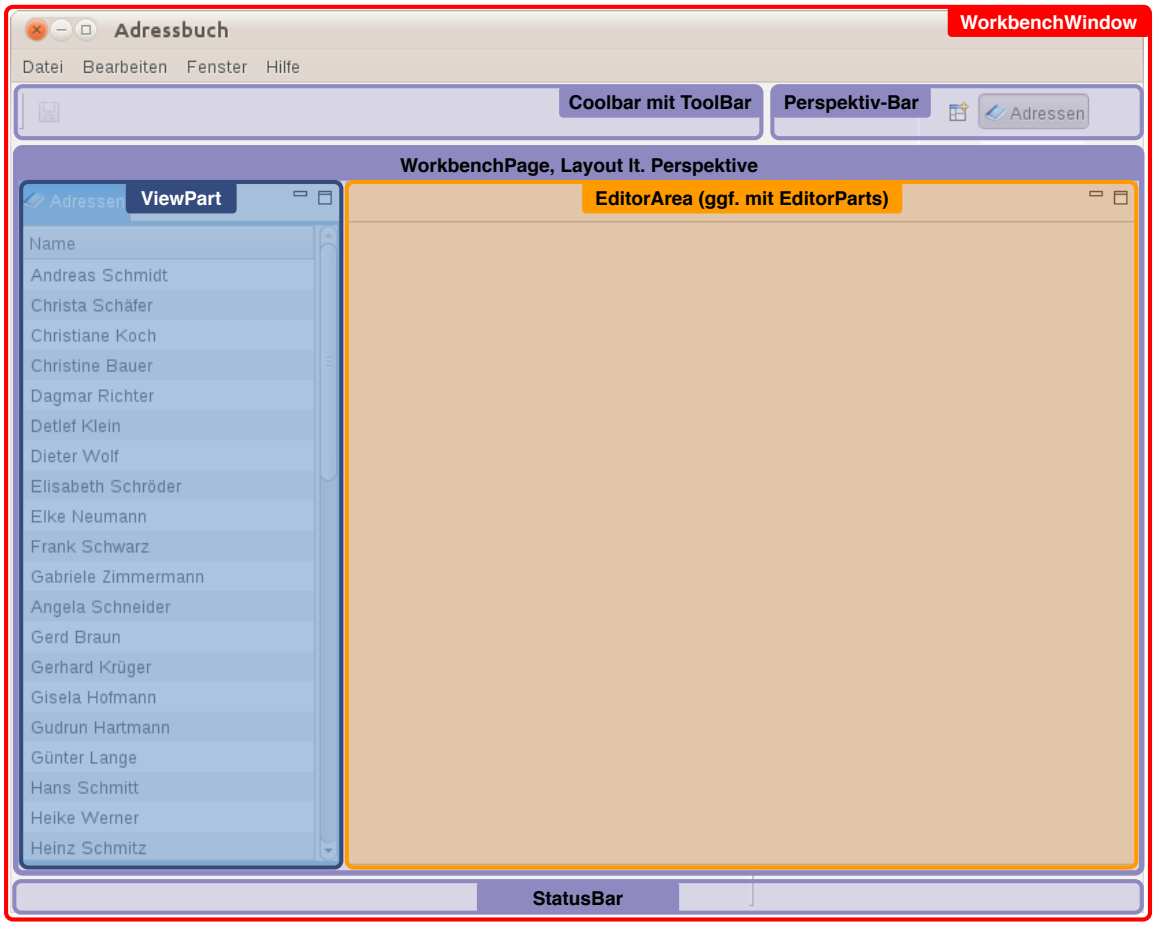
☐ Starten Sie die Adressbuch-Anwendung erneut um sicherzustellen, dass die Anwendung auch mit der soeben eingerichteten Target-Platform funktioniert.

KAPITEL 3

Workbench-Extensions für Views und Perspektiven

Die *Workbench* macht die grafische Benutzeroberfläche einer RCP-Anwendung aus. Der grundlegende Aufbau der Benutzeroberfläche ist dabei durch Eclipse RCP vorgegeben und ist durch die Anwendung lediglich mit Inhalten zu befüllen.

Ein Applikationsfenster wird als *Workbench Window* bezeichnet. Die Inhalte eines solchen Fensters werden von seiner *WorkbenchPage* verwaltet und über Reiterkarteien strukturiert, die als *Part* bezeichnet werden.



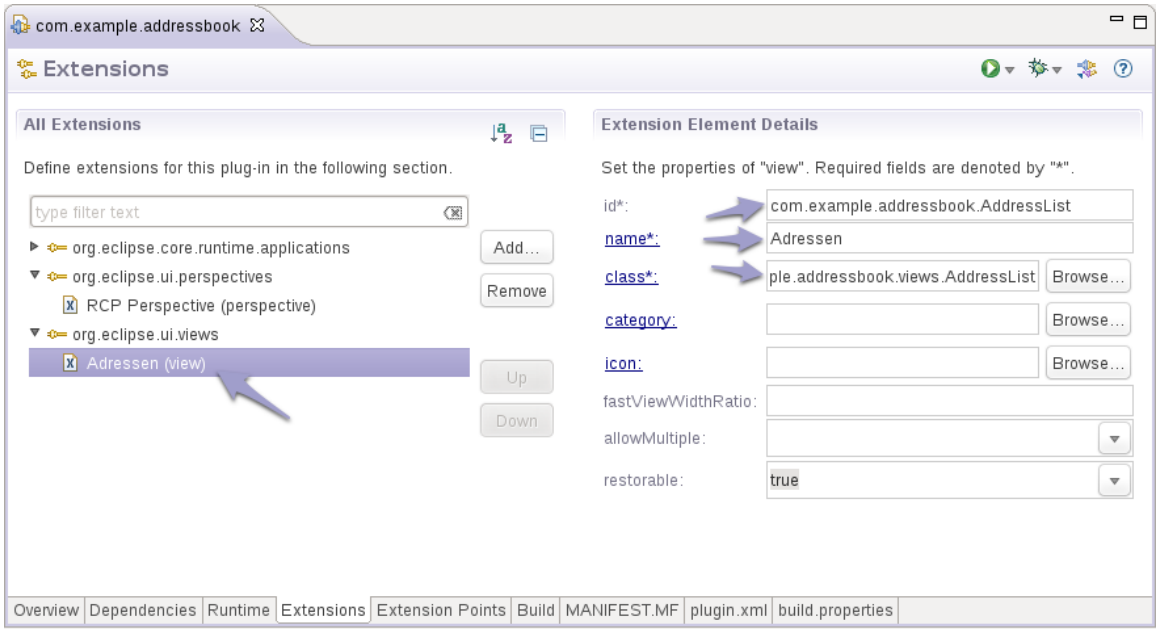
Es gibt zwei Arten von *Parts*, *Views* und *Editoren*. *Editor Parts* werden für die Dokumente der Anwendung verwendet und in der *Editor Area* der Anwendung angezeigt. *View Parts* sind zusätzliche Reiterkarteien, die um den Editierbereich herum angeordnet werden können. Die Perspektive beschreibt dabei die Anordnung dieser Reiterkarteien. Der Benutzer kann über die Perspektiv-Bar zwischen verschiedenen Perspektiven wechseln. Typisch sind Perspektiven für Tätigkeiten, die dann alle für diese Aufgabe benötigten Reiter enthalten (beispielsweise

“Mahnung schreiben”). Perspektiven können vorkonfiguriert mit der Anwendung ausgeliefert werden oder vom Benutzer verändert und angelegt werden.

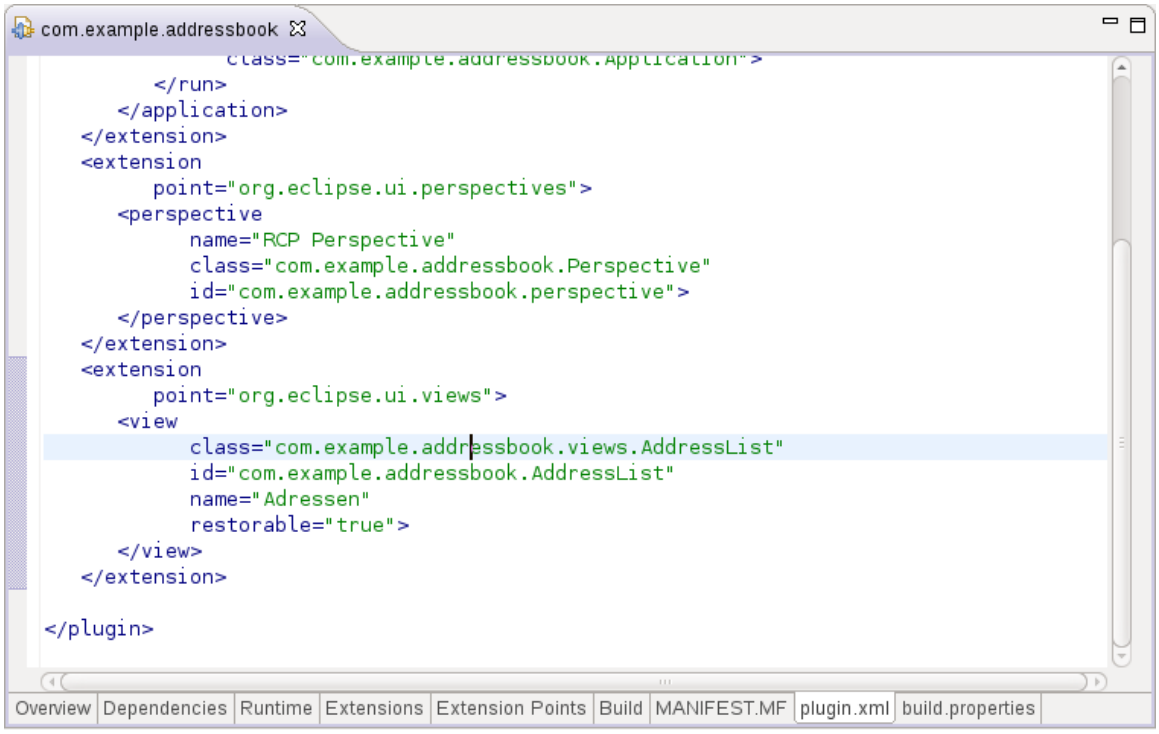
Eine RCP-Anwendung kann über sog. *Extensions* um neue Views und Perspektiven erweitert werden. Dabei handelt es sich um den Erweiterungsmechanismus der Eclipse-Plattform, mit dem Anwendungen mit Inhalten befüllt werden können. Die Eclipse-Plattform stellt dafür eine Vielzahl von Erweiterungspunkten (*Extension Points*) bereit.

So definiert die Eclipse-Workbench die Extension Points *org.eclipse.ui.views* und *org.eclipse.ui.perspectives*, mit dem neue Views und Perspektiven angelegt werden können. RCP-Entwickler verwenden solche Extension Points der Eclipse-Plattform, um ihre Anwendungen um neue Funktionalitäten zu erweitern. Dies erfolgt in dem Extension Editor des Plug-ins und wird in einer Datei *plugin.xml* im Projekthauptverzeichnis abgelegt:

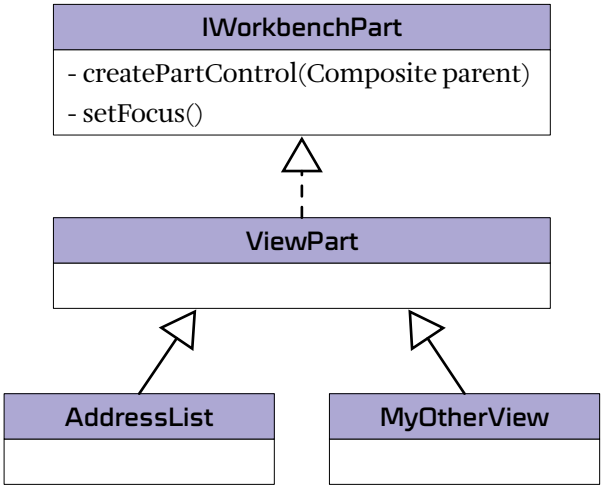
Ein View wird unter Angabe einer anwendungsweit eindeutigen ID, Reitertext sowie einer Implementierungsklasse hinzugefügt:



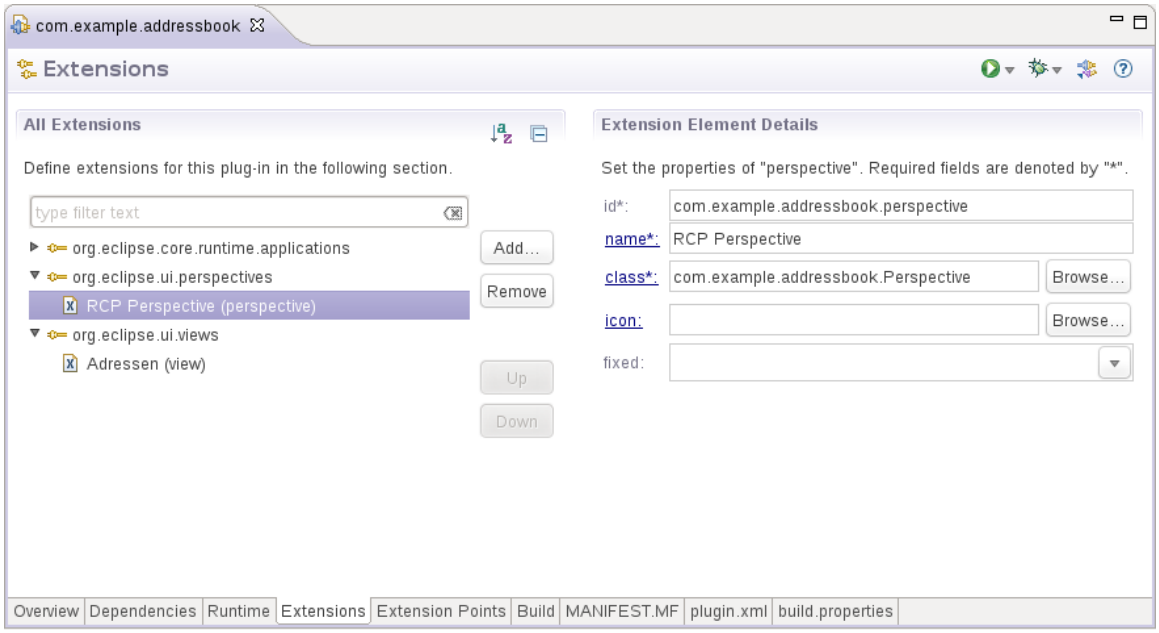
Abgelegt werden diese Informationen in der Datei *plugin.xml*:



Die Implementierungsklasse erbt von der Eclipse-Klasse *ViewPart* und muss v.a. die Methode *createPartControl* implementieren, um das View mit Inhalten zu befüllen:



Erweitert man die Anwendung so um ein weiteres View, ist dies zunächst nicht sichtbar, denn die Anordnung der Views wird durch die Perspektive festlegt. Perspektiven werden der Workbench ebenfalls über Extensions bekannt gegeben. Dazu wird der Extension Point *org.eclipse.ui.perspectives* verwendet und es ist ebenso eine eindeutige ID, Name und eine implementierende Klasse anzugeben:



Die Klasse muss *IPerspectiveFactory* implementieren und ist dafür zuständig, die initialen Inhalte und Anordnung der Reiter festzulegen. Dazu erhält sie eine *IPageLayout*-Instanz, mit der sie das Layout für die Perspektive konfigurieren kann. Ein View wird der Perspektive durch den Aufruf von *IPageLayout#addView* hinzugefügt. Folgendermaßen würde ein View mit 30% der Gesamtbreite links vom Editierbereich angeordnet werden:

```
public class MyPerspective implements IPerspectiveFactory {
    public void createInitialLayout(IPageLayout layout) {
        layout.addView("com.example.addressbook.views.AddressList",
            IPageLayout.LEFT, 0.3f, layout.getEditorArea());
    }
}
```

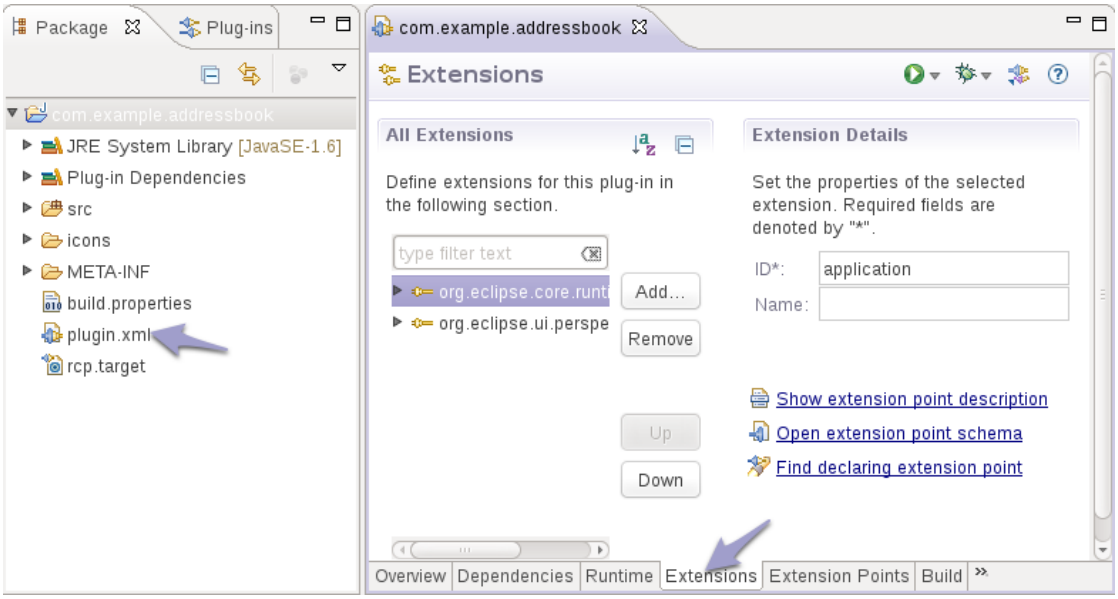
Weitere Informationen

- Eclipse Help: Plugging into the workbench
  http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/workbench.htm
- Eclipse Wiki: Eclipse User Interface Guidelines
  http://wiki.eclipse.org/index.php/User_Interface_Guidelines
- Presentation: UI Patterns in Eclipse
  http://www.slideshare.net/madhusamuel/patterns-in-eclipse
- Blog-Post: Replacing the Perspective-Switcher in RCP apps
  http://eclipsesource.com/blogs/2009/03/31/replacing-the-perspective-switcher-in-rcp-apps/

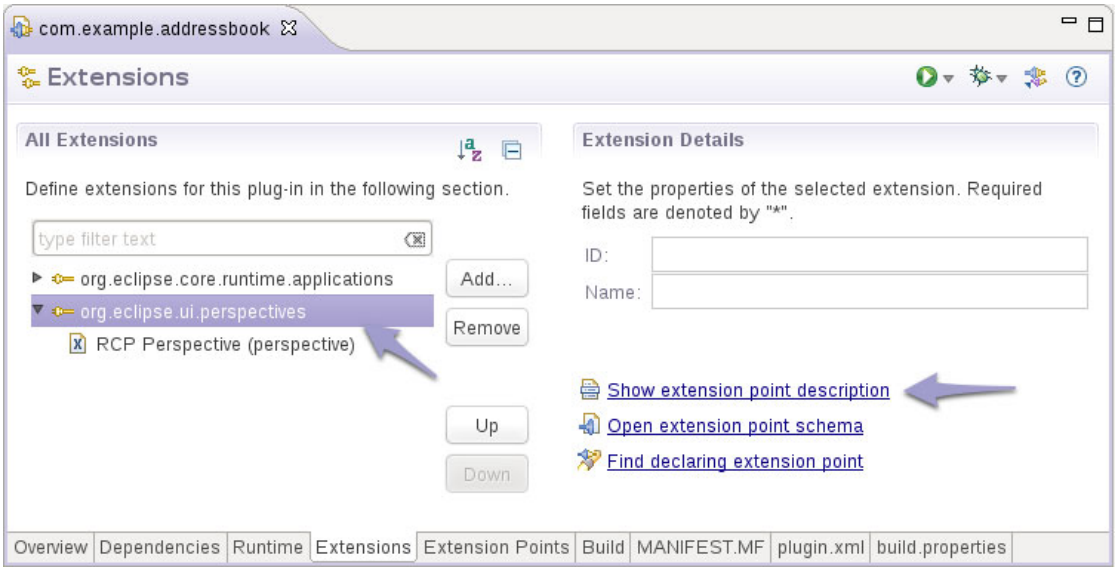
TUTORIAL 3.1

Anwendung um ein Adressen-View erweitern

☐ Öffnen Sie in dem Projekt *com.example.addressbook* die Datei “plugin.xml”. Wählen Sie den Tab “Extensions”. Sie sehen alle Erweiterungen, die die Anwendung aktuell einbringt:

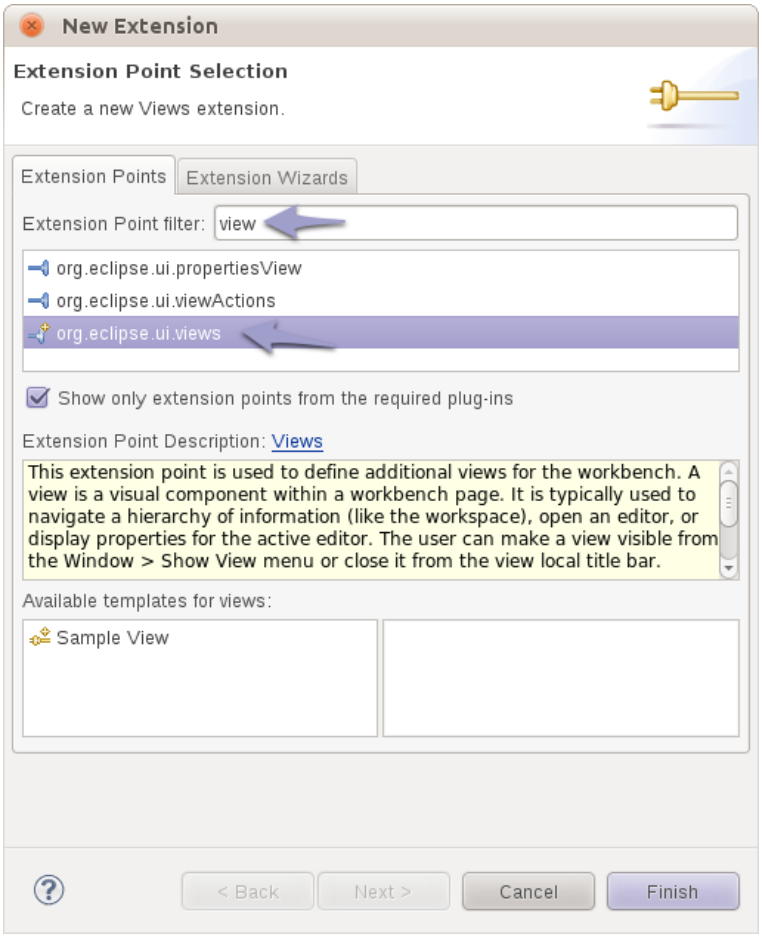


☐ Öffnen Sie mit *Show extension point description* einmal die Dokumentation zu dem Extension Point *org.eclipse.ui.perspectives*. So können Sie generell weitere Informationen zu der Funktionsweise eines Extension Points abrufen:



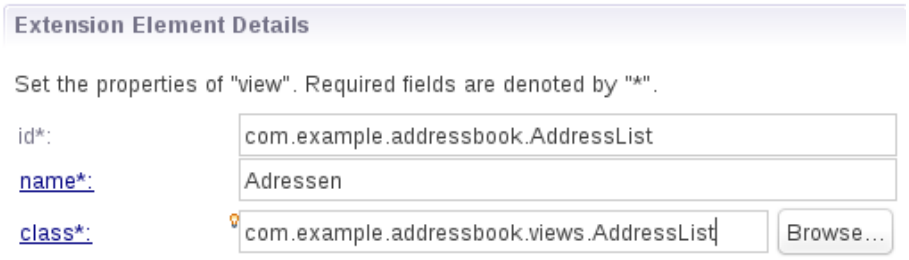
☐ Machen Sie sich mit der vorhandenen Extension zu *org.eclipse.ui.perspectives* vertraut.

- ☐ Fügen Sie eine neue Extension zu *org.eclipse.ui.views* hinzu, um der Workbench ein neues View bekannt zu machen. Klicken Sie dazu *Add...* und wählen den Extension Point *org.eclipse.ui.views*:

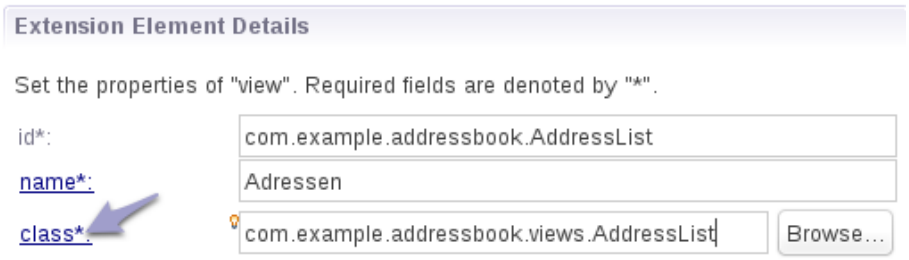


- ☐ Fügen Sie der Extension ein neues *view*-Element hinzu:

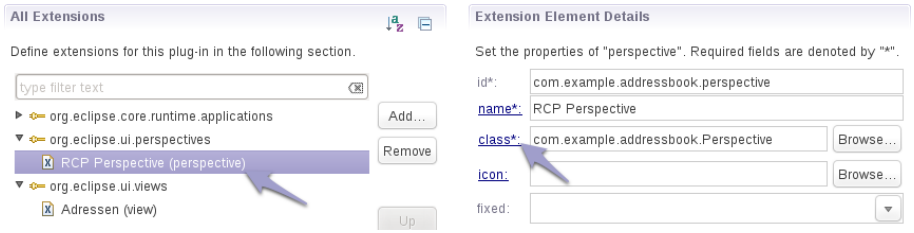
☐ Vergeben Sie als ID *com.example.addressbook.AddressList*, als Label *Adressen* sowie als Klasse *com.example.addressbook.views.AddressList*:



☐ Legen Sie für das View per Klick auf *class:* eine Java-Klasse an:



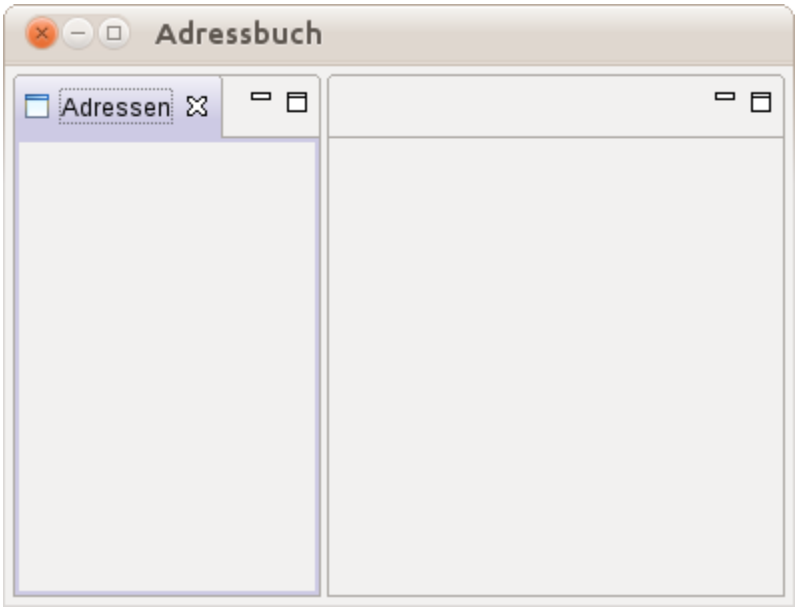
☐ Öffnen Sie die *PerspectiveFactory*-Klasse über die zugehörige Extension:



☐ Ergänzen Sie das soeben erstellte View über die Implementierung der *createInitialLayout*-Methode in der PerspectiveFactory-Klasse:

```
public void createInitialLayout(IPageLayout layout) {
    layout.addView("com.example.addressbook.AddressList",
        IPageLayout.LEFT, 0.4f, layout.getEditorArea());
}
```

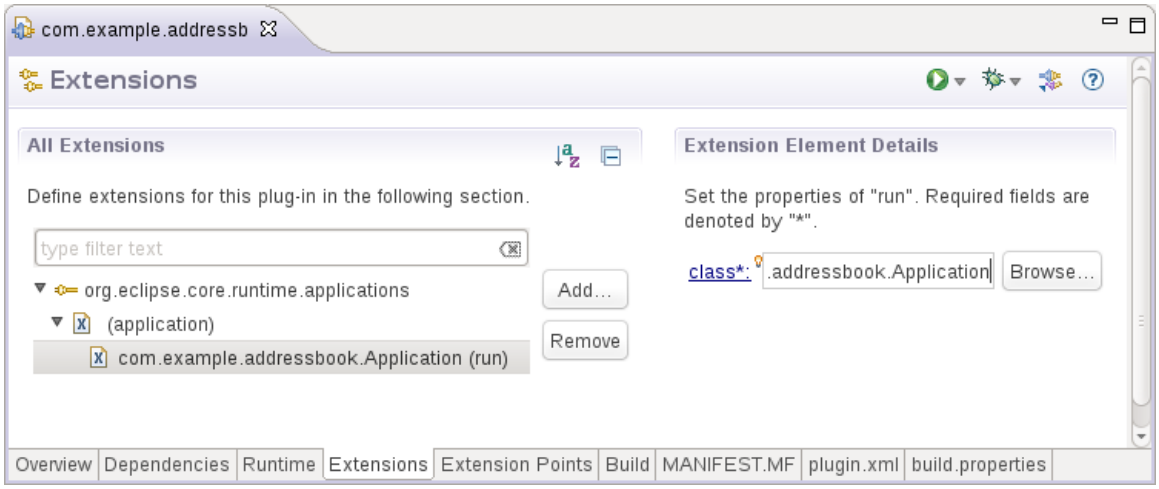
- ☐ Starten Sie die Anwendung und prüfen Sie, ob das View angezeigt wird:



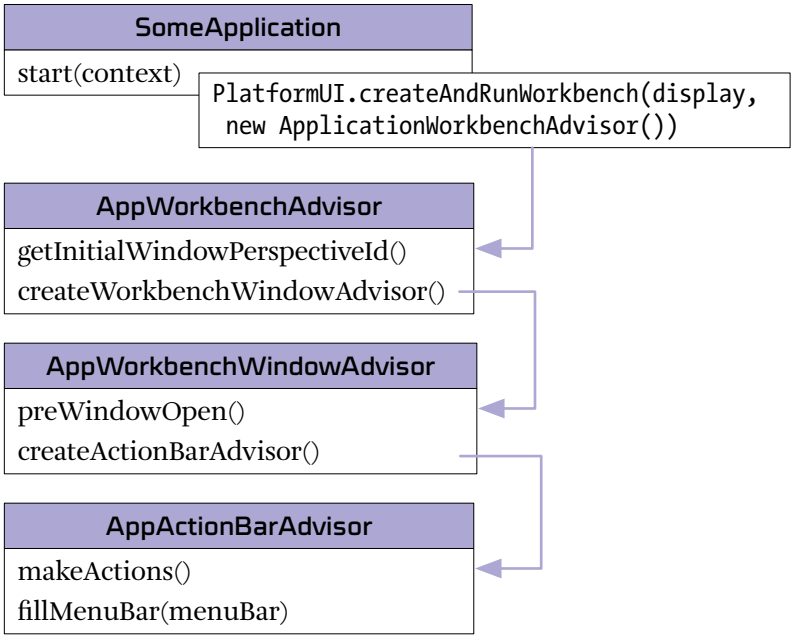
KAPITEL 4

Start und Konfiguration der Applikation

Eintrittspunkt in eine RCP-Applikation ist die Applikationsklasse, die über den Extension Point *org.eclipse.core.runtime.applications* registriert wird:



Diese Applikationsklasse implementiert das Interface *IApplication* ist für den Start und das Beenden der Applikation zuständig. Hier wird die Workbench initialisiert und gestartet. Dieser Vorgang wird von den drei *Advisor*-Klassen begleitet, mit denen die Workbench angepasst und konfiguriert werden kann:



Mittels des *WorkbenchAdvisors* werden Einstellungen konfiguriert, die für die gesamte Workbench gelten. Zum Beispiel die initial anzuzeigende Perspektive:

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    public String getInitialWindowPerspectiveId() {
        return "com.example.addressbook.Perspective";
    }

}
```

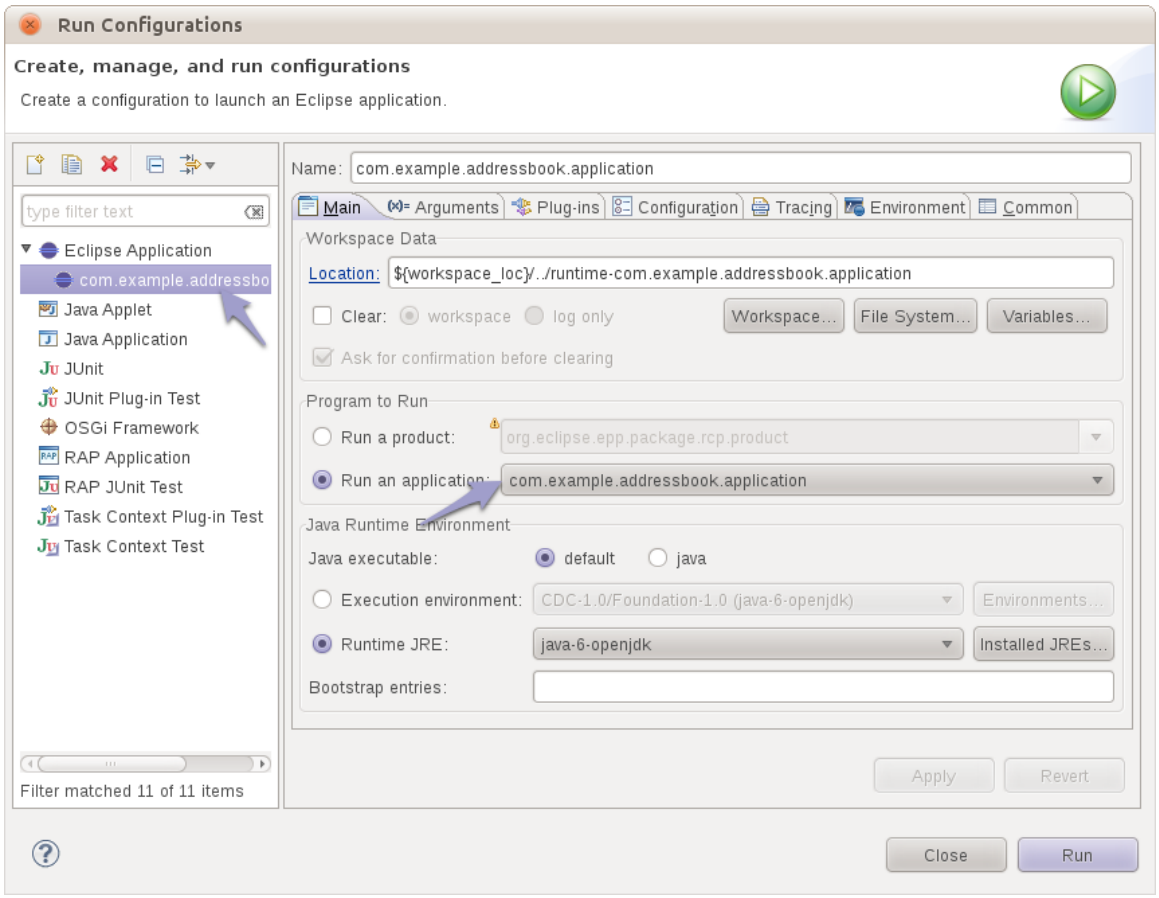
Der *WorkbenchWindowAdvisor* ist dafür zuständig, die einzelnen Workbench-Fenster zu konfigurieren. Hier kann z.B. Größe und Titel des WorkbenchWindow angegeben werden und *Coolbar, PerspectiveBar* und *StatusBar* ein- und ausgeblendet werden:

```
public class ApplicationWorkbenchWindowAdvisor extends WorkbenchWindowAdvisor {

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
        configurer.setInitialSize(new Point(800, 600));
        configurer.setShowCoolBar(false);
        configurer.setShowPerspectiveBar(true);
        configurer.setShowStatusLine(false);
        configurer.setTitle("Adressbuch");
    }

}
```

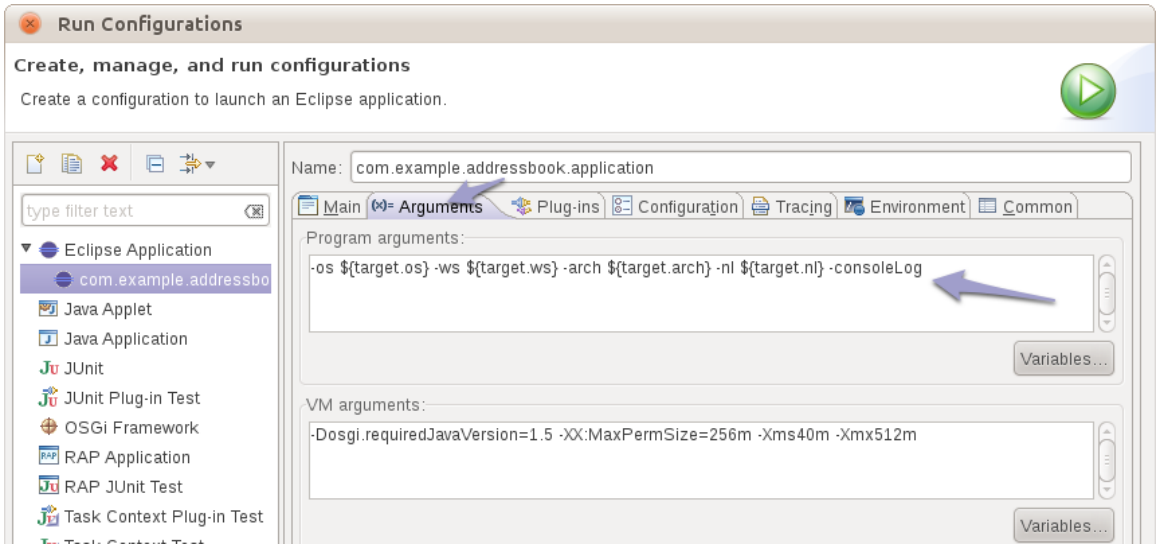
Der *ActionBarAdvisor* erlaubt die initiale Befüllung der Menüs und Toolbars der Anwendung, wurde mittlerweile jedoch vollständig durch die Verwendung von Commands abgelöst, siehe Commands (Seite 108).

Startkonfiguration

Zum Start der Applikation wird eine Startkonfiguration vom Typ *Eclipse Application* verwendet. In der Startkonfiguration wird die zu startende Applikation konfiguriert:



Unter *Arguments* › *Program arguments* können Laufzeitoptionen für die Eclipse-Anwendung angegeben werden:

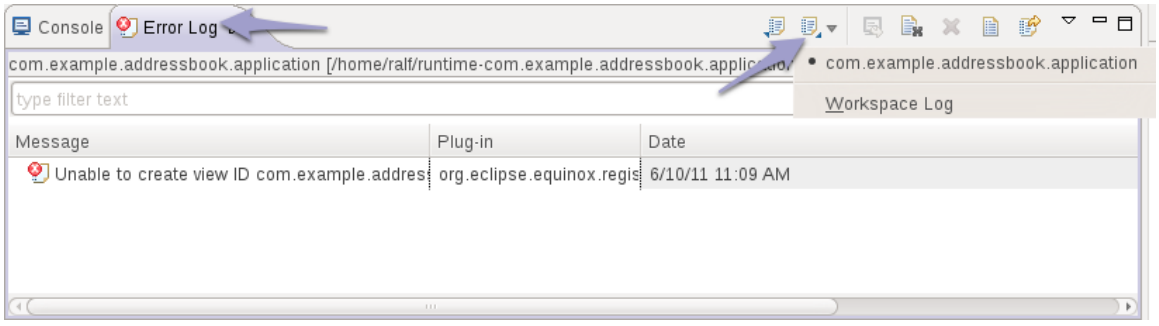


Standardmäßig wird immer die Plattform und Sprache der Testumgebung über Umgebungsvariablen definiert. Zudem wird die Anwendung mit *-consoleLog* so konfiguriert, dass alle Meldungen und Exceptions der Eclipse-Anwendung auch auf der System-Konsole ausgegeben werden:

```
-os ${target.os} -ws ${target.ws} -arch ${target.arch} -nl ${target.nl} -consoleLog
```

Eine vollständige Aufzählung aller Laufzeit-Optionen finden Sie in der Eclipse-Hilfe unter *Platform Plug-in Developer Guide* › *Reference* › *Other reference information* › *Runtime options*.

Neben der Aufgabe auf der Systemkonsole werden alle Meldungen der Eclipse-Komponenten auch in eine Log-Datei geschrieben. Diese können Sie in der IDE über *Window* › *Show View* › *Error Log* anzeigen. Standardmäßig werden die Meldungen der Eclipse IDE-Anwendung - Sie können jedoch zum *Error Log* Ihrer Anwendung wechseln:



Best Practice: Umgang mit IDs

IDs für Workbench-Elemente wie Views und Perspektiven sollten eindeutig und aussagekräftig gewählt werden. Für die Verwendung der IDs im Code empfiehlt es sich, String-Konstanten in einer separaten Klasse anzulegen, da dies übersichtlicher und leichter änderbar ist. Zum Beispiel:

```
public class AddressBook {

    // Views
    public static final String VIEW_ID_ADDRESS_LIST = "com.example.addressbook.AddressList";

    // Perspectives
    public static final String PERSPECTIVE_ID_ADDRESS = "com.example.addressbook.Addresses";

}
```

TUTORIAL 4.1

Anwendung konfigurieren

Anwendung über Advisor-Klassen konfigurieren:

- ☐ Die Extension zu *org.eclipse.core.runtime.applications* verweist auf die Applikationsklasse, mit der die Adressbuch-Anwendung definiert wird. Öffnen Sie diese Klasse und machen Sie sich mit dieser und allen Advisor-Klassen in dem Projekt vertraut. Finden Sie heraus in welcher Klasse die initiale Perspektive der Applikation festgelegt wird und welche Klasse den Titel und die initiale Größe der Applikation festlegt.

- ☐ Konfigurieren Sie die Fenstergröße in der entsprechenden Advisor-Klasse auf 800 x 600.

- ☐ Blenden Sie die Perspektiv-Bar ein, indem Sie in der *ApplicationWorkbenchWindowAdvisor*-Klasse unter *preWindowOpen* ergänzen:

  ```
  configurer.setShowPerspectiveBar(true);
  ```

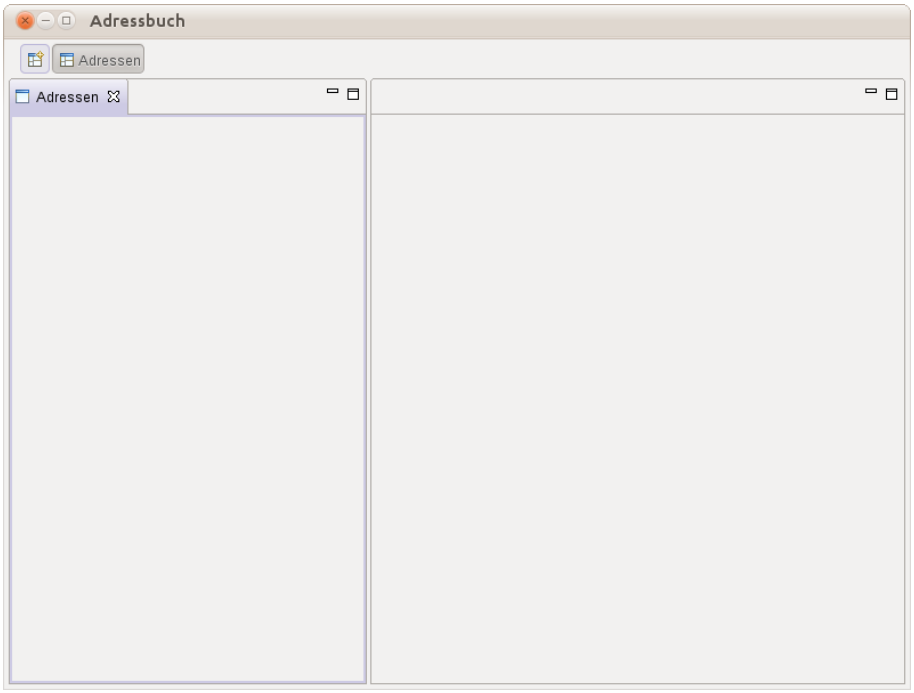
Perspektive umbenennen:

- ☐ Ändern Sie in der Extension zu *org.eclipse.ui.perspectives* das Label der existierende Perspektive von “RCP Perspective” zu “Adressen”.

- ☐ Ändern Sie hier ebenfalls die ID der Perspektive zu *com.example.addressbook.Addresses*. Passen Sie die Angabe der ID der initialen Perspektive in der Klasse *ApplicationWorkbenchAdvisor* entsprechend an.

- ☐ Benennen Sie die existierende Perspektivenklasse mittels *Refactor › Rename* in *AddressesPerspective* um und verschieben Sie sie mit *Refactor › Move* in das Package *com.example.addressbook.perspectives*. Dieses können Sie direkt im Refactoring-Dialog mit *Create Package...* anlegen.

☐ Starten Sie die Anwendung testweise:



Konstantenklasse anlegen:

☐ Legen Sie eine neue Klasse *com.example.addressbook.ids.AddressBook* zur Ablage von Konstanten an. Verschieben Sie die vorhandene ID-Konstante für die Perspektive aus dem *ApplicationWorkbenchAdvisor* mit *Refactor › Move...* in diese Konstantenklasse.

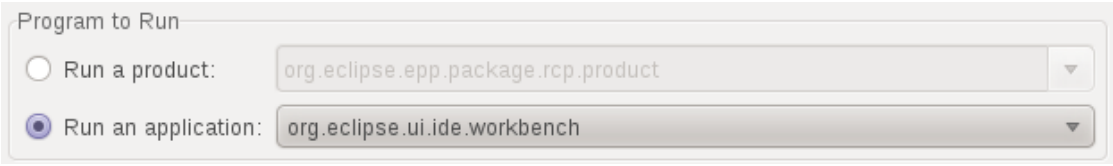
☐ Öffnen Sie die *AddressesPerspective*-Klasse und erstellen Sie für den ID-String *com.example.addressbook.AddressList* mit *Refactor › Extract Constant* eine Konstante *VIEW_ID_ADDRESS_LIST*. Verschieben Sie diese Konstante ebenfalls mit *Refactor › Move...* in die Konstantenklasse.

Startkonfiguration:

☐ Wählen Sie unter *Run › Run Configurations* die Startkonfiguration *com.example.addressbook.application* (diese wurde automatisch erstellt, als Sie *Launch an Eclipse application* verwendet haben).

☐ Machen Sie sich mit der Startkonfiguration vertraut.

- ☐ Wählen Sie einmal testweise als zu startende Anwendung *org.eclipse.ui.ide.workbench* aus, um einen Fehler beim Start der Anwendung zu provozieren:



- ☐ Starten Sie die Anwendung und prüfen Sie die Ausgabe auf der Systemkonsole. Sie sollten eine Meldung sehen:

```
Application "org.eclipse.ui.ide.workbench" could not be found in the registry.
The applications available are: com.example.addressbook.application
```

- ☐ Wählen Sie wieder die Anwendung *com.example.addressbook.application* aus und testen Sie, dass die Anwendung wieder korrekt startet.

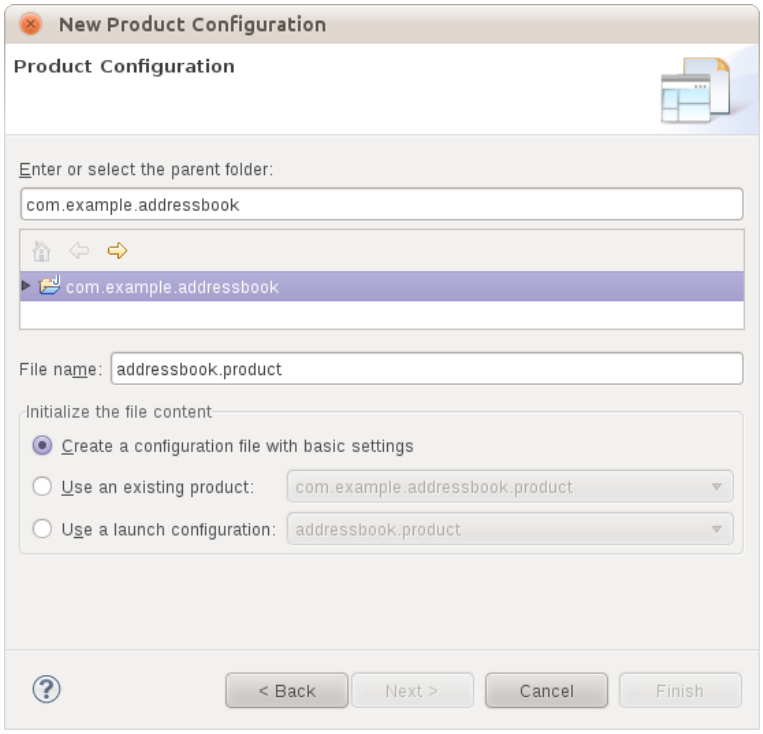
KAPITEL 5

Produktexport

Der Export einer ohne IDE lauffähigen Version einer RCP-Anwendung erfolgt mittels eines *Products*. Ein Produkt definiert die Anwendungsbestandteile und wie diese zusammenzupacken und zu exportieren sind. Zudem haben Sie über das Produkt die Möglichkeit, Ihre Anwendung zu "branden", d.h. mit optischen Details wie Titel, Splash-Screen und Icons zu versehen.

Ein Produkt besteht aus einer Extension zu *org.eclipse.core.runtime.products* und einer *.product*-Datei. Der Extension Point ist nur zur Laufzeit relevant und enthält die für die Anwendung selbst relevanten Informationen wie die Einstellungen zum Branding der Anwendung. Die *.product*-Datei hingegen ist nur zur Buildzeit relevant. Sie ist Ausgangspunkt für den *PDE Build*, den Eclipse-eigenen Buildprozess, der die Bestandteile der Anwendung compiliert, zu JAR-Dateien verpackt und eine lauffähige Gesamtanwendung zusammenstellt.

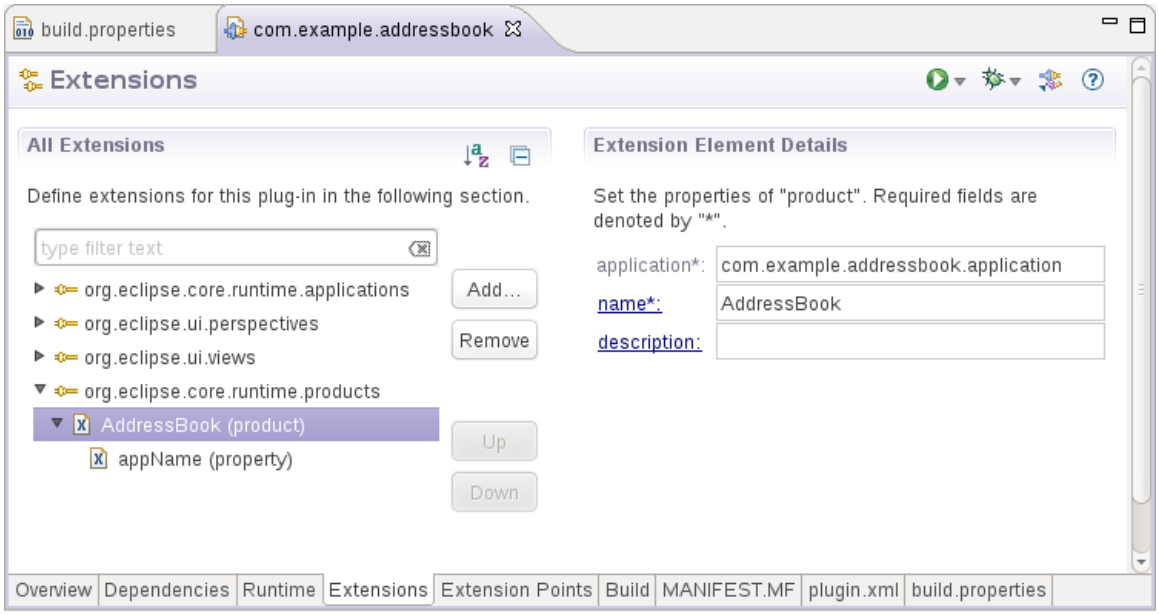
Ein neues Produkt wird erstellt mit *File* › *New* › *Other* › *Plug-in Development* › *Product Configuration*:



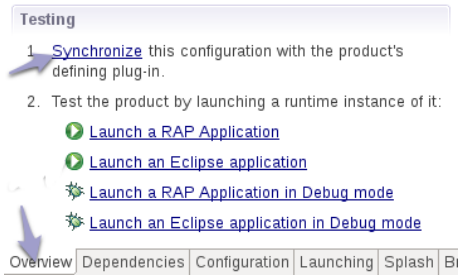
Damit erzeugen Sie die *.product*-Datei zur Konfiguration des Produkts. Am wichtigsten ist dabei die Festlegung unter *Dependencies*, aus welchen Plug-ins das Produkt besteht. Sie können Ihre Anwendungs-Plug-ins hinzufügen und mit *Add Required* alle abhängigen Plug-ins ergänzen lassen. Unter *Splash* und *Branding* können Sie das Aussehen der Anwendung konfigurieren.

Die Extension zu *org.eclipse.core.runtime.products* muss im Produkt einmalig unter *Overview* angelegt werden:

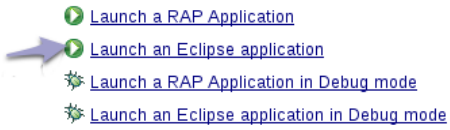
Dadurch wird für das Produkt eine eindeutige Produkt-ID erstellt, mit der das Produkt zur Laufzeit identifiziert wird. Zudem wird die Extension zu *org.eclipse.core.runtime.products* angelegt und alle zur Laufzeit relevanten Informationen, vor allem zum Branding der Anwendung, übertragen:



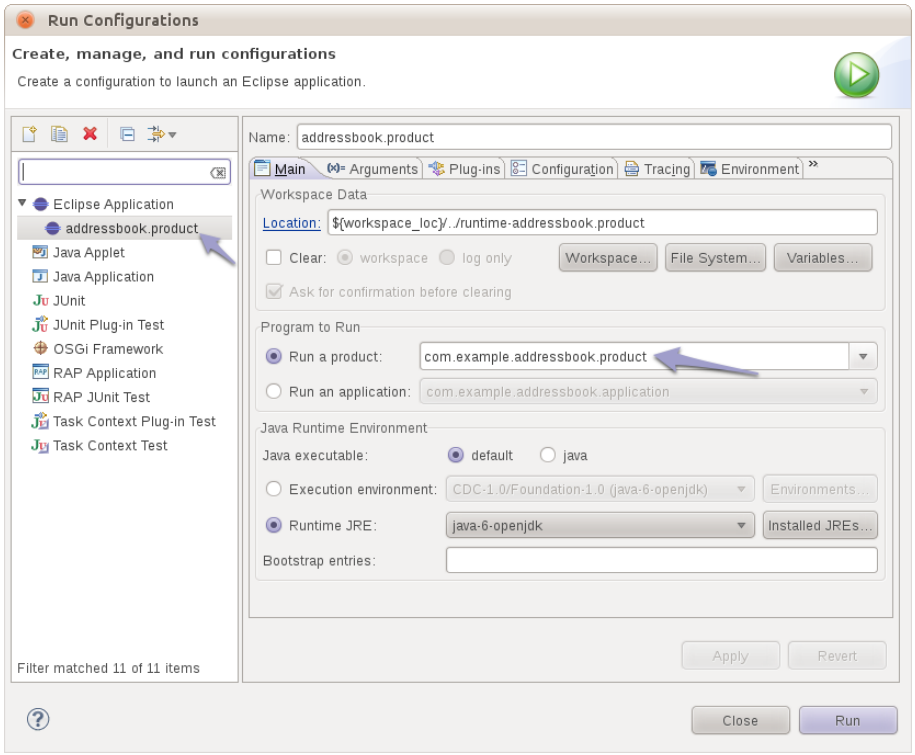
Ändern Sie zu einem späteren Zeitpunkt z.B. das Branding der Anwendung, können Sie die Extension erneut einem Klick auf *Synchronize* aktualisieren:



Ein Produkt kann auch zum Start in der IDE verwendet werden:



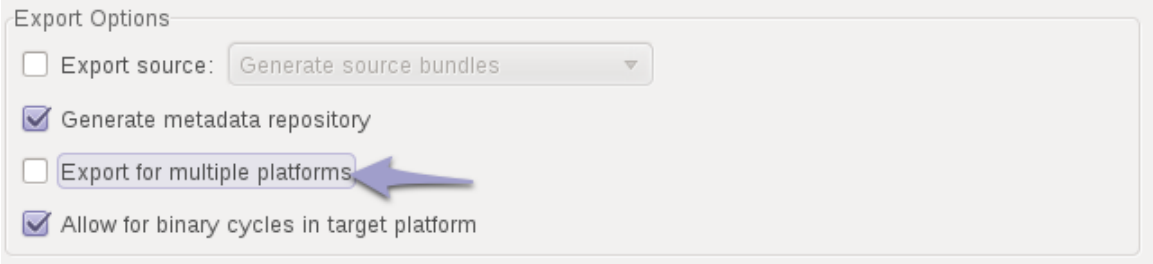
Dies erstellt bzw. aktualisiert eine produktbasierte Startkonfiguration:



Bitte beachten Sie unbedingt, dass diese Startkonfiguration unverändert bleibt, auch wenn die Angaben in der Produktdatei geändert werden. Nur durch einen erneuten Start über *Launch an Eclipse application* im Produkt wird auch die Startkonfiguration aktualisiert!

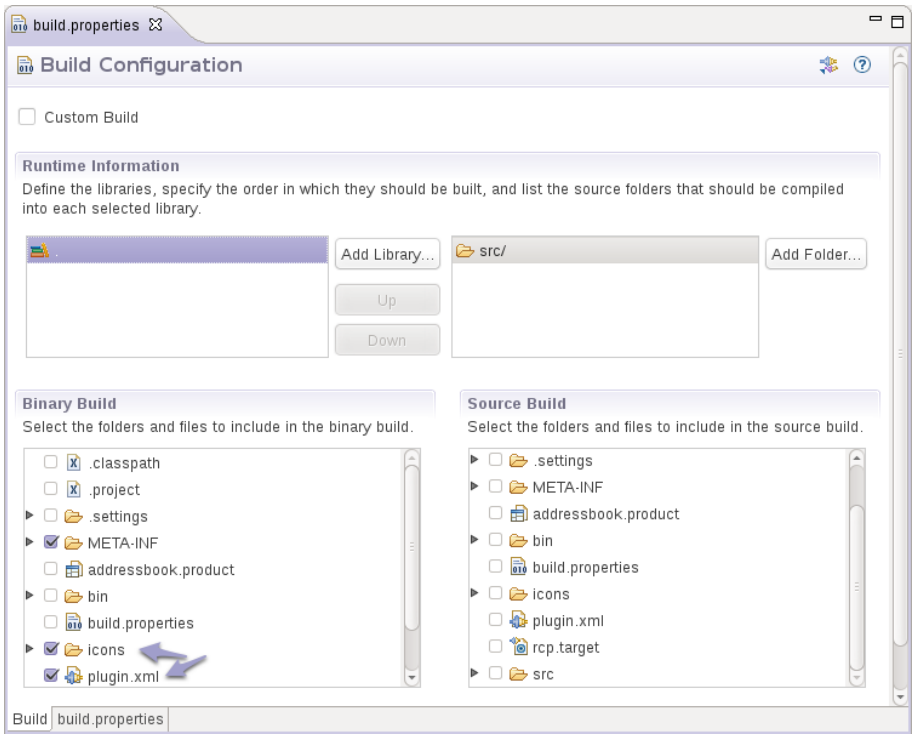
Produkte können mit *File > Export > Eclipse Product* exportiert werden. Dabei werden alle Anwendungsbestandteile zusammengepackt und mit einem *Binary Launcher* zum Start der Anwendung unter dem jeweiligem Betriebssystem versehen.

Der Produkt-Export kann auch für andere Betriebssystem erfolgen. Dazu muss das *Eclipse Delta Pack* in der Target-Plattform vorhanden sein (dies ist kein standardmäßiger Bestandteil von Eclipse RCP). Dann bietet der Produkt-Export die Option *Export for multiple platforms* an:



Build-Konfiguration build.properties

Alle Projekte werden beim Export durch den Eclipse-Buildprozess gebaut und zu einer selbstständig lauffähigen Anwendung verpackt. Dabei gilt es zu beachten, dass nur Klassen und Ressourcen-Dateien aus den Quellpfaden wie *src/* exportiert werden. Zusätzliche Ressourcen-Dateien, die zur Laufzeit zur Verfügung stehen sollen, müssen in der *build.properties*-Datei des Plug-ins konfiguriert werden:



Dies betrifft vor allem die *plugin.xml* Datei sowie den *icons*-Ordner: Wurden diese erst nach der Erstellung des Plug-ins hinzugefügt, müssen sie unter Umständen manuell zur *build.properties*

hinzugefügt werden. Häufig fällt das Fehlen dieser Konfiguration erst beim ersten Export auf, da beim Start von Plug-ins aus der IDE heraus alle Dateien im Plug-in-Ordner zur Verfügung stehen.

TUTORIAL 5.1

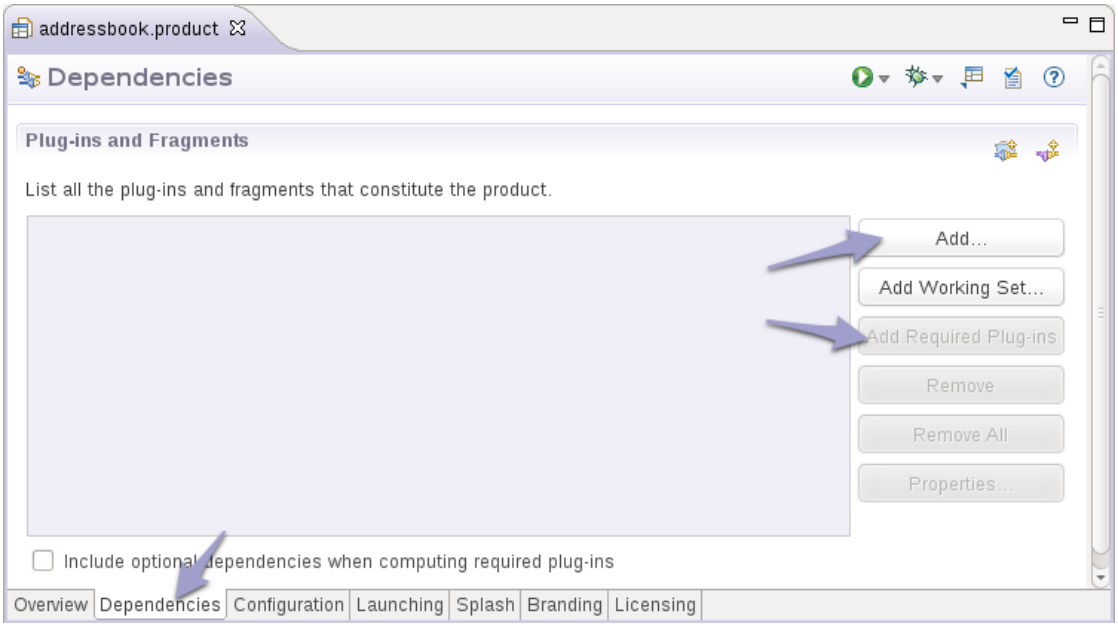
Produkt erstellen und exportieren

- ☐ Erstellen Sie mit *File › New › Other › Product Configuration* ein Produkt *addressbook.product* im *com.example.addressbook*-Projekt. Wählen Sie dabei *Create a configuration file with basic settings.*

- ☐ Legen Sie unter *Overview › Product Definition › New* eine neue Produkt-ID für das Produkt an.

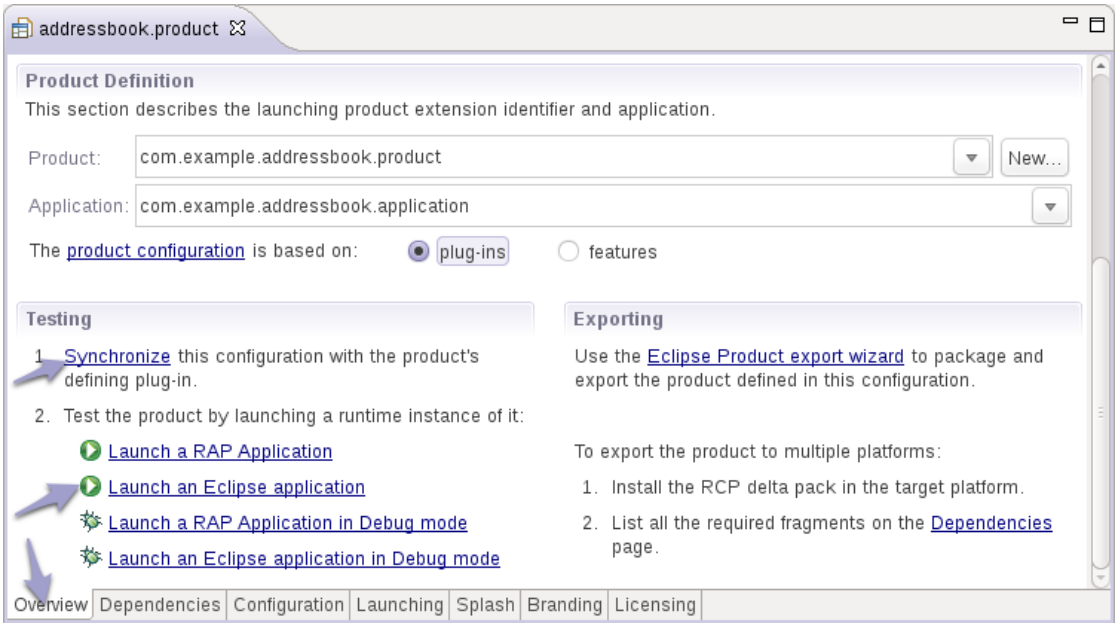
- ☐ Geben Sie *AddressBook* als *Product Name* an.

- ☐ Fügen Sie in der Produktkonfiguration unter Dependencies das Anwendungs-Plug-in *com.example.addressbook* hinzu und ergänzen Sie mit *Add Required Plug-ins* alle Abhängigkeiten:



- ☐ Geben Sie unter *Launching* als *Launcher Name* "*addressbook*" an.

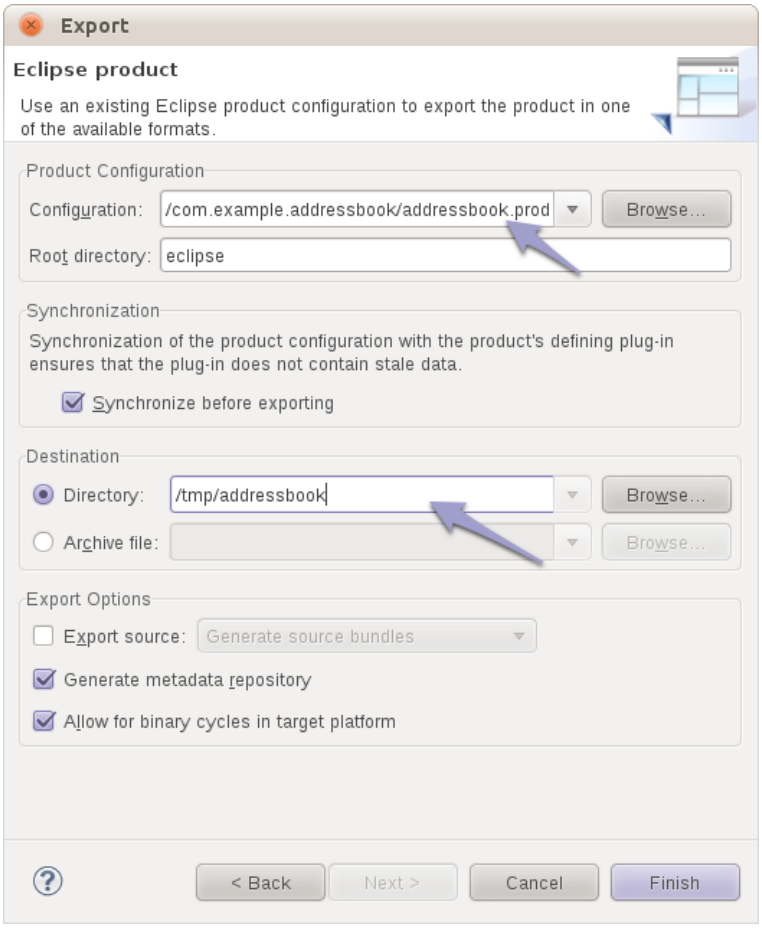
☐ Starten Sie das Produkt unter *Overview* mit *Launch an Eclipse Application*:



☐ Es wurde eine neue Startkonfiguration *addressbook.product* erstellt. Löschen Sie die alte Startkonfiguration *com.example.addressbook.application* unter *Run › Run Configurations*, damit dies nicht zu Verwirrungen führt:

Achtung: Die erstellte Startkonfiguration für das Produkt wird nur aktualisiert, wenn über das Produkt mit *Launch an Eclipse Application* gestartet wird. Wenn Sie die Produkt-Datei verändern und die Startkonfiguration direkt starten, verwenden Sie eine veraltete Startkonfiguration.

☐ Exportieren Sie das Produkt mittels *File* › *Export* › *Eclipse product*. Wählen Sie hier die Produkt-Datei und ein Zielverzeichnis:



☐ Starten Sie die exportierte Anwendung.

Fehlersuche und häufige Probleme beim Produktexport:

Product ... could not be found: Wahrscheinlich konnte das Plug-in, welches das Produkt enthält, aufgrund fehlender Abhängigkeiten nicht gestartet werden. Prüfen/Validieren Sie die Plug-ins der aus dem Produkt erstellten Startkonfiguration.

Es treten nicht erklärbare Fehler beim Produktexport auf: Verwenden Sie als Zielordner für den Export einen neuen, leeren Ordner an.

Die exportierte Anwendung startet nicht: Öffnen Sie die Log-Dateien im Anwendungsordner oder starten Sie die Anwendung auf der Kommandozeile mit dem Parameter *-consolelog*. Überprüfen Sie die Produktkonfiguration und insbesondere die *build.properties*-Dateien.

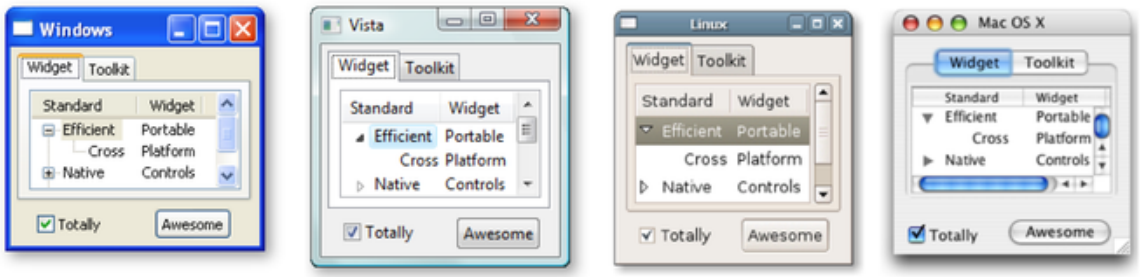
Die Anwendung wurde ohne *.exe* exportiert: Prüfen Sie die Option *The product includes native launcher artifacts* im Produkt unter *Overview* › *General Information*.

***Export for multiple platforms* funktioniert nicht**: Die Target Plattform für Eclipse RCP 3.7, die Sie in Kapitel 2 importiert haben, ist aktuell aufgrund von Bug 352361 nicht für den Export für fremde Zielplattformen geeignet.

KAPITEL 6

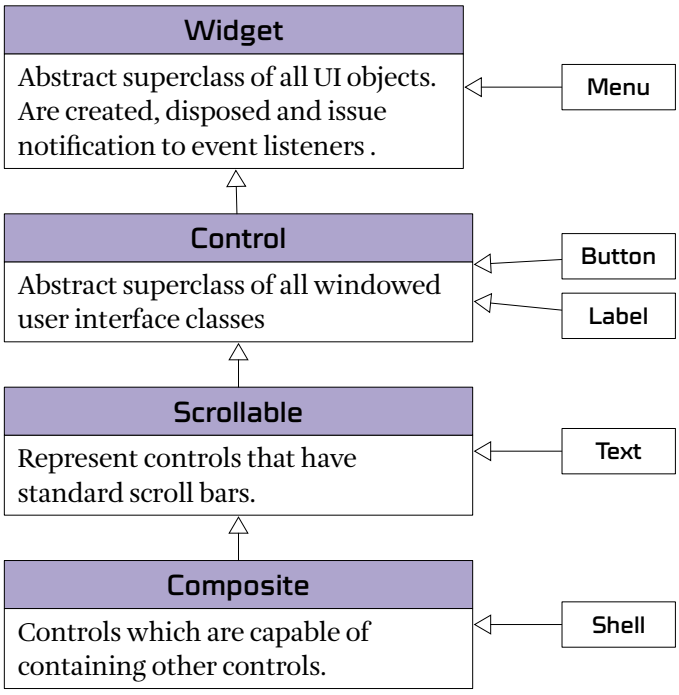
SWT Standard Widget Toolkit

Das Standard Widget Toolkit (SWT) ist das GUI-Framework der Eclipse-Plattform. Es stellt ein Java API für die nativen GUI-Bibliotheken von Windows, Linux (GTK) und Mac OS X (Cocoa) zur Verfügung. Durch die Nutzung der Steuerelemente des Betriebssystems wird auf allen unterstützten Plattformen ein natives Aussehen und Verhalten erzielt:



Widgets

Alle Standard-SWT-Widgets befinden sich im Package *org.eclipse.swt.widgets* und sind von den Basisklassen *Widget*, *Control*, *Scrollable* und *Composite* abgeleitet:



Eine Übersicht über alle Widgets finden Sie in Anhang 1: Übersicht SWT Widgets (Seite 196).

Widgets werden über den Konstruktor der Klasse erzeugt. Dabei sind das Eltern-Composite sowie ein Style-Flag anzugeben:

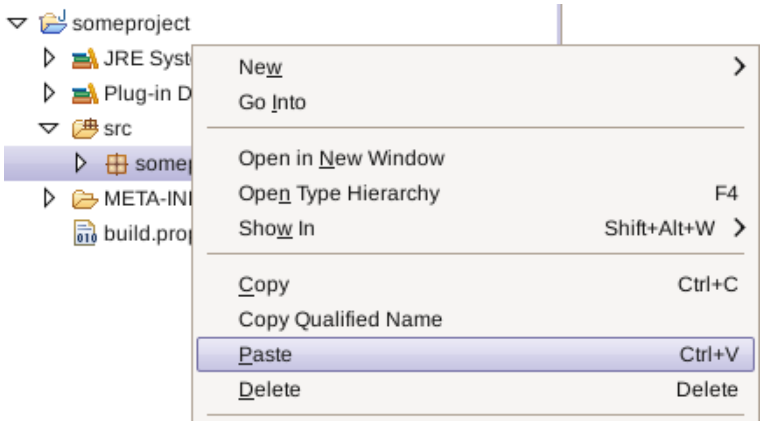
```
Button myButton = new Button(parentComposite, SWT.PUSH | SWT.RIGHT);
myButton.setText("Hello world!");
```

Von den Widgets der obersten Ebene wie *Shell* (Fenster) abgesehen haben alle Widgets ein Eltern-Element. Dieses wird das neu erzeugte Widget beherbergen.

Über die Style-Flags kann das Aussehen und Verhalten des Widgets beeinflusst werden. Es handelt sich dabei um Bitkonstanten, die in der Klasse *org.eclipse.swt.SWT* definiert sind. Die erlaubten Styles sind im JavaDoc der Widget-Klasse dokumentiert. Mehrere Style-Flags werden mit binärem Oder verknüpft. Möchte man keine Style-Flags angeben, verwendet man *SWT.NONE*.

SWT Snippets

Für alle Widgets finden Sie unter http://www.eclipse.org/swt/widgets/ kurze Beispiel-Snippets, die die korrekte Verwendung eines Widgets zeigen. Diese können über die Zwischenablage direkt in ein Eclipse-Projekt eingefügt werden:

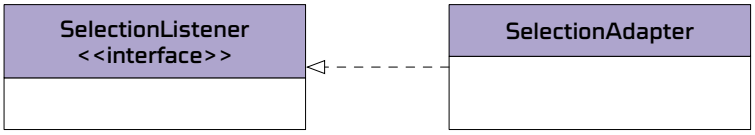


Ereignisbehandlung

Sobald ein Ereignis auftritt, werden alle beim jeweiligen SWT Control registrierten Listener darüber benachrichtigt. Je nach Control stehen verschiedene Listener-Methoden wie *addSelectionListener* oder *addFocusListener* zur Verfügung:

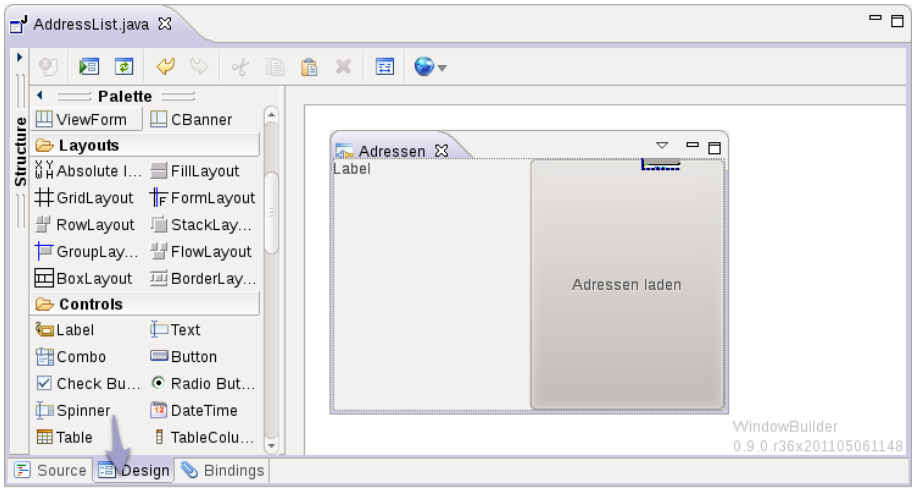
```
button.addSelectionListener(new SelectionListener() {

    public void widgetSelected(SelectionEvent e) {
        // Handle Button selection event here
    }

    public void widgetDefaultSelected(SelectionEvent e) {
        // Handle Button default selection event here
    }

});
```

Da für Interfaces immer sämtliche Methoden implementiert werden müssen, existiert zu den meisten Interfaces wie *SelectionListener* eine abstrakte Klasse *SelectionAdapter*, bei der nur die tatsächlich benötigten Methoden implementiert werden müssen:



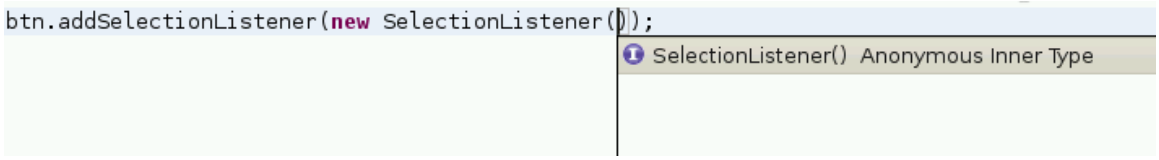
WindowBuilder

Google hat mit *WindowBuilder* einen freien GUI-Builder für SWT, JFace und Eclipse RCP veröffentlicht. Nach der Installation steht ein neuer *WindowBuilder Editor* zur Verfügung, mit dem UI-Komponenten wie Views oder Composites grafisch bearbeitet werden können. Hierbei erfolgt die Bearbeitung bidirektional, d.h. für erstellte Oberflächen wird Code generiert, der auch manuell verändert werden kann und mit der grafischen Darstellung abgeglichen wird:



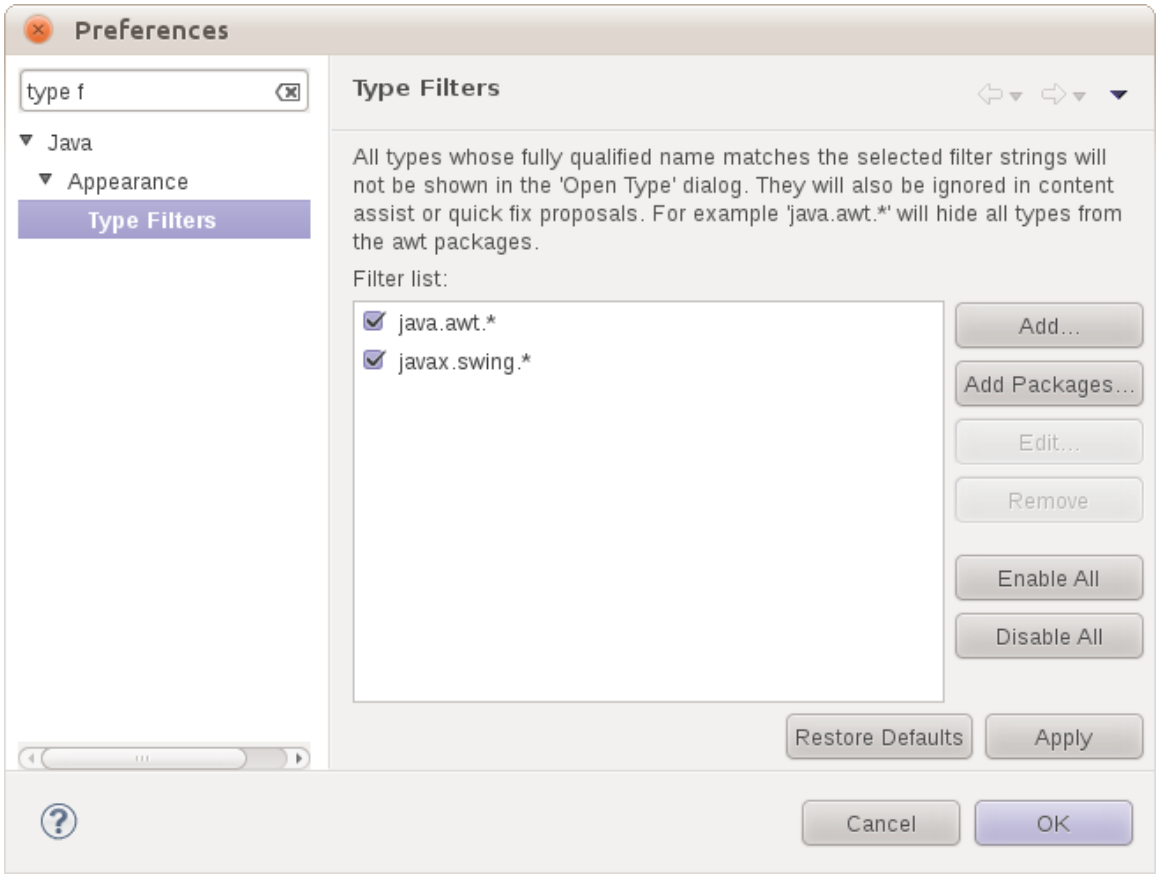
IDE Tipps & Tricks

Anonyme, innere Klassen können per *Ctrl-Space* Vervollständigung erzeugt werden:



Ggf. stören bei der Vervollständigung von Namen im Editor gleichnamige Klassen aus anderen Packages wie z.B. *java.awt*:

Diese Packages können in den Eclipse-Einstellungen unter *Java* › *Appearance* › *Type Filter* ausgeblendet werden:



Weitere Informationen zu SWT

- SWT: The Standard Widget Toolkit
  http://www.eclipse.org/swt/
- SWT Widgets / Snippets
  http://www.eclipse.org/swt/widgets/
- SWT Nebula Project: Supplemental Custom Widgets for SWT
  http://www.eclipse.org/nebula/
- Drag and Drop: Adding Drag and Drop to an SWT Application
  http://www.eclipse.org/articles/Article-SWT-DND/DND-in-SWT.html

TUTORIAL 6.1

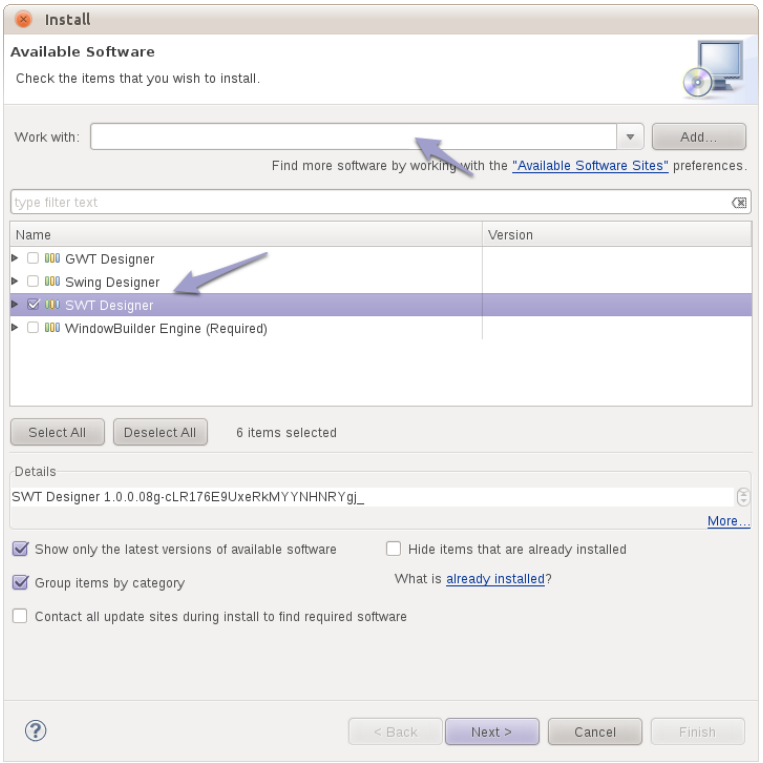
Widgets in Adresslisten-View einfügen

SWT Dokumentation verwenden:

- ☐ Führen Sie einige SWT-Snippets von http://www.eclipse.org/swt/widgets/ aus.
- ☐ Untersuchen Sie die SWT-Widget-Hierarchie, z.B. indem Sie die Klasse *Button* mit *Ctrl+Shift+T* öffnen und die Klassenhierarchie mit *F4* anzeigen.
- ☐ Ermitteln Sie mit den *JavaDocs* (*F2*) oder den SWT-Quelltexten (*F3*), welche Style-Flags für die Widget-Klassen *Button* und *Text* erlaubt sind.

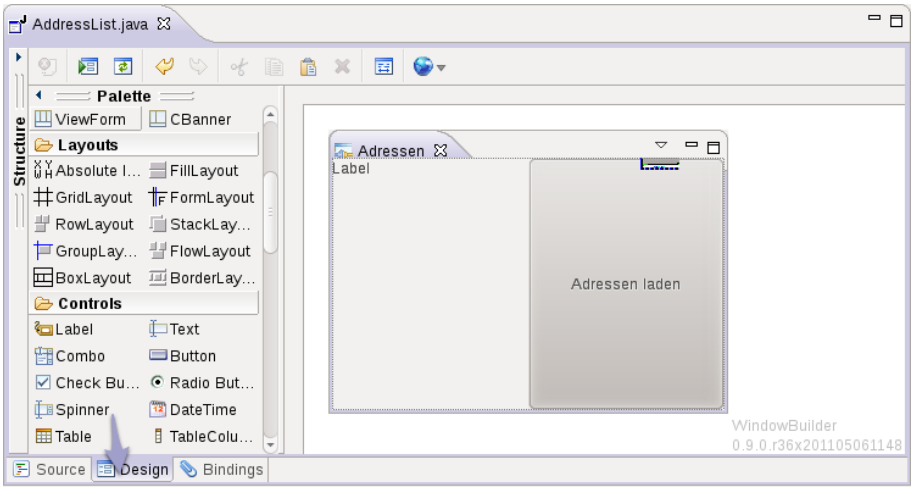
Installation Window Builder:

- ☐ Öffnen Sie *Help > Install New Software* und deaktivieren Sie die Option *Contact all update sites during install to find required software*.
- ☐ Ermitteln Sie unter *http://www.eclipse.org/windowbuilder/download.php* den Link zu der aktuellen Update Site für Eclipse 3.7 Indigo.
- ☐ Installieren Sie *SWT Designer* von dieser Update-Site:



Widgets zu View hinzufügen:

- ☐ Öffnen Sie die View-Klasse *AddressList* und wechseln Sie in die WindowBuilder Design-Sicht. Fügen Sie dem View ein *Label* und ein *Button* mit dem Text *Adressen laden* hinzu:



  (Hinweis: Sollte dieser Reiter nicht sichtbar sein, hilft ggf. Rechtsklick auf die View-Klasse › *Open With* › *WindowBuilder Editor*).

Ereignisbehandlung:

- ☐ Fügen Sie dem *Button* mit *Add event handler* › *selection* › *widgetSelected* einen Event-Listener hinzu.

- ☐ Implementieren Sie den Listener so, dass dem *Label* der Text *Adressen geladen* gesetzt wird, wenn der Button geklickt wurde. Um in dem Listener auf das Label zuzugreifen, müssen Sie die entsprechende Variable als *final* deklarieren oder zu einer Instanzvariable konvertieren (*Refactor* › *Convert Local Variable to Field*).

- ☐ Implementieren Sie die Methode *setFocus* der View-Klasse so, dass der Eingabefokus initial auf den Button gesetzt wird.

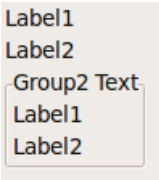
KAPITEL 7

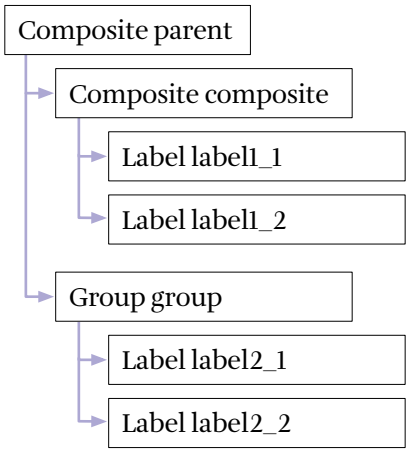
SWT Layout-Manager

Widget-Hierarchie, Composite und Group

Ein *Composite* ist ein Container für Steuerelemente. Möchten Sie zusätzlich einen sichtbaren Rahmen und eine Beschriftung um die enthaltenen Elemente, können Sie die Unterklasse *Group* verwenden:

```
Composite composite = new Composite(parent, SWT.NONE);
Label label1_1 = new Label(composite, SWT.NONE);
label1_1.setText("Label1");
Label label1_2 = new Label(composite, SWT.NONE);
label1_2.setText("Label2");

Group group = new Group(parent, SWT.NONE);
group.setText("Group2 Text");
Label label2_1 = new Label(group, SWT.NONE);
label2_1.setText("Label1");
Label label2_2 = new Label(group, SWT.NONE);
label2_2.setText("Label2");
```

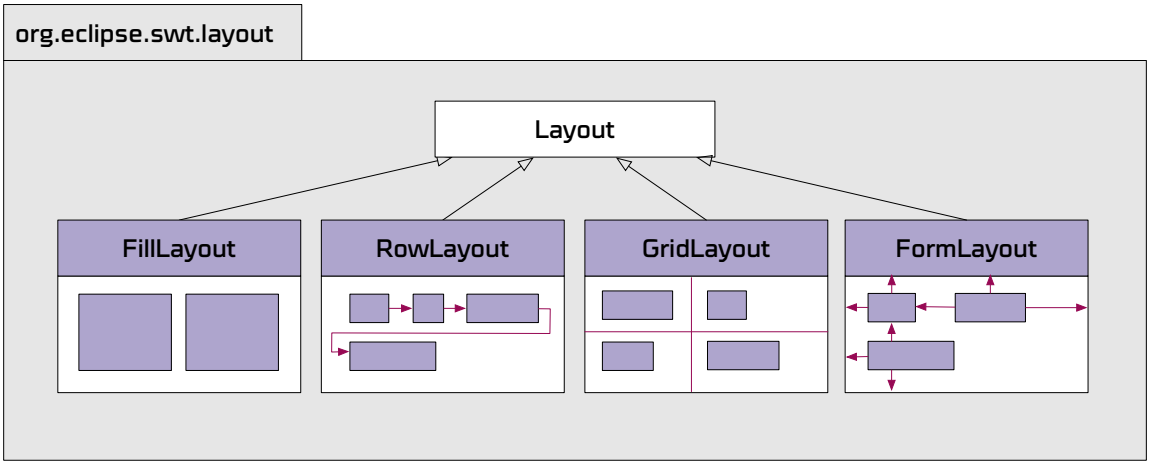




Layout-Manager

SWT kümmert sich zunächst nicht um die Position und Größe von Widgets. Da die Standardgröße für Controls (0, 0) ist, werden Controls erst dann sichtbar, wenn sie mit *setBounds*

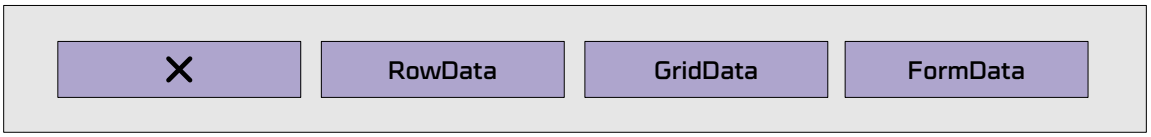
positioniert werden. Diese Aufgabe sollte man einem Layout-Manager überlassen. SWT stellt verschiedene Layout-Manager bereit:



Das Layout wird einem Composite gesetzt und positioniert dann die darin enthaltenen Controls:

```
composite.setLayout(new FillLayout());
```

Zu fast allen Layouts gehört eine *LayoutData*-Klasse:



Ein solches *LayoutData*-Objekt kann Controls gesetzt werden, um festzulegen, wie dieses einzelne Control auszurichten ist:

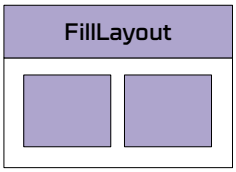
```
control.setLayoutData(layoutData);
```

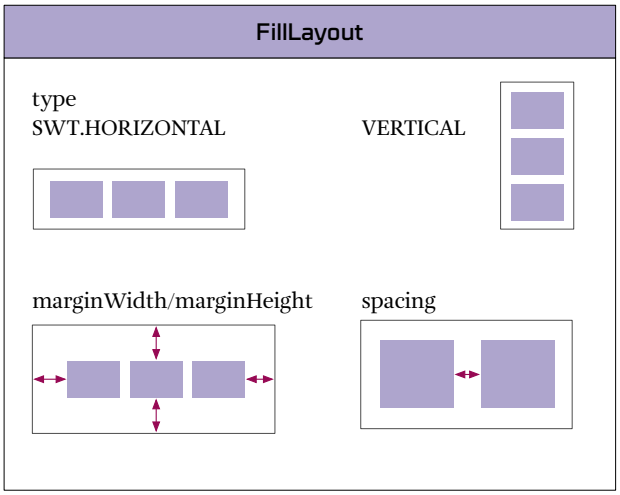
Beispiel: Einem Label in einem Composite mit *RowLayout* kann ein *RowData*-Objekt gesetzt werden, z.B. um eine abweichende Größe für dieses Label festzulegen:

```
Composite composite = new Composite(parent, SWT.NONE);
composite.setLayout(new RowLayout());

Label label = new Label(composite, SWT.NONE);
label.setLayoutData(new RowData(100, SWT.DEFAULT));
```

API FillLayout

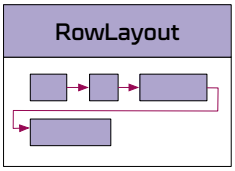
Das *FillLayout* verteilt sämtlichen zur Verfügung stehenden Platz gleichmäßig. Dabei können folgende Optionen gesetzt werden:

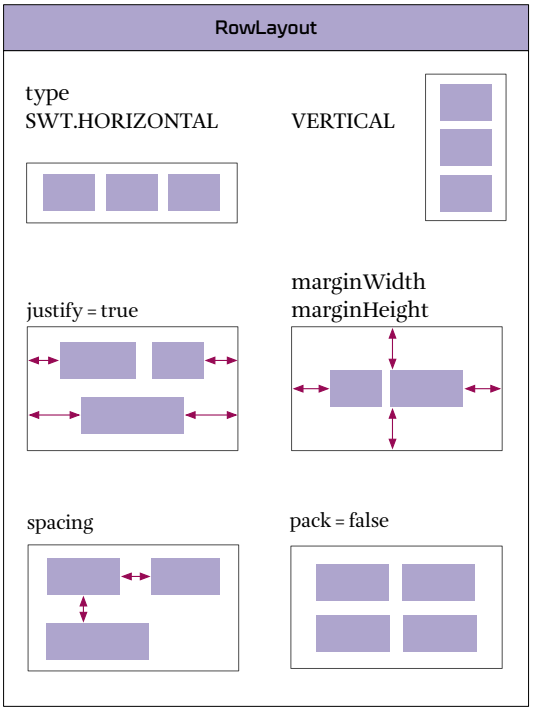


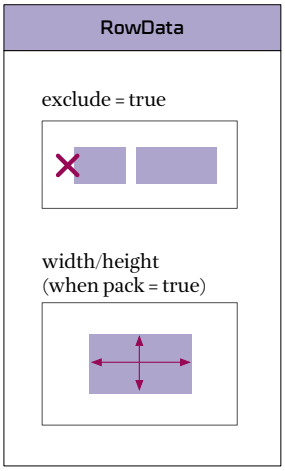


API RowLayout

RowLayout richtet alle Controls zeilen- oder spaltenweise aus und bricht ggf. um:

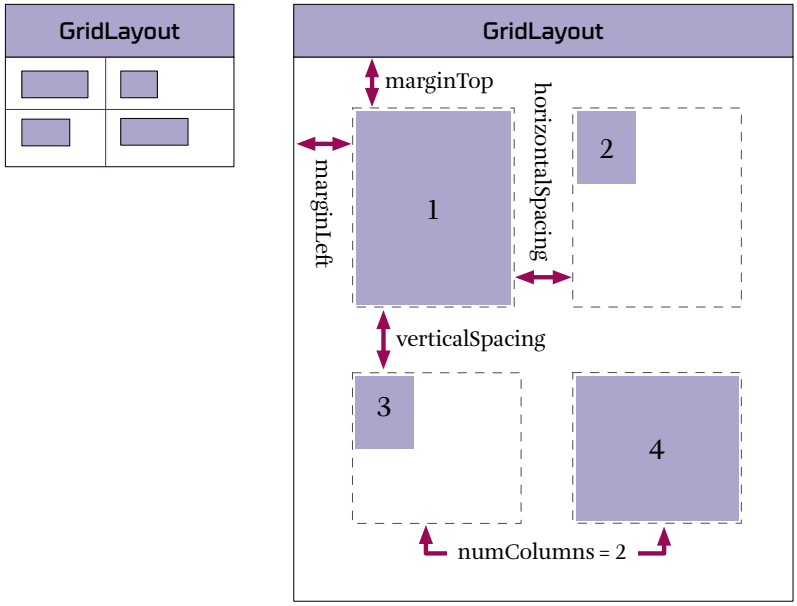




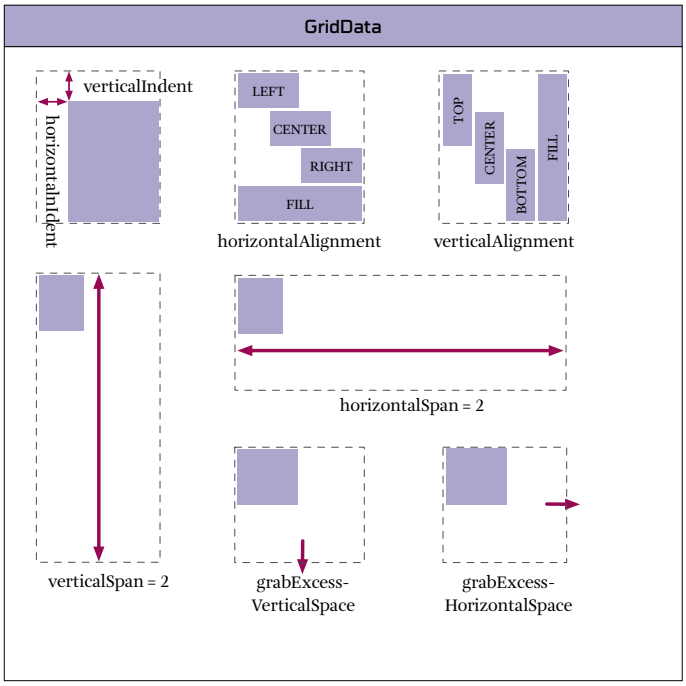


API GridLayout

GridLayout positioniert alle Controls von links-nach-rechts, oben-nach-unten in einer tabellarischen Anordnung. Mit folgenden Optionen können Sie das Layout steuern:



Das Layout für eine einzelne Tabellenzelle können Sie über das *GridData*-Objekt beeinflussen:



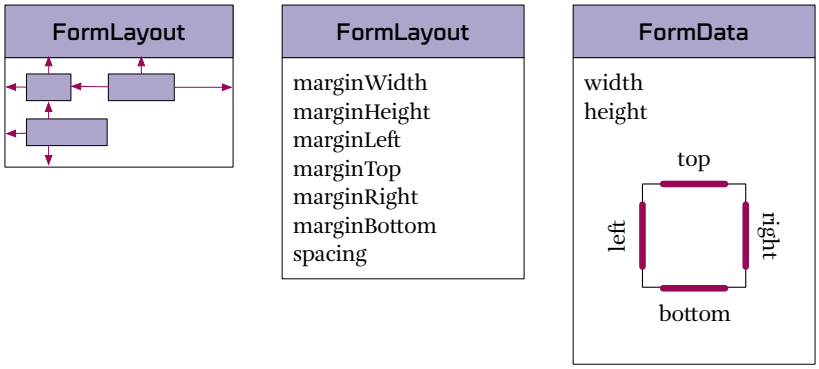
Tipp: Um *GridLayout* und *GridData*-Objekte erzeugen, können die JFace-Klassen *GridLayoutFactory* und *GridDataFactory* verwendet werden. Diese stellen ein API bereit, mit dem Sie diese Objekte mit Einzeilern erzeugen und an Controls zuweisen können:

```
GridLayoutFactory.swtDefaults().numColumns(3).margins(10, 5).applyTo(someComposite);
```

Leider werden die *LayoutFactory*-Klassen aktuell nicht von WindowBuilder unterstützt.

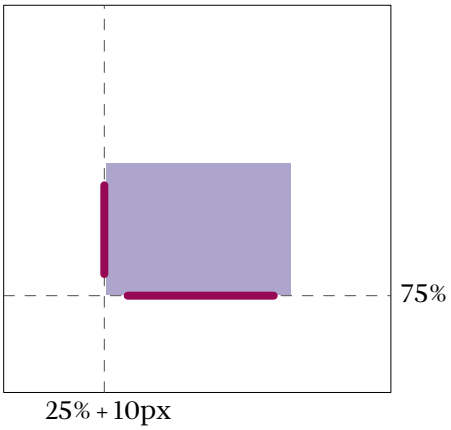
API FormLayout

FormLayout richtet Controls relativ zu den Kanten anderer Controls oder zu den Kanten des beinhaltenden Composite aus:



Sie können die Kanten eines Controls fixieren, indem Sie dem *FormData* für *left*, *top*, *right* oder *bottom* ein *FormAttachment* setzen. Alle verbleibenden Kanten werden automatisch berechnet. Die einfachste Möglichkeit ist die prozentuale Positionierung relativ zu den Kanten des umgebenden Composite:

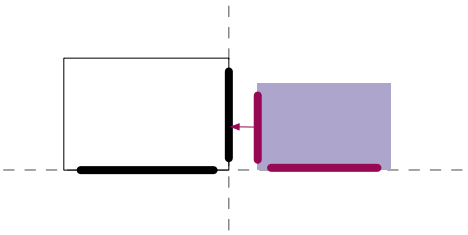
```
FormData formData = new FormData();
// Linke Kante von control auf 25% der Gesamtbreite + 10px Offset fixieren
formData.left = new FormAttachment(25, 10);
// Untere Kante von control auf 75% der Gesamthöhe fixieren
formData.bottom = new FormAttachment(75);
control.setLayoutData(formData);
```



Alternativ können Sie mit dem Konstruktor *new FormAttachment(control, offset, alignment)* eine Kante relativ zu einer Kante eines anderen Controls fixieren:

```
FormData formData = new FormData();
formData.left = new FormAttachment(otherControl, 10, SWT.RIGHT);
formData.bottom = new FormAttachment(otherControl, 0, SWT.BOTTOM);
control.setLayoutData(formData);
```

Hier wird die linke Kante mit 10 Pixel Abstand relativ zur rechten Kante von *otherControl* fixiert. Die untere Kante wird ohne Abstand direkt auf die untere Kante von *otherControl* gelegt:



Offset und Ausrichtung sind dabei optional. Wird keine Ausrichtung angegeben, wird an der entgegengesetzten Kante ausgerichtet (wenn Sie z.B. die *linke* Kante fixieren, wird diese an der *rechten* Kante des anderen Controls ausgerichtet).

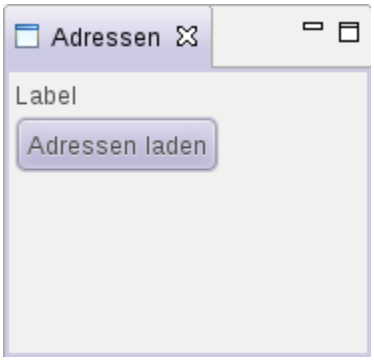
Weitere Informationen

- WindowBuilder Pro
  http://code.google.com/javadevtools/wbpro/index.html
- Understanding Layouts in SWT
  http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html
- Constructing SWT Layouts
  http://www.developer.com/java/other/article.php/3340621/Constructing-SWT-Layouts.htm
- GridLayout Snippets
  http://www.eclipse.org/swt/snippets/#gridlayout

TUTORIAL 7.1

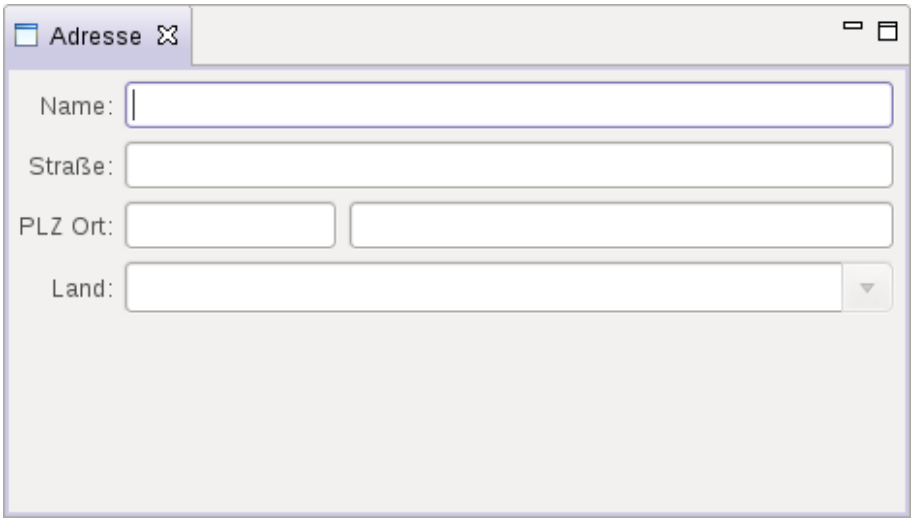
Adressmaske erstellen

☐ Verwenden Sie ein *RowLayout* für die Adressen-View:

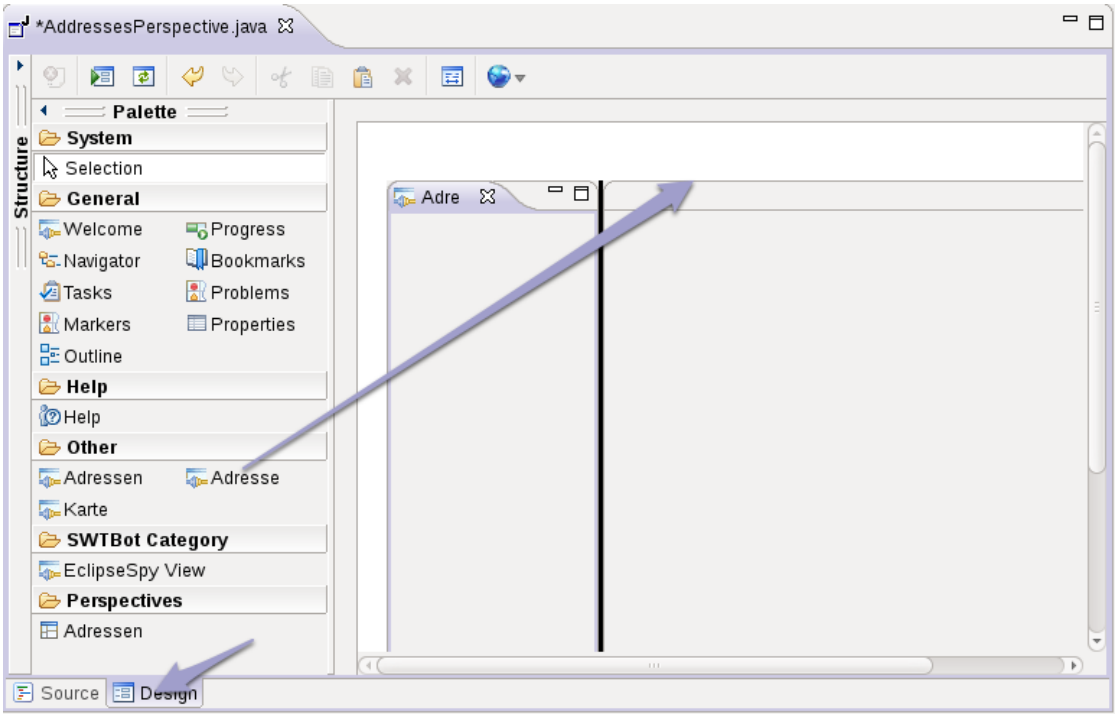


☐ Erstellen Sie ein neues View *Adresse*. Sie können dazu analog zu “Anwendung um ein Adressen-View erweitern” (Seite 23) den Extension Editor verwenden oder das View mit *File › New › Other › WindowBuilder › SWT Designer › RCP › ViewPart* erstellen. Verwenden Sie als View-ID *com.example.addressbook.Address*, als Name für den View-Reiter *Adresse* und *com.example.addressbook.views.AddressView* als Klassenname.

☐ Befüllen Sie das View mit *WindowBuilder* mit einer Adressmaske wie abgebildet. Verwenden Sie dabei ein *GridLayout*, so dass die Eingabefelder aneinander ausgerichtet sind und mit dem View skalieren:



- ☐ Öffnen Sie die Perspektivklasse *AddressesPerspective* im WindowBuilder (ggf. Rechtsklick auf die Perspektivklasse, *Open With* › *WindowBuilder Editor*) und fügen Sie das soeben angelegte View *Adresse* in die Perspektive ein:



KAPITEL 8

OSGi: Modularität der Eclipse-Plattform

Java stellt von Haus aus nur begrenzte Möglichkeiten bereit, Anwendungen in Module aufzuteilen und diese untereinander abzugrenzen. Packages erlauben zwar die Gruppierung von Klassen - zur Laufzeit gelten jedoch keine Restriktionen für die Sichtbarkeit von Packages untereinander. Auch für die Sichtbarkeit der Inhalte eines Packages nach außen gelten kaum Einschränkungen. Die einzige Option, Klassen als *package-protected* zu markieren, ist für die Strukturierung und Abgrenzung von Softwaremodulen meist unzureichend.

Dieses Problem wird von *OSGi* adressiert: Die *OSGi Service Plattform* ist eine Spezifikation für ein Framework, welches die Entwicklung von modularen Anwendungen auf der Java-Plattform ermöglicht. Anwendungen für die OSGi-Plattform bestehen aus voneinander isolierten Modulen, sogenannten *Bundles*. Bundles haben zur Laufzeit einen Lebenszyklus und können dynamisch geladen, gestartet und gestoppt werden, es ist also z.B. ein Update von Teilen der Anwendung ohne einen Neustart der gesamten Anwendung möglich. Ferner können Bundles über *Services* zusammenarbeiten.

Die OSGi-Spezifikation wird von einem Industriekonsortium, der *OSGi Alliance*, entwickelt und veröffentlicht. Es gibt verschiedene Implementierungen dieser Spezifikation. Die Referenzimplementierung stellt Eclipse Equinox, weitere Open-Source-Implementierungen sind Knopflerfish und Apache Felix.

Eclipse-Anwendungen bestehen aus OSGi-Bundles und werden auf der OSGi-Implementierung Eclipse Equinox ausgeführt.

Bundles

Ein Modul einer OSGi-basierten Anwendung bezeichnet man als Bundle. In der Eclipse-Welt wird bedeutungsgleich der Begriff “Plug-in” verwendet. Ein Bundle wird als JAR-Archiv verteilt und besteht aus einer Beschreibung des Bundles (das Manifest), Java-Klassen und Ressourcen:

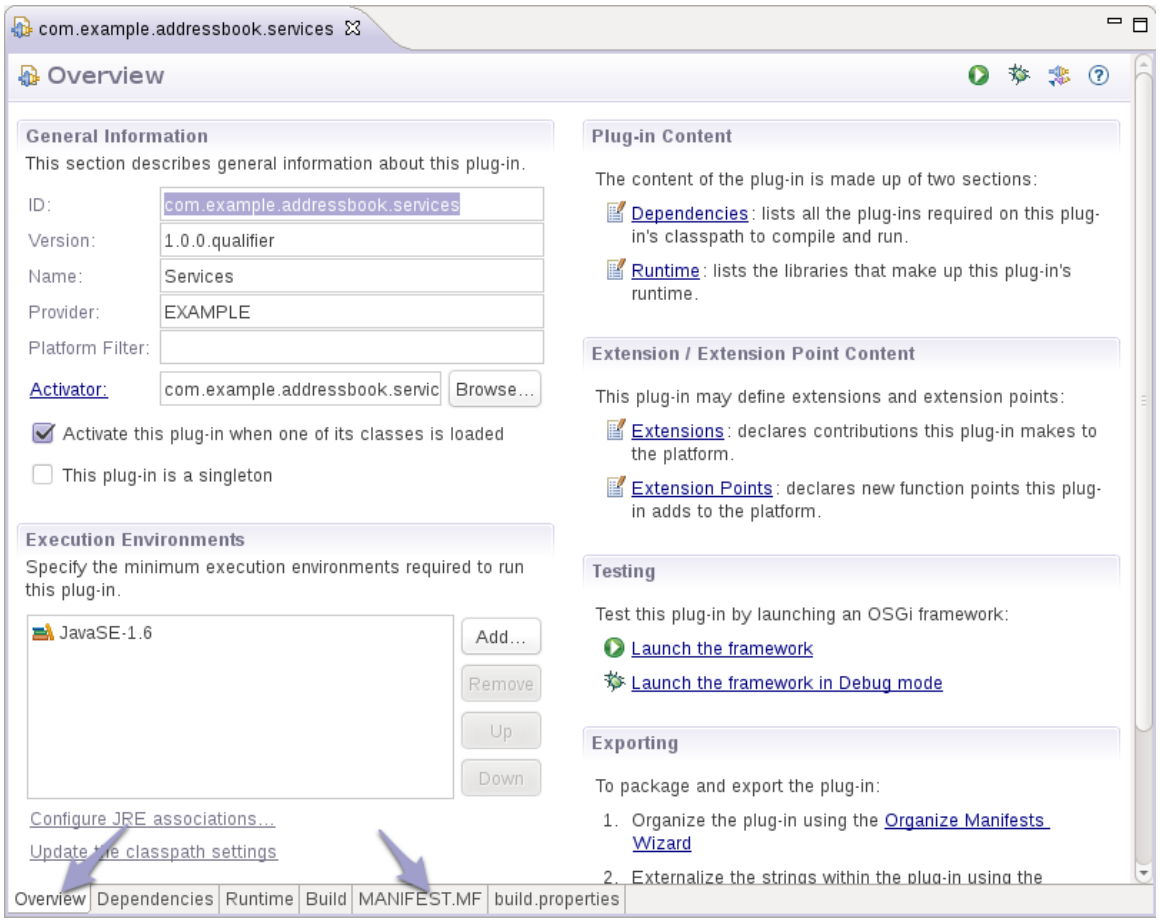
```
`-- com.example.addressbook.services_1.0.0.jar
    |-- META-INF
    |   `-- MANIFEST.MF
    |-- com
    |   `-- example
    |       |-- addressbook
    |           |-- services
    |               |-- Activator.class
```

Der Manifest-Datei *META-INF/MANIFEST.MF* kommt dabei eine besondere Rolle zu. Sie beschreibt das Bundle:

```
Bundle-Name: AddressBook Services
Bundle-SymbolicName: com.example.addressbook.services
Bundle-Version: 1.0.0
Bundle-Activator: com.example.addressbook.services.Activator
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

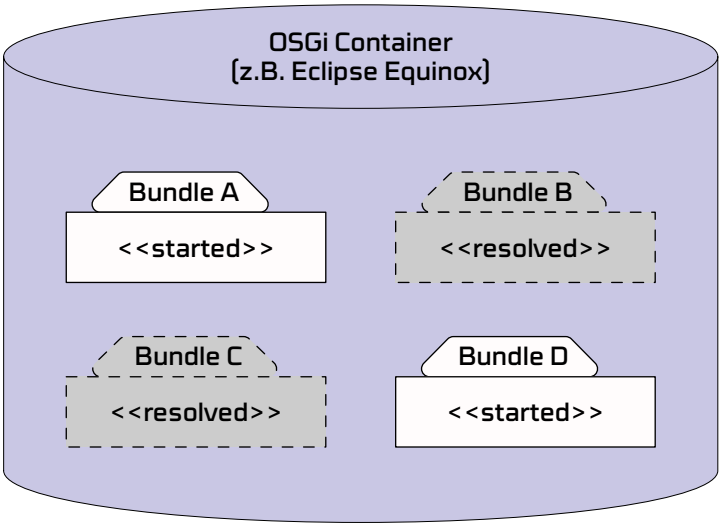
Bundle-Name ist eine kurze, aussagekräftige Beschreibung des Bundles. *Bundle-SymbolicName* ist ein eindeutiger Bezeichner für das Bundle, über den das Bundle referenziert wird. *Bundle-Version* gibt die Versionsnummer des Bundles an. Diese wird immer dann erhöht, wenn eine neue Version des Bundles veröffentlicht wird. *Bundle-Activator* spezifiziert optional eine *Activator*-Klasse, die benachrichtigt wird, wenn das Bundle startet oder stoppt. *Bundle-RequiredExecutionEnvironment* gibt die Mindest-Laufzeitumgebung an, die benötigt wird, um das Bundle auszuführen.

Öffnen Sie die Manifest-Datei, werden die Inhalte des Manifests in einem grafischen Editor dargestellt. Über den Reiter "MANIFEST.MF" können Sie den das Manifest als Textdatei anzeigen:

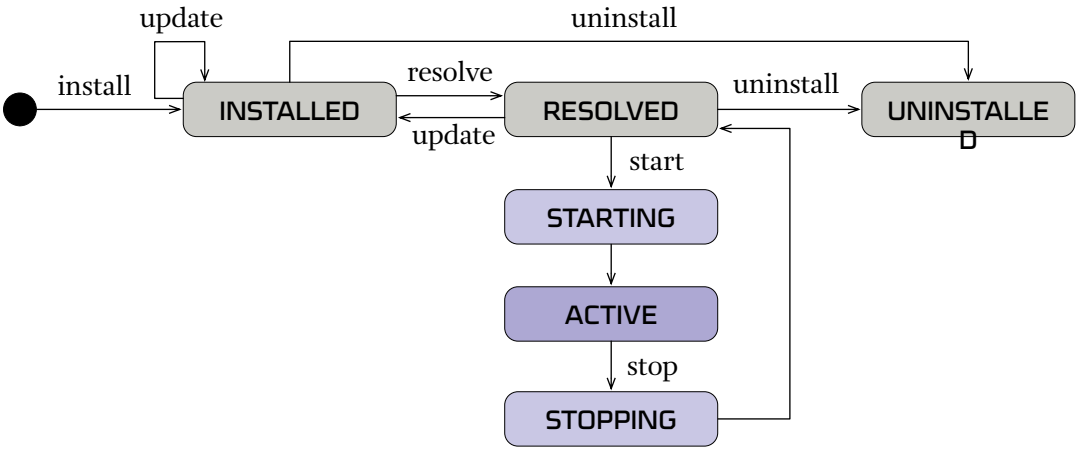


OSGi Container

OSGi-Implementierungen stellen eine Laufzeitumgebung, einen Container zur Ausführung von Bundles, bereit:



Bundles haben innerhalb eines solchen OSGi Containers einen Lebenszyklus. Sie können geladen bzw. installiert werden und befinden sich zunächst im Zustand *RESOLVED*. Wird ein Bundle gestartet, bekommt es den Status *STARTING* und ist nach erfolgreichem Start *ACTIVE*. Bundles können auch wieder beendet (*STOPPING*, dann *RESOLVED*) und deinstalliert werden. Das bedeutet, Bundles können dynamisch nachgeladen oder entfernt werden. Damit ist auch die Aktualisierung eines Bundles zur Laufzeit möglich, ohne die gesamte Anwendung neu zu starten.



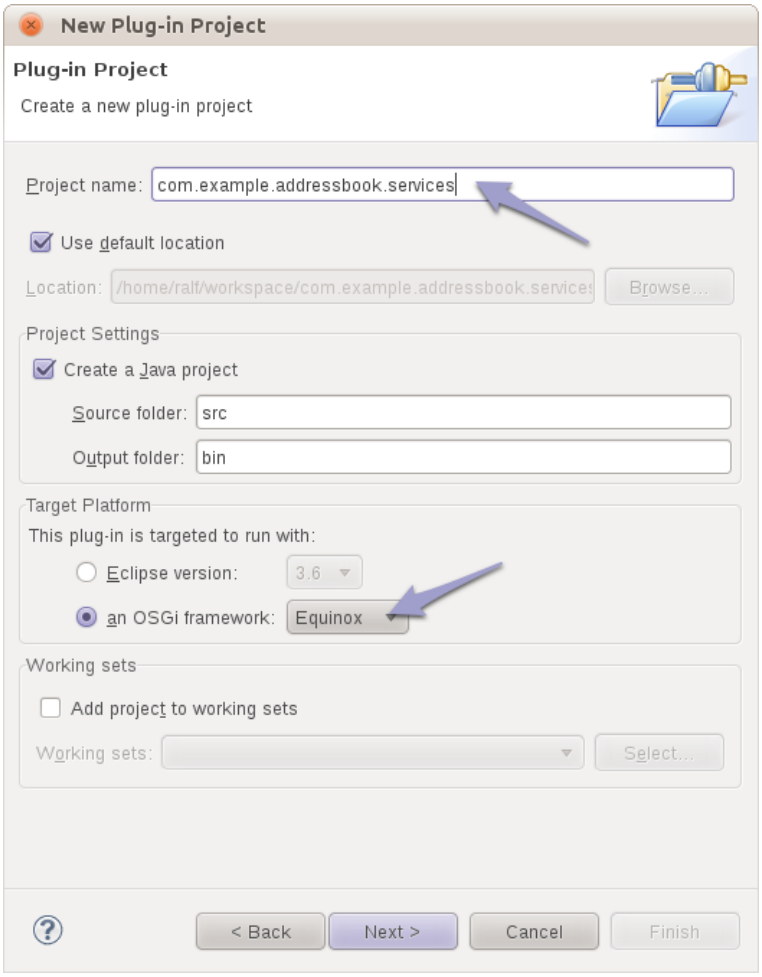
Weitere Informationen zu OSGi

- Die OSGI Service Platform - Eine Einführung mit Eclipse Equinox
  http://www.amazon.de/dp/389864457X
- OSGi Service Platform Specifications
  http://www.osgi.org/Download/HomePage
- Equinox Resources
  http://www.eclipse.org/equinox/resources.php
- Equinox QuickStart Guide
  http://www.eclipse.org/equinox/documents/quickstart.php
- Eclipse Version Numbering
  http://wiki.eclipse.org/index.php/Version_Numbering
- OSGi Tutorial
  http://www.vogella.de/articles/OSGi/article.html
- JAX 2009 Vortragsfolien "Einführung in die OSGi Service Plattform"
  http://www.it-agile.de/fileadmin/docs/Vortragsfolien/OSGi-Powerworkshop-JAX2009.pdf
- Eclipse Summit Europe 2009 Vortragsfolien "OSGi Versioning and Testing"
  http://www.slideshare.net/caniszczyk/osgi-versioning-and-testingppt
- Equinox OSGi Webinar von Jeff McAffer and Chris Aniszczyk
  http://eclipse.dzone.com/articles/equinox-osgi-webinar-online
- Artikel "Eclipse, Equinox and OSGi" von Jeff McAffer und Simon Kaegi, Januar 2007
  http://www.theserverside.com/tt/articles/article.tss?l=EclipseEquinoxOSGi

TUTORIAL 8.1

Ein OSGi Bundle erstellen und starten

- ☐ Wählen Sie *File* > *New* > *Project* > *Plug-in Development* > *Plug-in Project*. Geben Sie *com.example.addressbook.services* als Name für das Plug-in an und wählen *OSGi Framework Equinox* aus, da dieses Bundle keine Eclipse-spezifischen Features wie z.B. Extensions und Extension Points benötigt:

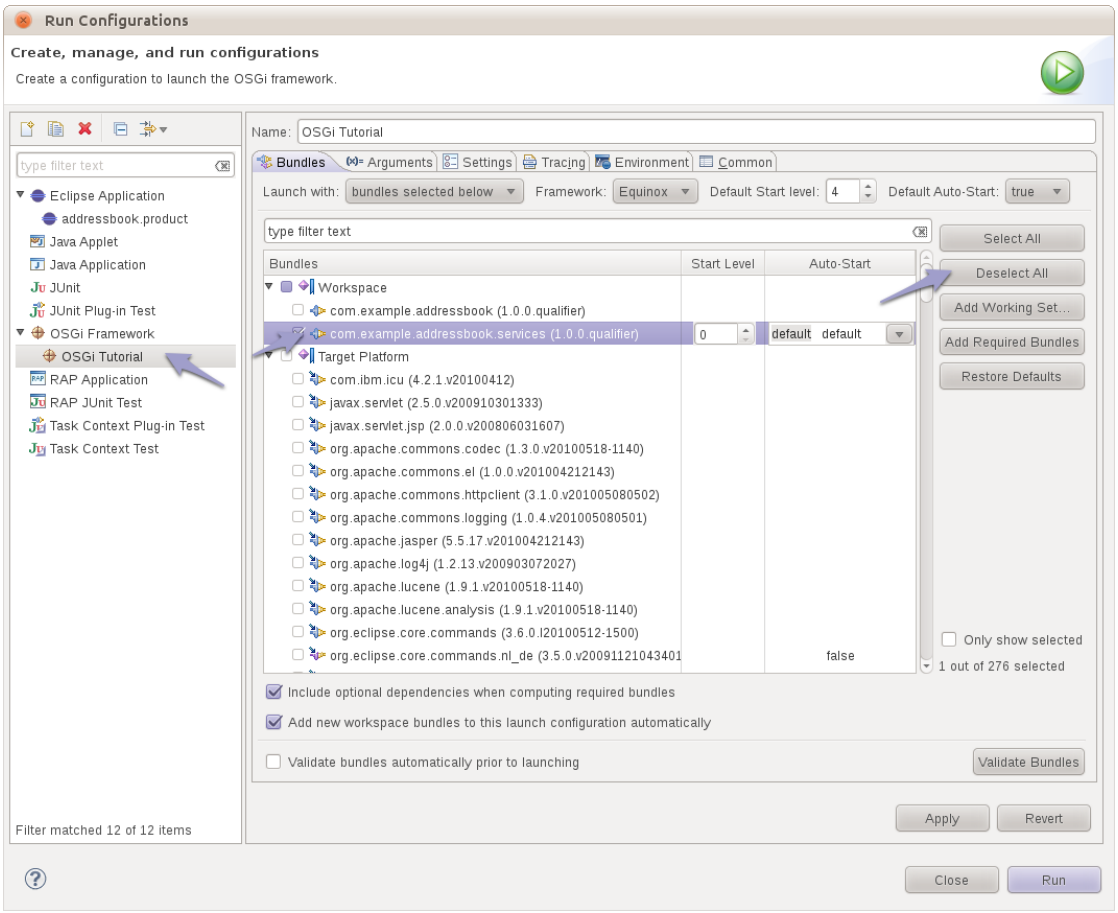


- ☐ Belassen Sie alle anderen Einstellungen und wählen die Option *Create a plug-in using one of the templates* ab, um ein leeres Projekt zu erstellen.

☐ Ergänzen Sie die generierte *Activator*-Klasse um jeweils ein *System.out.println*-Statement, um den Lebenszyklus des Bundles auf der Konsole verfolgen zu können:

```
public class Activator implements BundleActivator {

    public void start(BundleContext context) throws Exception {
        System.out.println("Starting com.example.addressbook.services");
    }

    public void stop(BundleContext context) throws Exception {
        System.out.println("Stopping com.example.addressbook.services");
    }

}
```

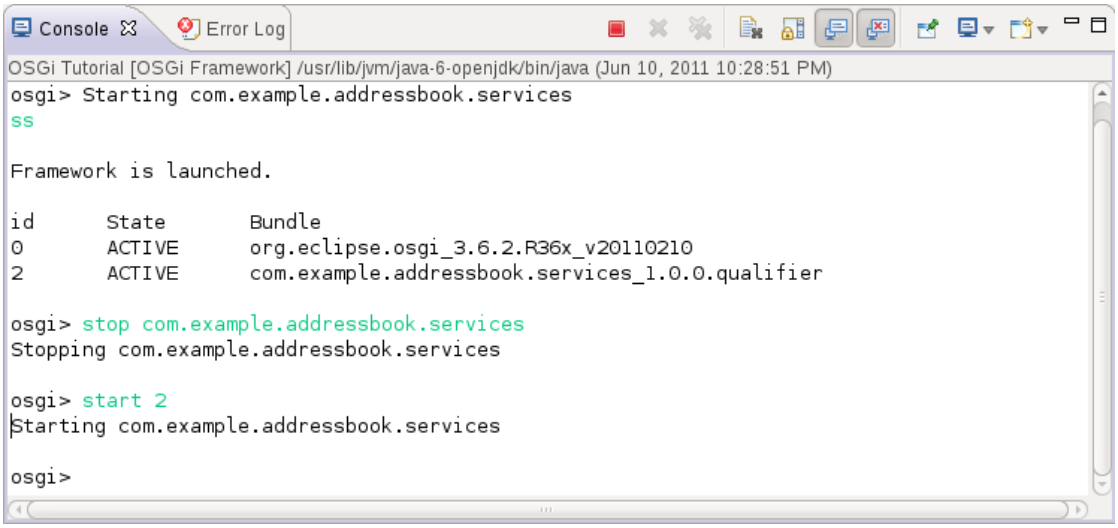
☐ Erzeugen Sie mit *Run* › *Run Configurations* › *OSGi-Framework* testweise eine neue Start-Konfiguration. Klicken Sie *Deselect All* und selektieren nur das soeben erzeugte Bundle:



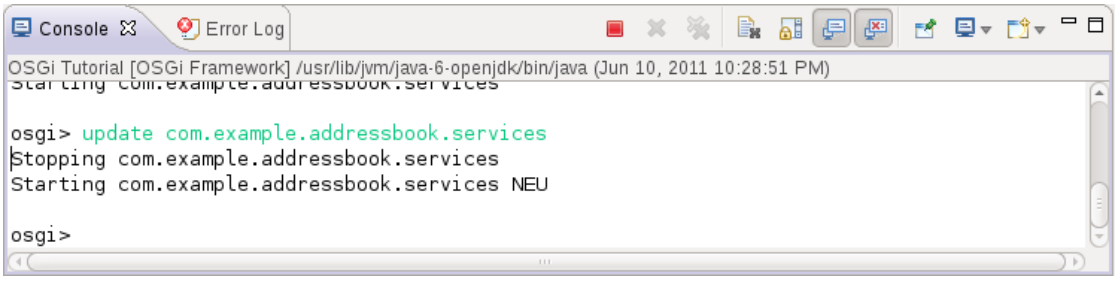
☐ Starten Sie den OSGi-Container über die erstellte Startkonfiguration. Sie sollten in der Konsole die Activator-Ausgabe sowie den OSGi-Prompt *osgi*› sehen.

- ☐ Geben Sie in der Konsole *help* ein, um alle Kommandos des OSGi-Containers aufzulisten.

- ☐ Rufen Sie mit *ss* eine Liste aller gestarteten Bundles auf und stoppen und starten Sie das erstellte Bundle:



- ☐ Verändern Sie die Ausgaben in der *Activator*-Klasse und laden Sie diese Änderung mit dem *update*-Kommando ohne die Anwendung neu zu starten:



KAPITEL 9

OSGi: Abhängigkeiten zwischen Bundles

Require-Bundle

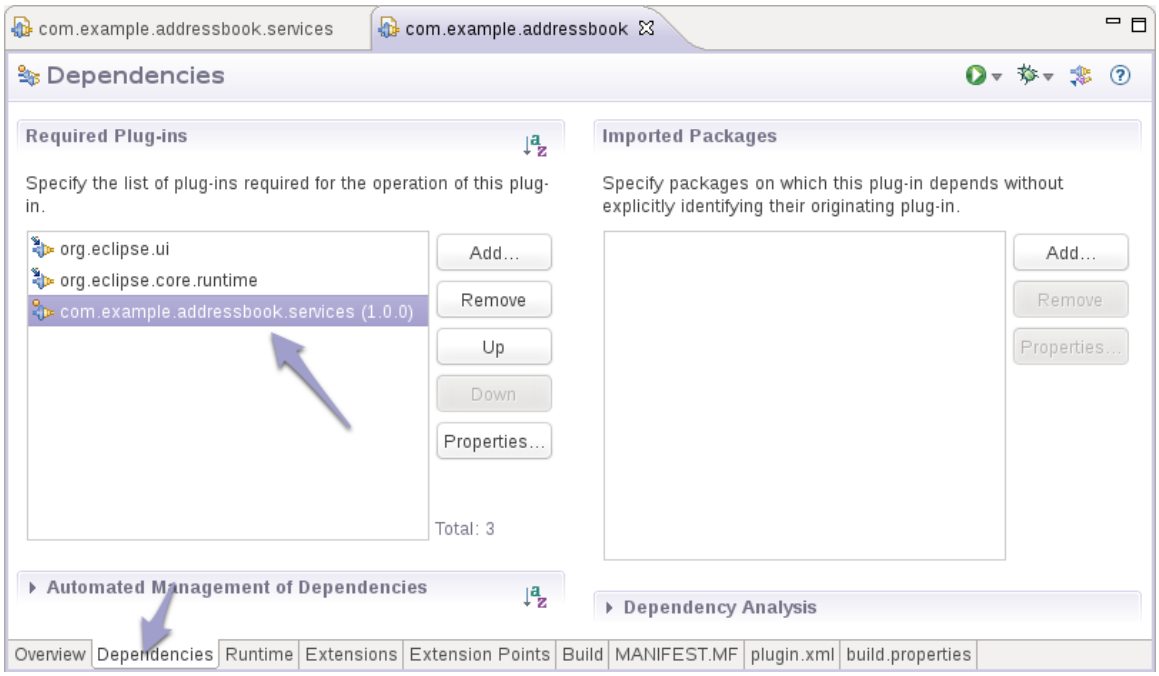
Die Sichtbarkeiten von Klassen bzw. Packages von Bundles sind in der OSGi-Umgebung eingeschränkt, jedes Bundle hat seinen eigenen *Classpath* und ist von den anderen Bundles isoliert. Klassen in einem Bundle können also nicht ohne weiteres Klassen in einem anderen Bundle verwenden.

Dazu muss eine Abhängigkeit deklariert werden, d.h. das Bundle muss definieren, welche anderen Bundles es benötigt, um zu funktionieren. Dies erfolgt über die Manifest-Option *Require-Bundle*:

```
Bundle-SymbolicName: com.example.addressbook
Require-Bundle: com.example.addressbook.services
```

Diese Angabe bedeutet: *com.example.addressbook* benötigt das Bundle *com.example.addressbook.services*. Steht dieses nicht zur Verfügung, kann das Bundle nicht starten.

Die Liste der Abhängigkeiten eines Bundles kann über den Reiter *Dependencies* konfiguriert werden:

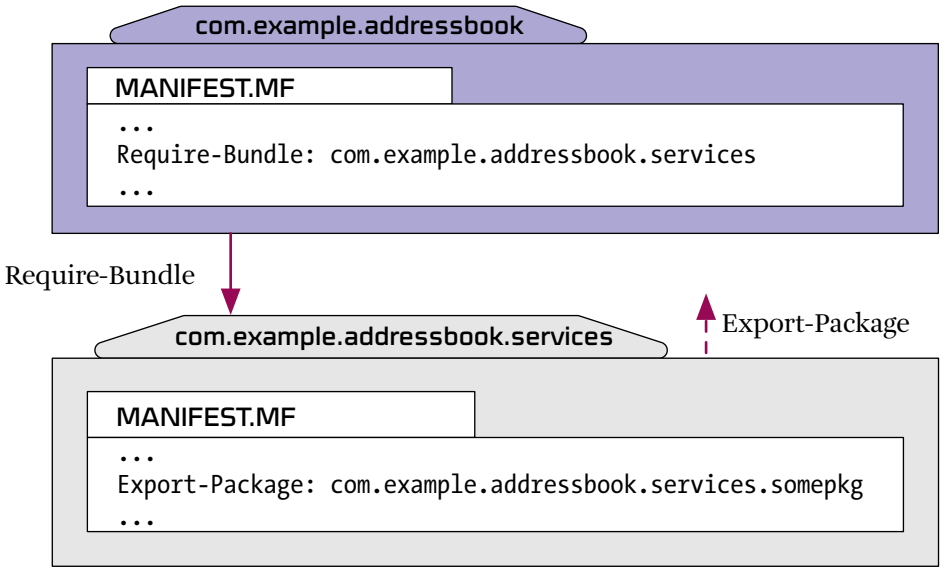


Export-Package

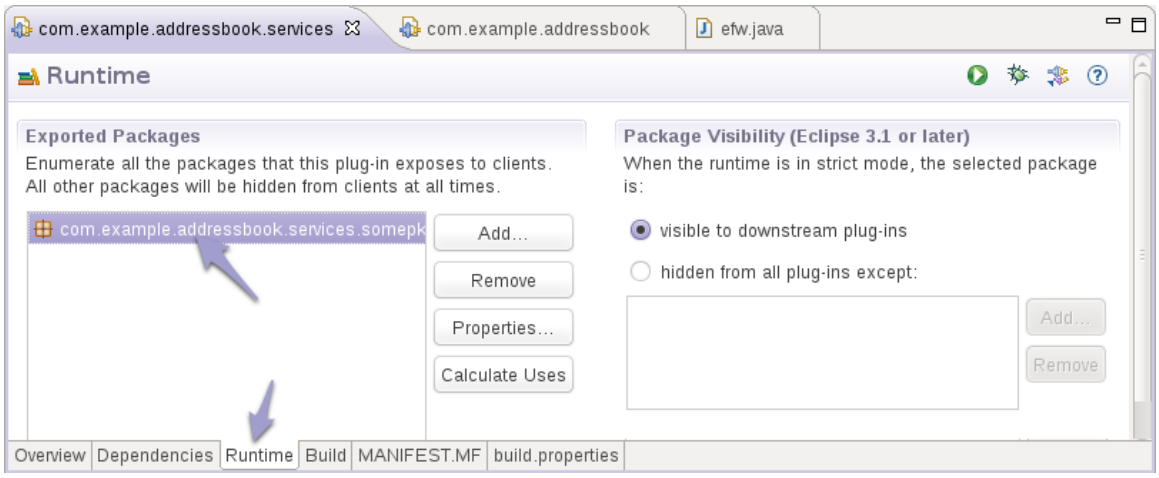
Ein Bundle kann außerdem mit der Manifest-Option *Export-Package* Java-Packages exportieren, z.B.

```
Bundle-SymbolicName: com.example.addressbook.services
Export-Package: com.example.addressbook.services.somepkg
```

Das bedeutet, dass Klassen im Package *com.example.addressbook.services.somepkg* exportiert werden, d.h. für Plug-ins mit einer Abhängigkeit auf das Services-Plug-in sichtbar sind:



Die Liste der exportieren Packages kann über den Reiter *Runtime* konfiguriert werden:



Import-Package

Neben *Require-Bundle* gibt es mit *Import-Package* eine weitere Manifest-Option, mit der eine Abhängigkeit zu einem anderen Bundle spezifiziert werden kann:

```
Bundle-SymbolicName: com.example.addressbook
Import-Package: com.example.addressbook.services.somepkg
```

Durch die Angabe einer Abhängigkeit mit *Import-Package* wird die Entscheidung für ein konkretes Bundle dem OSGi-Framework überlassen. Im Beispiel wird lediglich ein Bundle gefordert, welches das Package *com.example.addressbook.services.somepkg* exportiert. Der OSGi-Container wird zur Laufzeit die Entscheidung, welches Bundle durch die Abhängigkeit auf das Package angezogen wird, treffen. Auf diesem Weg kann man die Unabhängigkeit von einer konkreten Implementierung erreichen - im Beispiel spielt es keine Rolle, welches Bundle das Package bereitstellt.

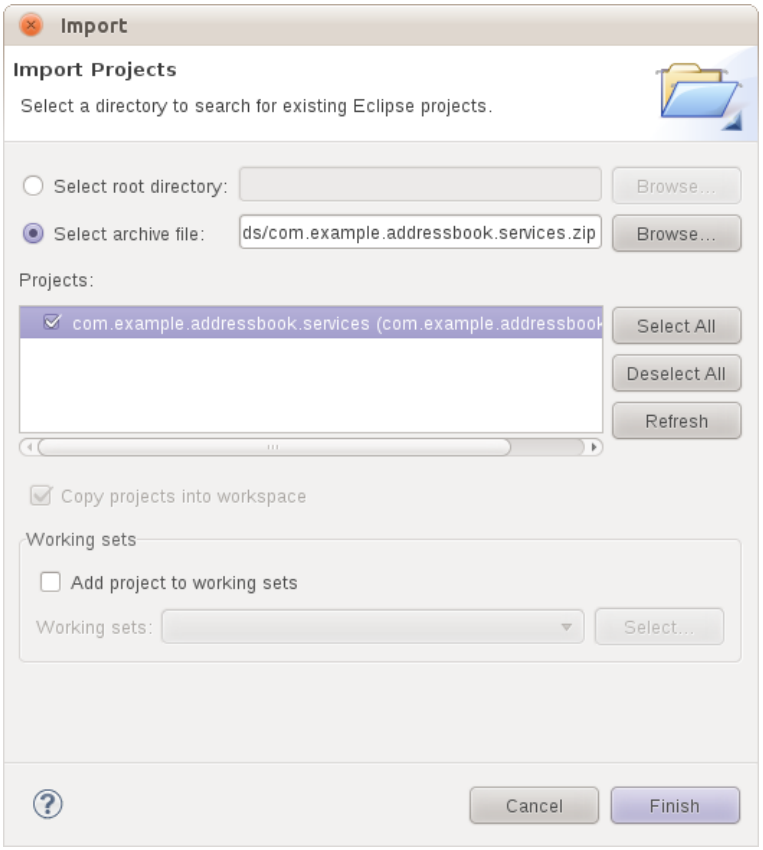
TUTORIAL 9.1

Plug-in zur Datenhaltung einbinden

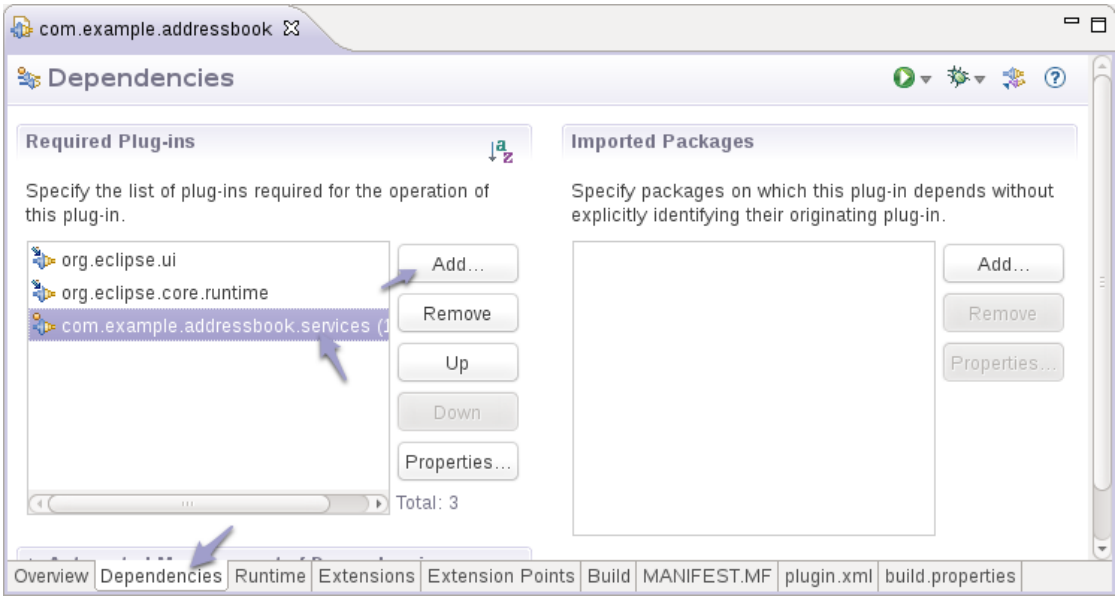
- ☐ Für die Datenhaltung des Adressbuchs steht ein fertiges Bundle *com.example.addressbook.services* zur Verfügung, welches die Datenhaltung für das Adressbuch bereitstellt.

- ☐ Löschen Sie das zuvor erstellte Projekt *com.example.addressbook.services*. Wählen Sie dabei die Option *Delete project contents on disk*.

- ☐ Laden Sie das Plug-in Projekt *com.example.addressbook.services* von http://www.ralfebert.de/eclipse_rcp/downloads/ und importieren es mittels *File* › *Import* › *Existing Projects into Workspace* › *Select archive file* in Ihren Workspace:



☐ Fügen Sie dem Plug-in *com.example.addressbook* eine Abhängigkeit auf das soeben importierte Plug-in hinzu, damit dieses die Service-Klassen verwenden kann:



☐ Passen Sie das *AddressList*-View so an, dass beim Klick auf den Button *Adressen laden* die Adressen folgendermaßen vom *AddressService* geladen werden:

```
IAddressService service = AddressBookServices.getAddressService();
List<Address> allAddresses = service.getAllAddresses();
```

Setzen Sie dem Label als Text die Anzahl der geladenen Adressen (*"... Adressen geladen"*).

☐ Starten Sie die Anwendung testweise. Sie sollten folgende Fehler auf der Konsole sehen:

```
...
!MESSAGE Missing required bundle com.example.addressbook.services_1.0.0.
...
java.lang.RuntimeException: No application id has been found.
...
```

Die Adressbuch-Applikation kann nicht gefunden werden, da das Adressbuch-Plug-in nun eine Abhängigkeit zu dem Service-Plug-in hat, dieses jedoch nicht mit gestartet wird.

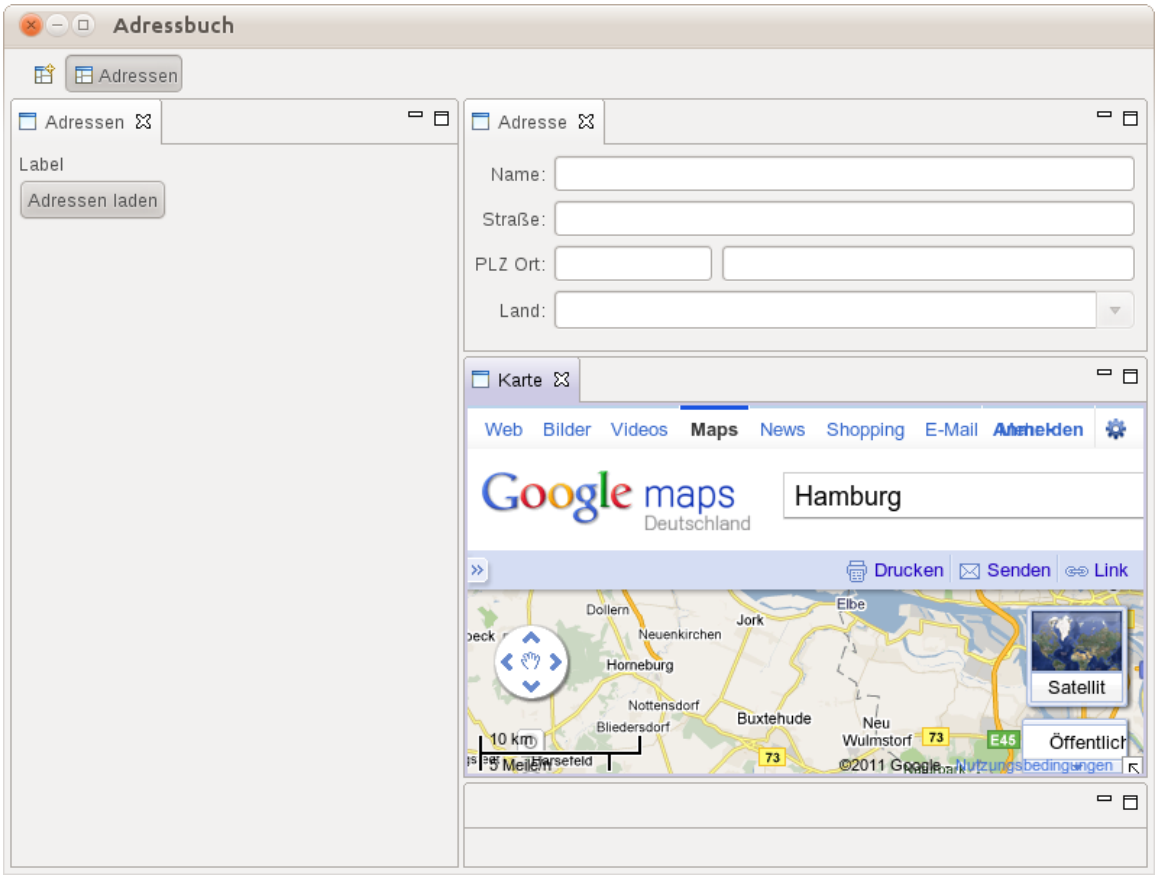
☐ Fügen Sie das Services-Plug-in im Produkt unter *Dependencies* hinzu.

☐ Starten Sie die Anwendung aus dem Produkt heraus mit *Launch an Eclipse Application*, um die Startkonfiguration zu aktualisieren. Die Anwendung sollte nun wieder korrekt starten.

TUTORIAL 9.2

Adressbuch mit einem Karten-Plug-in erweitern

Im Folgenden wird eine Karten-Erweiterung zu dem Adressbuch erstellt, d.h. das Adressbuch wird von einem Plug-in um eine neue View "Karte" erweitert:



ACHTUNG: Dazu sind keinerlei Änderungen in *com.example.addressbook* notwendig! Die Karte wird nur über Erweiterungen in dem neu erstellten Plug-in *com.example.addressbook.maps* realisiert, so dass das Adressbuch auch ohne die Kartenfunktionalität lauffähig ist.

☐ Erstellen Sie ein neues Plug-in-Projekt mit folgenden Einstellungen:

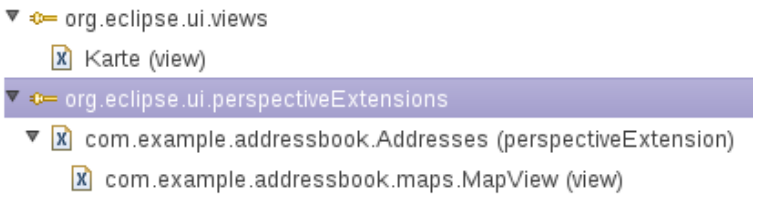
Project Name *com.example.addressbook.maps*
Targeted to run with *Eclipse version 3.7*
Generate an activator aktiviert
Create a rich client application: *No*
Ohne Template

☐ Fügen Sie eine neue View-Extension in dem soeben erstellten Plug-ins hinzu. Legen Sie eine neue Klasse *MapView* für die View an.

- ☐ Fügen Sie ein SWT *Browser* Widget in das Karten-View ein und zeigen darin Google Maps an. Die URL zu einer Google Maps Karte für eine Adresse können Sie setzen mit:

```
browser.setUrl("http://maps.google.de/maps?q=" +
        URLEncoder.encode("Hamburg", "UTF-8"));
```

- ☐ Mit einer Extension zu *org.eclipse.ui.perspectiveExtensions* kann das Karten Plug-in die Karten-View in die bestehende Adressperspektive einhängen. Erstellen Sie dazu im Karten-Plug-in eine Extension zu *org.eclipse.ui.perspectiveExtensions*. Wählen Sie als *targetID* für die *perspectiveExtension* die ID der Adressperspektive aus und fügen Sie ein *view*-Element hinzu, um das Karten-View in Relation zu einem bestehenden View in die Perspektive einzuhängen:

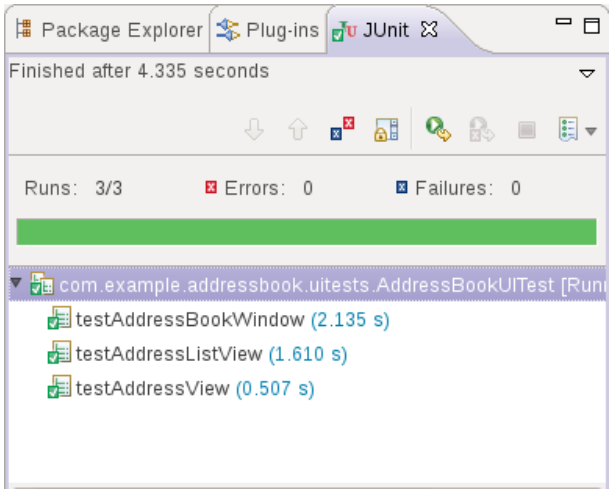


- ☐ Starten Sie die Anwendung mit Karten-Plug-in, indem Sie das Karten-Plug-in in Ihrem Produkt unter *Dependencies* hinzufügen und das Produkt mit *Launch an Eclipse application* starten.

- ☐ Starten Sie die Anwendung einmal ohne Karten-Plug-in, um sicherzustellen, dass Ihre Anwendung auch weiterhin ohne das Karten-Plug-in lauffähig ist.

KAPITEL 10

Automatisierte GUI-Tests mit SWTBot



SWTBot ermöglicht die Durchführung von automatisierten Oberflächentests für SWT- und RCP-Anwendungen. Mit dem API von SWTBot kann dazu die Oberfläche der Anwendung ferngesteuert werden. Da dabei die echte Anwendung auf dem Bildschirm aktiv ist, können alle Feinheiten der Oberfläche unter realistischen Bedingungen getestet werden. So können auch Fälle getestet werden, die nur unter realen Bedingungen nachzuvollziehen sind - beispielsweise Probleme bei der Behandlung des Eingabe-Fokus oder mit Nebenläufigkeiten in der Anwendung.

SWTBot ist ein eigenständiges Eclipse-Projekt, d.h. die SWTBot-Plug-ins müssen in die Target Platform aufgenommen werden. Die Target Plattform aus Tutorial 2: Target Plattform einrichten (Seite 16) erfüllt diese Voraussetzung.

SWTBot wird meist in JUnit-Tests verwendet, die in separaten Plug-ins abgelegt werden. Die Ausführung erfolgt mit einer *JUnit Plug-in Test*-Startkonfiguration, die die Eclipse Anwendung regulär startet und die Oberflächentests parallel zur Anwendung ausführt (in einigen Anleitungen wird noch die Notwendigkeit eines IDE-Test-Runners erwähnt - dieser wird seit Eclipse 3.5 nicht mehr benötigt).

Funktionsweise und API

Der Haupteintrittspunkt in das SWTBot-Framework ist die Klasse *SWTBot*. Diese enthält ein umfangreiches API um SWT-Oberflächen zu bedienen. Für Eclipse RCP-Anwendungen verwendet man die Unterklasse *SWTWorkbenchBot*, die Erweiterungen zur Bedienung der Eclipse Workbench-Oberfläche bereitstellt:

```
private final SWTWorkbenchBot bot = new SWTWorkbenchBot();
```

Mit Methodenaufrufen auf dem Bot kann in den Testmethoden nach Steuerelementen gesucht und Aktionen auf diesen ausgelöst werden. Dazu werden die die *[widget]With[Condition]* Methoden verwendet, z.B.:

```
SWTBotText textField1 = bot.textWithLabel("Name:");
SWTBotText textField2 = bot.textWithTooltip("Name");
```

Die *[widget]WithLabel*-Methoden finden Steuerelemente, die auf ein Label mit der angegebenen Beschriftung folgen. So erhält man ein *SWTBot[Widget]*-Objekt, auf dem man nun Aktionen ausführen kann:

```
textField1.setFocus();
textField1.typeText("Hello");
```

SWTBot löst diese Aktionen aus, indem es Ereignisse auf der SWT Ereignis-Warteschlange einstellt. Aus Sicht der Anwendung sind diese Ereignisse nicht von der Interaktion mit einem realen Anwender zu unterscheiden. Praktisch ist zudem, dass bei den Abfragemethoden immer für eine kurze Zeit auf das jeweilige Widget gewartet wird. Es macht also nichts, wenn ein Widget erst in Folge der vorhergehenden Aktion erscheint. Wird beispielsweise ein Dialog geöffnet, könnte man mit *bot.shell("title")* auf diesen Dialog warten und dann in dem Dialog weiterarbeiten. Alle Operationen haben standardmäßig ein Timeout von 5 Sekunden, d.h. es wird maximal 5 Sekunden auf das Ereignis gewartet.

Steuerelemente referenzieren

Mit der Möglichkeit, Controls anhand ihrer Beschriftungen zu lokalisieren, können Tests nah an der Vorgehensweise eines realen Anwenders formuliert werden. Nachteil dabei ist jedoch, dass die Tests angepasst werden müssen, wenn sich Beschriftungen ändern. Zudem entsteht bei internationalisierten Anwendungen Mehraufwand bzw. die Tests können nur für eine Sprache ausgeführt werden. Wurde die Applikation mit dem Eclipse-NLS-Mechanismus übersetzt, können Sie dieses Problem lösen, indem Sie die *Messages*-Konstantenklassen exportieren und in den Tests verwenden. Eine alternative Möglichkeit besteht darin, für die Controls der Anwendung IDs zu vergeben, die SWTBot den Weg weisen:

```
someControl.setData("org.eclipse.swtbot.widget.key", "someId")
```

Mit *setData* werden einem SWT Control Zusatzinformationen angefügt. Der *key* hierfür ist mit der Konstante *SWTBotPreferences.DEFAULT_KEY* definiert. Diese sollten Sie in Ihrer Anwendung jedoch nicht verwenden, um keine Abhängigkeit zu SWTBot einzuführen. Optional deklarieren Sie eine eigene Konstante für diesen Wert. So markierte Controls können Sie mit den *[widget]WithId*-Methoden lokalisieren. Dies funktioniert besonders robust, da das Control eindeutig markiert und identifiziert wurde. Auch wenn Sie Ihre Maske umbauen, wird keine Änderung am Testcode notwendig:

```
SWTBotText textField3 = bot.textWithId("someId");
```

Menüs

Ebenfalls erfreulich einfach gelöst ist die Bedienung von Menüs. Dieses erreichen Sie über die *SWTBot.menu()*-Methoden. Analog erfolgt die Steuerung von Kontextmenüs über *contextMenu()*, beispielsweise:

```
// Hauptmenü-Eintrag finden und klicken
bot.menu("Datei").menu("Neu").click();

// Tabelleneintrag selektieren und Kontextmenü bedienen
SWTBotTable table = bot.table();
table.select("Heike Winkler");
table.contextMenu("Adresse öffnen").click();
```

Auf die Workbench zugreifen

Für das Testen von RCP-Anwendungen ist vor allem die zielstrebige Bedienung der Eclipse Workbench entscheidend. Mit den Methoden von *SWTWorkbenchBot* können gezielt die Elemente der Eclipse Workbench angesteuert werden. So können Views oder Editoren anhand ihrer ID oder dem Reiter-Titel aufgefunden werden. Ist noch kein passender Reiter geöffnet, wartet SWTBot auch hier kurze Zeit auf das Erscheinen, nach einem Timeout wird eine *WidgetNotFoundException* geworfen, die den Test zum Fehlschlagen bringt.

Views und Editoren lokalisieren Sie folgendermaßen:

```
SWTBotView view1 = bot.viewById("com.example.someViewId");
SWTBotView view2 = bot.viewByTitle("Adressen");

SWTBotEditor editor1 = bot.editorById("com.example.someEditorId");
SWTBotEditor editor2 = bot.editorByTitle("Hans Schmidt");
```

Sobald Sie ein solches View- oder Editor-Objekt zur Verfügung haben, können Sie mit der *bot()*-Methode einen SWTBot erhalten, der auf die Inhalte dieses View bzw. Editor-Reiters beschränkt ist. So können alle Methoden von SWTBot verwendet werden, um in den Inhalten des Reiters zu navigieren:

```
SWTBot editorBot = editor1.bot();
editorBot.textWithLabel("Name:").setText("Heike Muster");
```

Das API von *SWTBotView* und *SWTBotEditor* stellt zusätzlich Methoden bereit, um den Reiter selbst zu bedienen. Beispielsweise können Editoren gespeichert und geschlossen werden:

```
assertTrue(editor1.isDirty());
editor1.save();
assertFalse(editor1.isDirty());
editor1.close(),
```

Auch vor Perspektiven und Commands macht der SWTBot bei der Fernsteuerung der Eclipse Workbench keinen Halt. Das Prinzip ist dabei analog zu den bereits gezeigten Methoden.

Robuste Tests schreiben

Analog zur Entwicklung von regulären Tests ist es auch für die erfolgreiche Umsetzung und Pflege von SWTBot-Testsuiten entscheidend, für jeden Testfall saubere und definierte Ausgangsbedingungen zu gewährleisten. Theoretisch müsste man die gesamte Anwendung vor jeder Testmethode neu starten, um eine unangetastete Umgebung zu verwenden und so die Unabhängigkeit der Tests untereinander zu garantieren. Da dies die Ausführungszeit der Tests sehr in die Höhe treiben würde, muss hier ein wenig improvisiert werden.

In jedem Fall sollten die Testmethoden voneinander unabhängig gehalten werden. Kein Testfall sollte auf dem vorherigen aufbauen, da sonst fehlschlagende Tests häufig alle nachfolgenden Tests mitreißen und die Möglichkeit fehlt, einzelne Tests schnell zu verifizieren.

Idealerweise bringen Sie mit einer *@Before*-Methode die Anwendung vor jedem Testfall in einen ausreichend definierten Zustand. Dazu kann die Methode *SWTBot#resetWorkbench* verwendet werden, die die Workbench soweit möglich in ihren Ursprungszustand zurücksetzt.

Ein weiteres Problem sind die Testdaten der Anwendung. Auch hier sollte jede Testmethode einen sauberen Ausgangszustand vorfinden. Sofern hinter der Oberfläche der Anwendung eine Datenbank- oder Service-Schicht steckt, empfiehlt sich die Wiederverwendung von hier ggf. vorhandenen Testdaten. So könnten die Tests der GUI-Anwendung über einen Aufruf um die Bereitstellung von Testdaten bitten. So können auch alle existierenden Werkzeuge verwendet werden, z.B. das *DBUnit*-Framework. Eine weitere empfehlenswerte Variante ist das Schreiben von client-seitigen Mocks für die Backend-Schnittstellen. Voraussetzung ist jedoch, dass das

Backend selbst bereits gut getestet ist, da die Tests der GUI-Anwendung das Backend-System dann nicht mehr mit testen.

Auch wenn das Aufsetzen der Test-Voraussetzungen zu Beginn häufig mühevoll ist, lohnt (und rechnet) sich der Aufwand in den meisten Fällen durch kürzere Laufzeiten der Testsuiten und zuverlässigere, stabilere Testergebnisse.

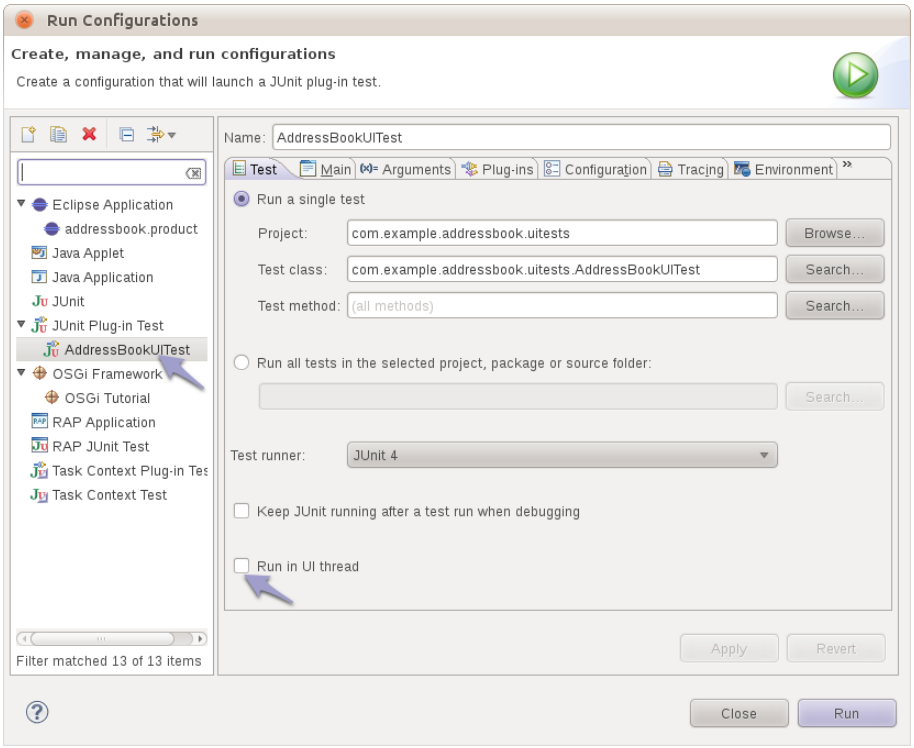
Weitere Informationen

- SWTBot – UI Testing for SWT and Eclipse
  http://www.eclipse.org/swtbot/
- Contributing to SWTBot
  http://wiki.eclipse.org/SWTBot/Contributing
- Eclipse GUI Testing Is Viable With SWTBot, Blog-Post von David Green
  http://greensopinion.blogspot.com/2008/09/eclipse-gui-testing-is-viable-with.html
- Eclipse Summit Europe 2009: OSGi Versioning and Testing
  http://www.slideshare.net/caniszczyk/osgi-versioning-and-testingppt
- The Hamcrest Tutorial
  http://code.google.com/p/hamcrest/wiki/Tutorial
- WindowTester Pro: UI test generation tool for testing SWT and Swing Java applications
  http://code.google.com/javadevtools/wintester/html/index.html
- Growing Object-Oriented Software, Guided by Tests
  http://www.amazon.com/Growing-Object-Oriented-Software-Guided-Tests/dp/0321503627

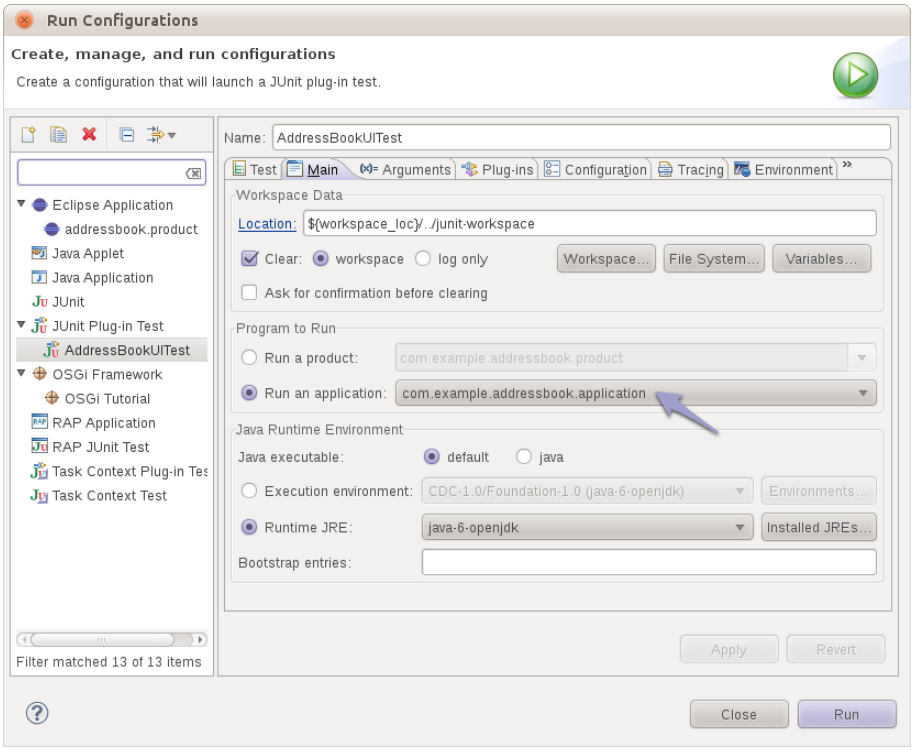
TUTORIAL 10.1

GUI-Tests ausführen

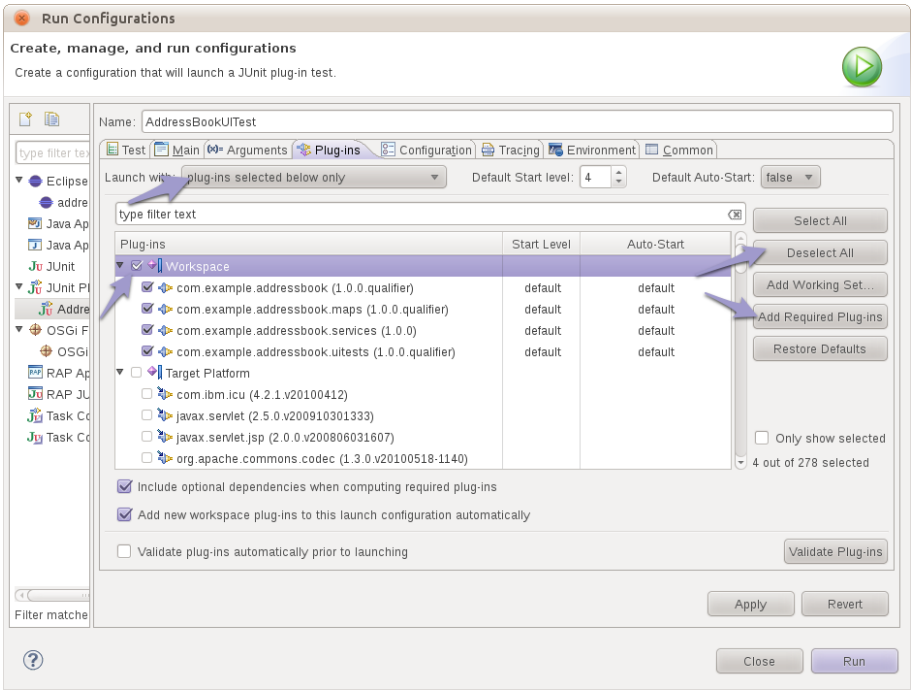
- ☐ Laden Sie das *uitest*-Plug-in von http://www.ralfebert.de/eclipse_rcp/downloads/ und importieren Sie es mit und importieren es mittels *File › Import › Existing Projects into Workspace › Select archive file* in Ihren Workspace.
- ☐ Öffnen Sie die Testklasse *com.example.addressbook.uitests.AddressBookUITest*.
- ☐ Die Testklasse verwendet die Konstantenklasse aus Tutorial 5: Anwendung konfigurieren (Seite 32). Exportieren Sie daher im Adressbuch-Plug-in das entsprechende Package und importieren Sie das Package im *uitest*-Plugin.
- ☐ Führen Sie den Test per Rechtsklick auf die Testklasse mit *Run as › JUnit Plug-in Test* aus. Dadurch wird eine neue Startkonfiguration angelegt, die noch anzupassen ist.
- ☐ Öffnen Sie die soeben erstellte Startkonfiguration für *AddressBookUITest* und wählen unter *Test* die Option *Run in UI thread* ab, da SWTBot-Tests nicht auf dem UI-Thread ausgeführt werden dürfen:



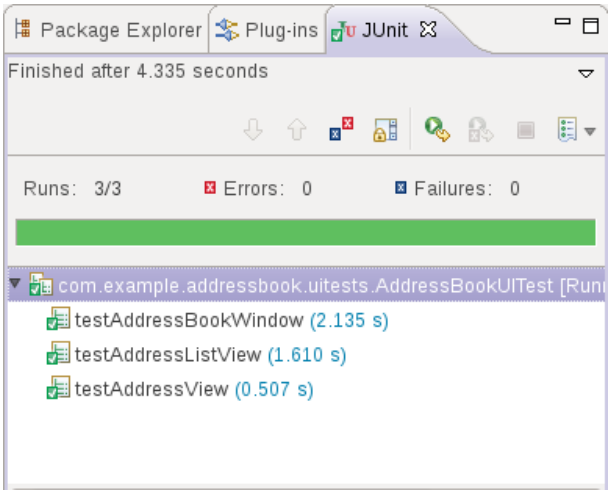
☐ Wählen Sie unter *Main* › *Run a product / application* die zu startende RCP-Anwendung:



☐ Unter *Plug-ins* wählen Sie *Launch with: plug-ins selected below only*, klicken Sie *Deselect All*, haken alle Workspace-Plug-ins an und klicken *Add Required Plug-ins* um alle benötigten Plug-ins automatisch hinzuzufügen.



- Dadurch wird die RCP-Anwendung gestartet und nach der Ausführung der Test-Methoden wieder beendet. Sollte dabei etwas schief gehen, untersuchen Sie das Fehlerprotokoll der Anwendung und prüfen insbesondere die Startkonfiguration auf fehlende Abhängigkeiten. Bei fehlschlagenden Tests gelangen Sie per Doppelklick zu der verursachenden Zeile in der Testklasse. Ggf. ist es notwendig, Beschriftungen oder IDs Ihrer Anwendung anpassen.



KAPITEL 11

Eclipse Job-Framework

UI-Thread

Dem Thread, der das SWT Display erzeugt hat und die Event Loop ausführt, kommt in SWT-Anwendungen eine besondere Rolle zu. Man bezeichnet ihn als *UI-Thread*. Alle Events werden im UI-Thread strikt nacheinander abgearbeitet. Kein Event kann verarbeitet werden, bis der vorhergehende Event Handler fertig abgearbeitet ist.

Daher sind Listener so zu implementieren, dass sie den UI-Thread nur solange wie unbedingt notwendig in Anspruch nehmen. Langlaufende Aktivitäten sollten in Hintergrund-Threads durchgeführt werden.

Wichtig ist zudem, dass der Zugriff auf SWT-Widgets nur dem UI-Thread gestattet ist. Zugriffe auf SWT-Widgets außerhalb des UI-Threads führen zu einer *SWTException*:

```
Exception in thread "Thread-N"
org.eclipse.swt.SWTException: Invalid thread access
```

Mit den Display-Methoden *(a)syncExec* kann ein Runnable auf dem UI-Thread ausgeführt werden, z.B.:

```
display.asyncExec(new Runnable() {

    @Override
    public void run() {
        // Executed on UI-Thread
    }

});
```

Die Methode *syncExec* wartet dabei auf die Abarbeitung des *Runnable*, d.h. der aufrufende Thread wird für diese Zeit blockiert.

In Eclipse RCP-Anwendungen gibt es mit dem *Job-Framework* eine praktische Alternative zur Verwendung von *Threads* bzw. *Display.(a)syncExec*. Um Aktivitäten im Hintergrund auszuführen, kann ein *Job* implementiert und durch den Aufruf von *schedule* dem Job-Scheduler zur Abarbeitung übergeben werden:

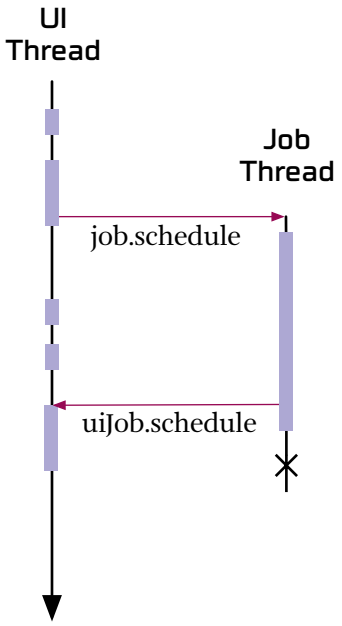
```
Job job = new Job("Test") {

    protected IStatus run(IProgressMonitor monitor) {
        // Long-running operation here
        return Status.OK_STATUS;
    }

};
job.schedule();
```

Um außerhalb der Event-Verarbeitung (z.B. aus Hintergrund- oder Timer-Threads) auf das UI zugreifen, kann analog ein *UIJob* verwendet werden:

```
UIJob uiJob = new UIJob("Update UI") {

    public IStatus runInUIThread(IProgressMonitor monitor) {
        // Update UI here
        return Status.OK_STATUS;
    }

};
uiJob.schedule();
```

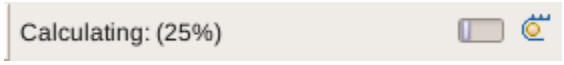
Job / UIJob Threading

Ein typisches Anwendungsszenario für das Job-Framework in RCP Anwendungen ist es, länger dauernde Aktivitäten als Job im Hintergrund auszuführen, so dass in der Zwischenzeit Ereignisse auf dem UI-Thread verarbeitet werden können. Die Job stellen nach Fertigstellung dann bei Bedarf einen *UIJob* ein, der die Benutzeroberfläche aktualisiert:



Fortschrittsanzeige für Jobs

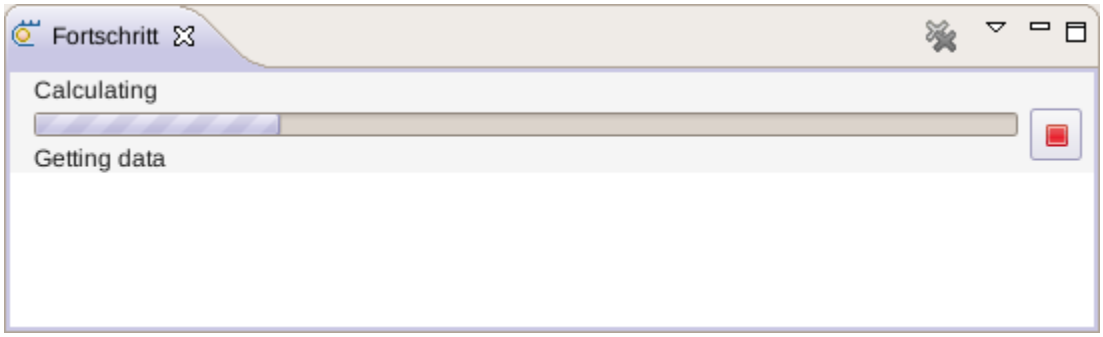
Die Eclipse Workbench kann eine Fortschrittsanzeige für laufende Jobs anzeigen:



Diese Anzeige muss lediglich im *WorkbenchWindowAdvisor* aktiviert werden:

```
configurer.setShowProgressIndicator(true);
```

Zusätzlich kann die aus Eclipse bekannte View zur Fortschrittsanzeige in RCP-Anwendungen eingebunden werden:



Dazu ist das *progressView* als View der Anwendung zu registrieren. Als Klasse wird die *ExtensionFactory* angegeben, die das entsprechende View erzeugt:

```
<extension point="org.eclipse.ui.views">
    <view
        id="org.eclipse.ui.views.ProgressView"
        class="org.eclipse.ui.ExtensionFactory:progressView"
        icon="/icons/pview.gif"
        allowMultiple="false"
        name="Progress"/>
</extension>
```

Weitere Informationen

- Decorating Jobs with Icons
  http://eclipsophy.blogspot.com/2009/12/decorating-your-jobs.html
- UIProcess: Job + UIJob
  http://github.com/ralfebert/rcputils/blob/master/de.ralfebert.rcputils/src/de/ralfebert/rcputils/concurrent/UIProcess.java

TUTORIAL 11.1

Adressen im Hintergrund laden

- ☐ Beobachten Sie das Verhalten der Anwendung während des Ladens der Adressliste nach dem Klick auf den *Adresse laden*-Button: Das Laden der Adressen ist künstlich verlangsamt um einen Netzwerkzugriff zu simulieren. In dieser Zeit reagiert die Anwendung nicht, da die Operation direkt im Listener, d.h. auf dem UI-Thread, ausgeführt wird.

- ☐ Extrahieren Sie den Code, der den Label-Text setzt mit *Refractor › Extract Method* in eine neue Methode *setAddressList(List<Address> allAddresses)*.

- ☐ Extrahieren Sie den Code, der die Adressen lädt, mit *Refractor › Extract Method* in eine neue Methode *refresh()*. Rufen Sie hier *setAddressList* auf, sobald die Daten zur Verfügung stehen.

- ☐ Verlagern Sie das Laden der Adressen in den Hintergrund, indem Sie in der *refresh*-Methode einen Job einstellen, der die Adressen lädt, statt die Adressen direkt zu laden.

- ☐ Stellen Sie sicher, dass die Methode *setAddressList* auf dem UI-Thread ausgeführt wird, indem Sie hier einen UIJob einstellen, der die Benutzeroberfläche aktualisiert.

- ☐ Aktivieren Sie die Anzeige der Status-Bar und der Fortschrittsanzeige für die Anwendung in der *WorkbenchWindowAdvisor*-Klasse.

- ☐ Entkommentieren Sie die Test-Methode *testLoadAddressesJob* in *AddressBookUITest* und führen Sie alle Tests aus.

KAPITEL 12

JFace Überblick, JFace Structured Viewers

SWT beschränkt sich auf die Bereitstellung eines Java-APIs für die zugrunde liegenden nativen GUI-Bibliotheken. Darüber hinausgehende Funktionalität wird von JFace angeboten. Hier finden sich unter anderem:

- *JFace Structured Viewer*: Tabellen, Bäume, Listen und Comboboxen mit Daten befüllen
- *ResourceManager*: Verwaltung von Image, Color, Font-Ressourcenobjekten
- Basisklassen für Dialoge und Assistenten
- *ControlDecoration*: Dekoration von SWT Controls mit Bildern
- *FieldAssist*: Automatische Eingabevervollständigung
- *JFace Data Binding*

Ressourcenfreigabe: Farben, Fonts und Schriften

SWT-Objekte binden Systemressourcen, die nicht von der Java Garbage Collection aufgeräumt werden können. Daher ist es wichtig, dass diese Objekte durch einen expliziten Aufruf der *dispose*-Methode wieder freigegeben werden, wenn sie nicht mehr benötigt werden.

Für Controls bedeutet dies meist keine gesonderte Behandlung, da der Aufruf von *dispose* auf einem Composite auch alle Kindelemente freigibt. In RCP-Anwendungen wird dies von der Workbench erledigt - sobald die Inhalte eines Views nicht mehr benötigt werden, werden diese freigegeben.

Ressourcenobjekte wie *Color*, *Font* und *Image*-Objekte müssen hingegen manuell durch den Aufruf von *dispose()* freigegeben werden, sobald sie nicht mehr benötigt werden. Denn diese Objekte werden nicht automatisch freigegeben, wenn ein verwendendes Control freigegeben wird. Dies liegt darin begründet, dass sie in verschiedenen Composites verwendet werden könnten.

JFace erleichtert diese Aufgabe mit der Klasse *LocalResourceManager*. Ein *ResourceManager* verwaltet solche Ressourcenobjekte und gibt sie automatisch frei, sobald ein Besitzer-Widget freigegeben wird:

```
LocalResourceManager resources
    = new LocalResourceManager(JFaceResources.getResources(), ownerWidget);
```

ResourceManager funktionieren hierarchisch, das erste Argument des Konstruktors ist der Eltern-*ResourceManager*. Alle Ressourcen in diesem stehen auch dem erzeugten

ResourceManager zur Verfügung. *JFaceResources.getResources()* stellt einige Standard-Ressourcen bereit.

Hat man einen *ResourceManager* erzeugt, kann man diesen zur Erzeugung von Farben, Fonts und Bildern verwenden, die automatisch verwaltet und freigegeben werden:

```
Color color = resources.createColor(new RGB(200, 100, 0));
```

Bilder und Fonts werden über *ImageDescriptor* bzw. *FontDescriptor*-Objekte beschrieben. Diese sind noch keine Ressourcenobjekte (müssen also im Gegensatz zu *Image* und *Font* nicht freigegeben werden). In RCP-Anwendungen können Bilder am einfachsten über die statische Methode *getImageDescriptor* auf dem Activator des Plug-ins verwendet werden:

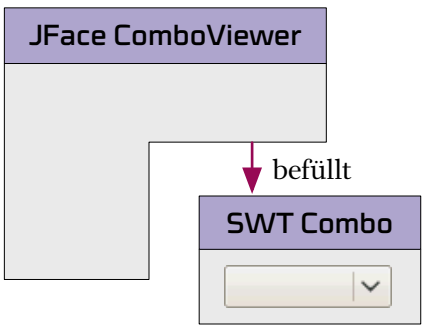
```
Font font = resources.createFont(FontDescriptor.createFrom("Arial", 10, SWT.BOLD));
Image image = resources.createImage(Activator.getImageDescriptor("/icons/someimage.png"));
```

JFace Structured Viewer

JFace Structured Viewer stellen eine Abstraktion bereit, die Inhalte aus beliebigen Java-Datenstrukturen in einem SWT-Widget zur Anzeige bringt. Folgende Viewer stehen zur Verfügung:

- JFace *TableViewer* → SWT *Table*
- JFace *ComboViewer* → SWT *Combo*
- JFace *ListViewer* → SWT *List*
- JFace *TreeViewer* → SWT *Tree*

Ein JFace *ComboViewer* ist ein JFace Structured Viewer. Er ist für die Verwaltung der Inhalte eines SWT Combo-Controls zuständig (analog JFace *ListViewer* und SWT *List*):



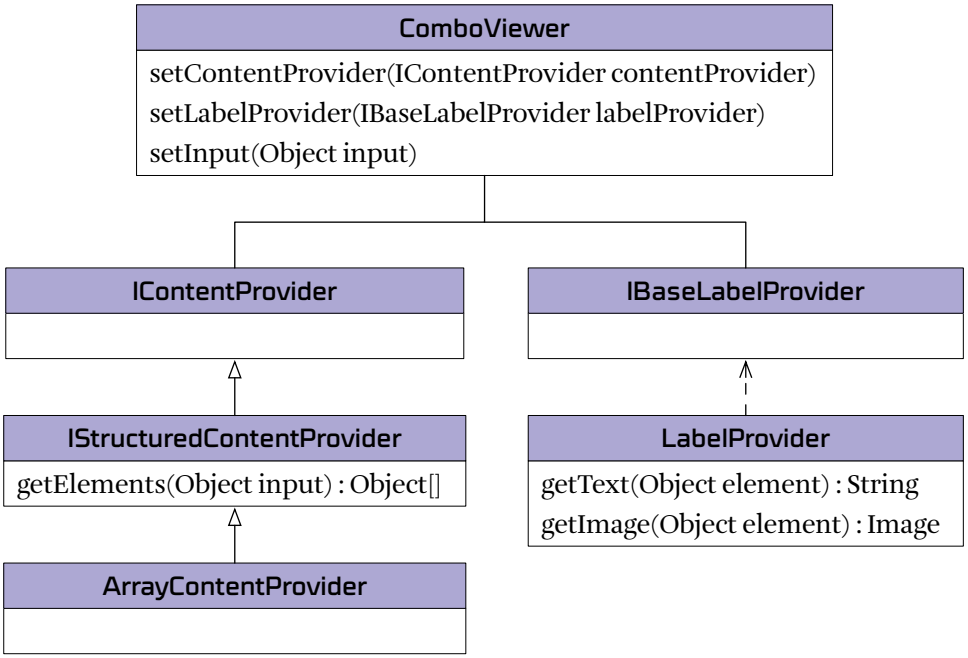
Ein *ComboViewer* kann zu einem bereits vorhandenen *Combo*-Control erzeugt werden:

```
ComboViewer comboViewer = new ComboViewer(someCombo);
```

Alternativ kann man dem Viewer die Erzeugung des Controls überlassen:

```
ComboViewer comboViewer = new ComboViewer(parentComposite, SWT.READ_ONLY);
Combo combo = comboViewer.getCombo();
```

Ein Viewer wird mit einem *ContentProvider*, *LabelProvider* und *Input*-Objekt versehen. Diese Objekte beschreiben die anzuzeigenden Daten:



Das Input-Objekt repräsentiert die anzuzeigenden Inhalte. Es kann sich um ein beliebiges Java-Objekt handeln:

```
comboViewer.setInput(someData);
```

Woher weiß der *Viewer*, wie mit diesem Eingabe-Objekt umzugehen ist? Dazu wird dem *Viewer* ein *ContentProvider* gesetzt. Der *ContentProvider* ist dafür zuständig, die Daten aus dem Input-Objekt für die Anzeige durch den Viewer bereitzustellen. Liegt das Input-Objekt bereits als Java-*Array* oder Java-*Collection* vor, sollte hierfür der von JFace bereitgestellte *ArrayContentProvider* verwendet werden:

```
comboViewer.setContentProvider(ArrayContentProvider.getInstance());
comboViewer.setInput(someCollection);
```

Weiterhin ist ein *LabelProvider* ist dafür zuständig, zu einem Objekt die zur grafischen Anzeige benötigten Informationen wie Label oder Bild zu liefern:

```
comboViewer.setContentProvider(ArrayContentProvider.getInstance());
comboViewer.setLabelProvider(new LabelProvider() {
    @Override
    public String getText(Object element) {
        SomeObject element = (SomeObject) element;
        return element.getName();
    }
});
comboViewer.setInput(someCollection);
```

Der *setInput*-Aufruf sollte immer zuletzt erfolgen, da es die Befüllung des SWT-Widgets auslöst. Zu diesem Zeitpunkt muss bereits ein *Content*- und *LabelProvider* gesetzt sein.

Selektion für JFace Structured Viewer

An die aktuelle Selektion eines JFace-Viewers gelangen Sie über *getSelection()*. Das zurückgegebene Interface *ISelection* ist ein allgemeines Interface für eine beliebige Selektion. Selektionen, die Elemente beinhalten, werden als *IStructuredSelection* abgebildet. Alle *StructuredViewer* geben eine *IStructuredSelection* zurück. Mittels der *IStructuredSelection* können Sie das erste selektierte Element abfragen oder über alle selektierten Elemente iterieren:

```
ISelection selection = someViewer.getSelection();
if (!selection.isEmpty()) {
    IStructuredSelection structuredSelection = (IStructuredSelection) selection;

    // a) get first element
    SomeObject selectedObject = (SomeObject) structuredSelection.getFirstElement();

    // b) iterate over all elements
    for (Iterator iterator = structuredSelection.iterator(); iterator.hasNext();) {
        Object element = iterator.next();
    }

}
```

Die aktuelle Selektion kann zudem mit einem *ISelectionChangedListener* beobachtet werden:

```
someViewer.addSelectionChangedListener(new ISelectionChangedListener() {
    public void selectionChanged(SelectionChangedEvent event) {
        IStructuredSelection selection = (IStructuredSelection) event.getSelection();
        SomeObject selectedObject = (SomeObject) selection.getFirstElement();
    }
});
```

Weitere Informationen

- JFace Snippets
  http://wiki.eclipse.org/index.php/JFaceSnippets

TUTORIAL 12.1

Länderauswahl

- ☐ Befüllen Sie die Länderliste in der Adressmaske mit einem *JFace ComboViewer*. Eine Liste aller Länder erhalten Sie über *AddressBookServices.getAddressService().getAllCountries()*:

- ☐ Geben Sie eine Meldung auf der Systemkonsole aus, wenn sich die Länderselektion ändert.
- ☐ Führen Sie alle Tests inkl. *testCountryList* aus.

TUTORIAL 12.2

Verzeichnisbaum mit TreeViewer anzeigen

- ☐ Machen Sie sich mittels der JavaDocs mit der *TreeViewer*-Klasse vertraut und finden Sie insbesondere heraus, welche *ContentProvider* für einen *TreeViewer* geeignet sind.
- ☐ Erstellen Sie ein neues Plug-in *com.example.experiments.filebrowser*.
- ☐ Fügen Sie ein neue *Dateien*-Perspektive mit einem *Dateien*-View hinzu.
- ☐ Implementieren Sie das *Dateien*-View so, dass es die Verzeichnisstruktur der Laufwerke Ihres Rechners als Baum anzeigt. Implementieren Sie dazu einen *ITreeContentProvider*, der mit *File* bzw. *File[]*-Objekten umgehen kann. Zur Ermittlung der Verzeichnisstruktur können Sie die *listRoots()*-Methode von *java.io.File* verwenden.
- ☐ Erweitern Sie das View so, dass standardmäßig die erste Ebene des Baumes aufgeklappt ist.
- ☐ Implementieren Sie den *LabelProvider* so, dass für alle Elemente entsprechende Icons angezeigt werden. Zur Ermittlung der Icons kann *Program#findProgram#getImageData* sowie *PlatformUI#getWorkbench#getSharedImages* verwendet werden.

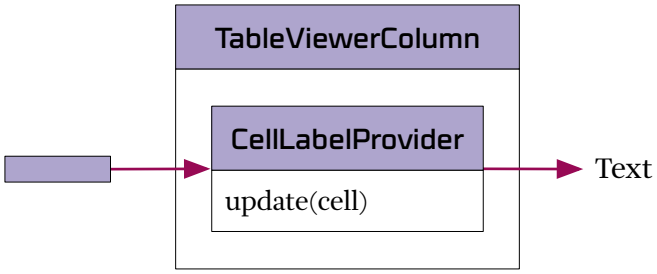
KAPITEL 13

JFace TableViewer

Die Anzeige von Tabellen mit dem JFace *TableViewer* funktioniert analog zum *Combo-/ListViewer*. Neu ist hier, dass ein *LabelProvider* für jede Spalte gesetzt wird. Dazu ist zunächst für jede Spalte ein JFace *TableViewerColumn*-Objekt anzulegen. Dieses erzeugt und verwaltet eine SWT *TableColumn*:

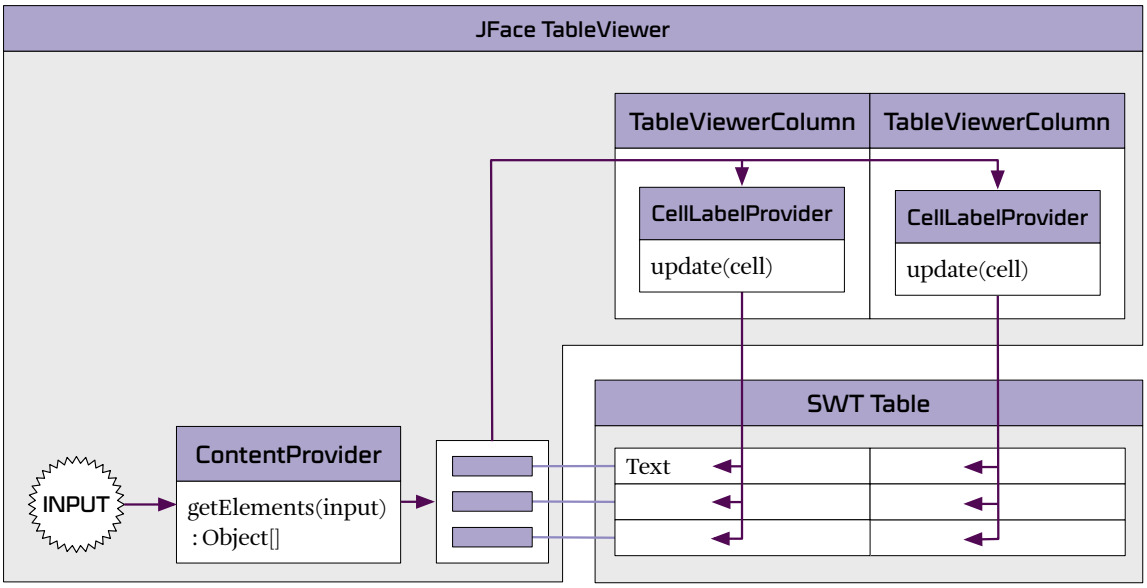
```
TableViewerColumn viewerNameColumn = new TableViewerColumn(tableViewer, SWT.NONE);
viewerNameColumn.getColumn().setText("Some Column");
viewerNameColumn.getColumn().setWidth(100);
```

Einer *TableViewerColumn* muss zudem ein *CellLabelProvider* oder *ColumnLabelProvider* gesetzt werden, der die Zellen dieser Spalte befüllt. Dabei hat er das Element für die jeweilige Tabellenzeile zur Verfügung:




```
TableViewerColumn viewerNameColumn = new TableViewerColumn(tableViewer, SWT.NONE);
viewerNameColumn.setLabelProvider(new ColumnLabelProvider() {

    @Override
    public String getText(Object element) {
        SomeObject person = (SomeObject) element;
        return person.getSomeStringProperty());
    }

});
```

Zusammengefasst: Ein *TableViewer* befüllt eine SWT *Table* mit Inhalten aus seinem *Input*-Objekt. Der *ContentProvider* konvertiert dazu das Input-Objekt in ein Array von Objekten. Jede Spalte des TableViewers bekommt einen *LabelProvider* gesetzt, der die Zellen der Spalte mit den Daten aus dem Zeilenobjekt befüllt:




```
// JFace TableViewer
TableViewer tableViewer = new TableViewer(parentComposite, SWT.NONE);
// SWT Table
Table table = tableViewer.getTable();
// Spaltenköpfe und Zeilenbegrenzungen sichtbar machen
table.setHeaderVisible(true);
table.setLinesVisible(true);

// ArrayContentProvider, da Input-Objekt hier eine Java Collection ist
tableViewer.setContentProvider(ArrayContentProvider.getInstance());

// Für jede Spalte ein TableViewerColumn erzeugen
TableViewerColumn viewerNameColumn = new TableViewerColumn(tableViewer, SWT.NONE);
viewerNameColumn.getColumn().setText("Test");
viewerNameColumn.getColumn().setWidth(100);

// LabelProvider für jede Spalte setzen
viewerNameColumn.setLabelProvider(new ColumnLabelProvider() {
    public String getText(Object element) {
        return ((SomeObject) element).getSomeProperty();
    }
```

```
});

// Input-Objekt setzen
tableViewer.setInput(someList);
```

Tabellen sortieren

Um die Reihenfolge der Zeilen eines JFace *TableViewers* festzulegen, kann dem TableViewer ein *ViewerComparator* gesetzt werden:

```
tableViewer.setComparator(ViewerComparator comparator)
```

Die Sortierung erfolgt dabei auf Anzeigeebene, das zugrunde liegende Input-Objekt bleibt unverändert. Der *ViewerComparator* legt mit seiner *compare*-Methode die Reihenfolge fest:

```
// a) wenn angezeigte Objekte das Interface Comparable implementieren
tableViewer.setComparator(new ViewerComparator());

// b) Reihenfolge explizit mit compare-Operation festlegen
tableViewer.setComparator(new ViewerComparator() {

    @Override
    public int compare(Viewer viewer, Object e1, Object e2) {
        if (e1 instanceof SomeObject && e2 instanceof SomeObject) {
            return ((SomeObject) e1).getName().compareToIgnoreCase(
                   ((SomeObject) e2).getName());
        }
        throw new IllegalArgumentException("Not comparable: " + e1 + " " + e2);
    }

});
```

Filtern

Um die Zeilen eines JFace *StructuredViewers* zu filtern, können die *addFilter* bzw. *setFilters*-Methoden verwendet werden. Ein Objekt wird nur dann zur Anzeige gebracht, wenn alle gesetzten Filter der Anzeige zugestimmt haben. So lässt sich beispielsweise eine Suche über die Inhalte einer Tabelle realisieren:

```
ViewerFilter searchFilter = new ViewerFilter() {

    @Override
    public boolean select(Viewer viewer, Object parentElement, Object element) {
        return ((SomeObject) element).getName().contains("searchString");
    }

};

tableViewer.addFilter(searchFilter);
```

Editierbare Tabellen

Die Spalten in einem JFace-Viewer können editierbar gemacht werden, indem auf der *TableViewerColumn* ein *EditingSupport*-Objekt für die Spalte gesetzt wird. Die *EditingSupport*-Implementierung legt fest, welche Zellen bearbeitbar sein sollen, welches Control zur Bearbeitung verwendet werden soll (im Beispiel *TextCellEditor*) und wie die Werte zu holen und zu setzen sind:

```
tableViewerColumn.setEditingSupport(new EditingSupport(tableViewer) {

    protected boolean canEdit(Object element) {
        return true;
    }

    protected CellEditor getCellEditor(Object element) {
        return new TextCellEditor(tableViewer.getTable());
    }

    protected Object getValue(Object element) {
        return ((SomeObject) element).getName();
    }

    protected void setValue(Object element, Object value) {
        ((SomeObject) element).setName(String.valueOf(value));
```

```
            tableViewer.refresh(element);
        }

});
```

Als *CellEditor* stehen *TextCellEditor*, *ColorCellEditor*, *ComboBoxViewerCellEditor*, *CheckboxCellEditor* sowie *DialogCellEditor* zur Verfügung.

StyledCellLabelProvider

Zur Auszeichnung der Texte mit Farben oder Schriften kann ein *StyledCellLabelProvider* verwendet werden. Hierbei muss die Methode *update* überschrieben werden. Diese Methode bekommt eine *ViewerCell* übergeben. Dieser kann man mit *setText()* den Text setzen und *setStyleRanges(StyleRanges)* die den Text formatieren. Ein Beispiel dazu findet sich im "Eclipse JFace Table - Advanced Tutorial" in den weiteren Informationen.

Prozentuale Spaltenbreiten

Mit *TableColumnLayout* kann die Breite von Tabellenspalten prozentual festgelegt werden. Dazu wird ein *Composite* um die Tabelle herum benötigt, welches nur das *Table*-Widget enthält. Diesem wird ein *TableColumnLayout* gesetzt. Das Layout übernimmt dann die Verteilung der Spaltenbreiten in der Tabelle:

```
Composite tableComposite = new Composite(parent, SWT.NONE);
TableColumnLayout tableColumnLayout = new TableColumnLayout();
tableComposite.setLayout(tableColumnLayout);
TableViewer tableViewer = new TableViewer(tableComposite, SWT.BORDER | SWT.FULL_SELECTION);
```

Auf dem Layout-Objekt kann für jede Spalte ein *ColumnLayoutData*-Objekt gesetzt werden, um dem Layout den Platzbedarf der Tabellenspalte mitzuteilen. *ColumnPixelData* erlaubt die absolute Angabe einer Spaltenbreite in Pixeln:

```
tableColumnLayout.setColumnData(viewerNameColumn.getColumn(), new ColumnPixelData(50));
```

ColumnWeightData erlaubt die Angabe einer prozentualen Spaltengewichtung, minimaler Breite in Pixel sowie die Angabe, ob die Spaltenbreite vom Benutzer verändert werden darf:

```
tableColumnLayout.setColumnData(viewerNameColumn.getColumn(), new ColumnWeightData(20, 200, true));
```

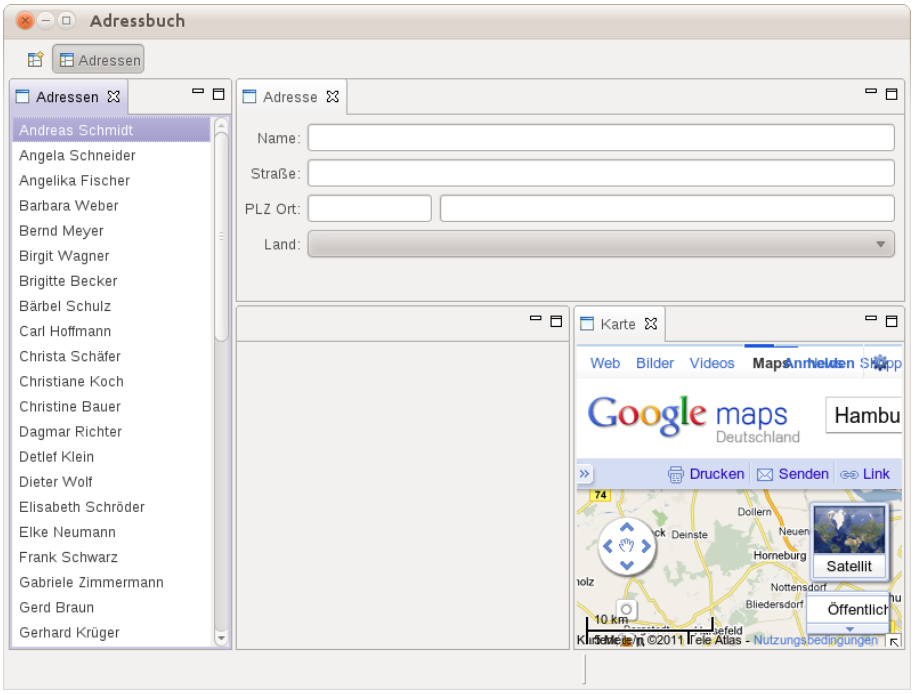
Weitere Informationen

- JFace Snippets
  http://wiki.eclipse.org/index.php/JFaceSnippets
- Eclipse JFace Table - Tutorial
  http://www.vogella.de/articles/EclipseJFaceTable/article.html
- Eclipse JFace Table - Advanced Tutorial
  http://www.vogella.de/articles/EclipseJFaceTableAdvanced/article.html
- Building tables made easy: TableViewerBuilder
  http://www.ralfebert.de/blog/eclipsercp/tableviewerbuilder/
- Java ist auch eine Insel: Vergleichen von Objekten mit Comparator und Comparable
  http://openbook.galileodesign.de/javainsel5/javainsel11_008.htm#Rxx747java11008040003941F04D114

TUTORIAL 13.1

Adresstabelle

☐ Ersetzen Sie das Label *"... Adressen geladen"* im Adressen-View durch einen *JFace TableViewer* und konfigurieren Sie ihn mit einem geeigneten *Content-* und *LabelProvider*.



☐ Ändern Sie den *UIJob* so ab, dass hier das Input-Objekt des Viewers statt des Labels gesetzt wird.

☐ Entfernen Sie den “Lade Adressen”-Button und rufen Sie die *refresh*-Methode direkt in *createPartControl* auf.

☐ Entfernen Sie die nun hinfälligen Test-Methoden *testAddressListButton* und *testLoadAddressesJob* und führen Sie alle Tests bis *testAddressTable* aus.

Häufige Probleme:

Tabellenspalten nicht sichtbar: Wurde den Tabellenspalten eine Breite gesetzt?

Tabelleninhalte nicht sichtbar: Prüfen Sie, ob Sie den TableViewer mit einem existierenden SWT Table-Widget konstruieren. Wenn ja, darf hier kein Style-Flag angegeben werden, da dies eine neue Tabelle konstruiert (siehe Methodensignatur der jeweiligen Konstruktoren).

KAPITEL 14

Workbench APIs

Die Workbench stellt ein umfangreiches API zur Verfügung, um Aktionen in der Workbench auszulösen. Die meisten Methoden finden Sie in der *WorkbenchPage*, die die Inhalte des *WorkbenchWindow* verwaltet. Einige Aufrufe erfolgen auch global auf dem *Workbench*-Objekt.

In einem View gelangen Sie über die *ViewSite* an diese Objekte. Die *ViewSite* ist die Schnittstelle zwischen dem View und der Workbench:

```
IWorkbenchWindow workbenchWindow = getSite().getWorkbenchWindow();
IWorkbench workbench = workbenchWindow.getWorkbench();
IWorkbenchPage workbenchPage = workbenchWindow.getActivePage();
```

So können Sie z.B. programmatisch ein View öffnen oder schließen:

```
workbenchPage.showView(SomePlugin.VIEW_ID_EXAMPLE);

IViewReference viewReference = workbenchPage.findViewReference(SomePlugin.VIEW_ID_EXAMPLE);
workbenchPage.hideView(viewReference);
```

Views entkoppeln über den SelectionService

Ein weiteres API der Workbench ist der *SelectionService*. Jedes Workbench-Fenster hat eine globale Selektion, die vom *SelectionService* verwaltet wird. Die Selektion folgt dem gerade aktiven View, d.h. sind mehrere Views mit einer Selektion geöffnet, gilt die Selektion des fokussierten Views.

Um an diesem Mechanismus teilzunehmen, muss ein View einen *SelectionProvider* bei der Workbench registrieren. Jeder JFace Viewer implementiert das Interface *SelectionProvider*, kann also direkt verwendet werden. Die Registrierung erfolgt über die *ViewSite*:

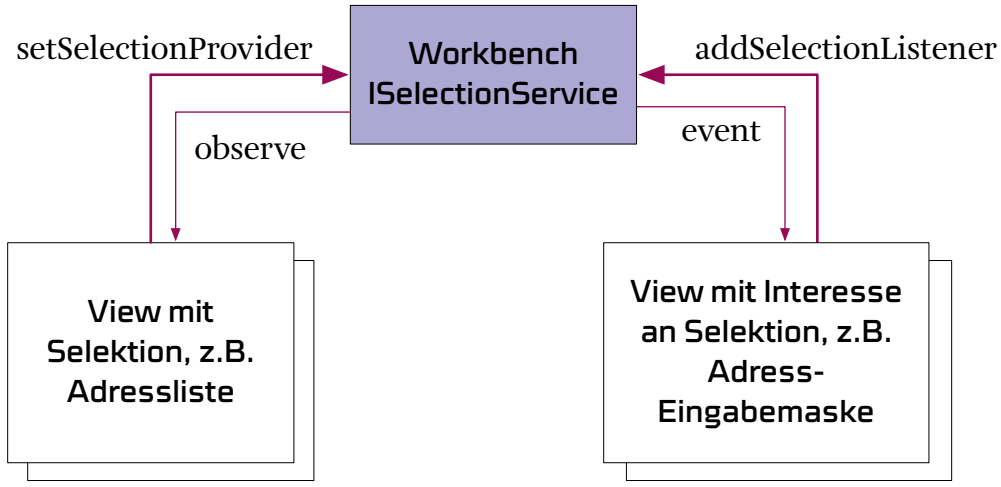
```
getSite().setSelectionProvider(tableViewer);
```

Andere Views können über den *SelectionService* nun die Selektion beobachten, ohne das View direkt zu kennen. Hierbei ist darauf zu achten, dass so alle Selektionen von allen Views beobachtet werden, z.B. indem nur auf interessante Ereignisse reagiert wird:

```
ISelectionService selectionService = getSite().getWorkbenchWindow()
        .getSelectionService();

selectionService.addSelectionListener(new ISelectionListener() {
```

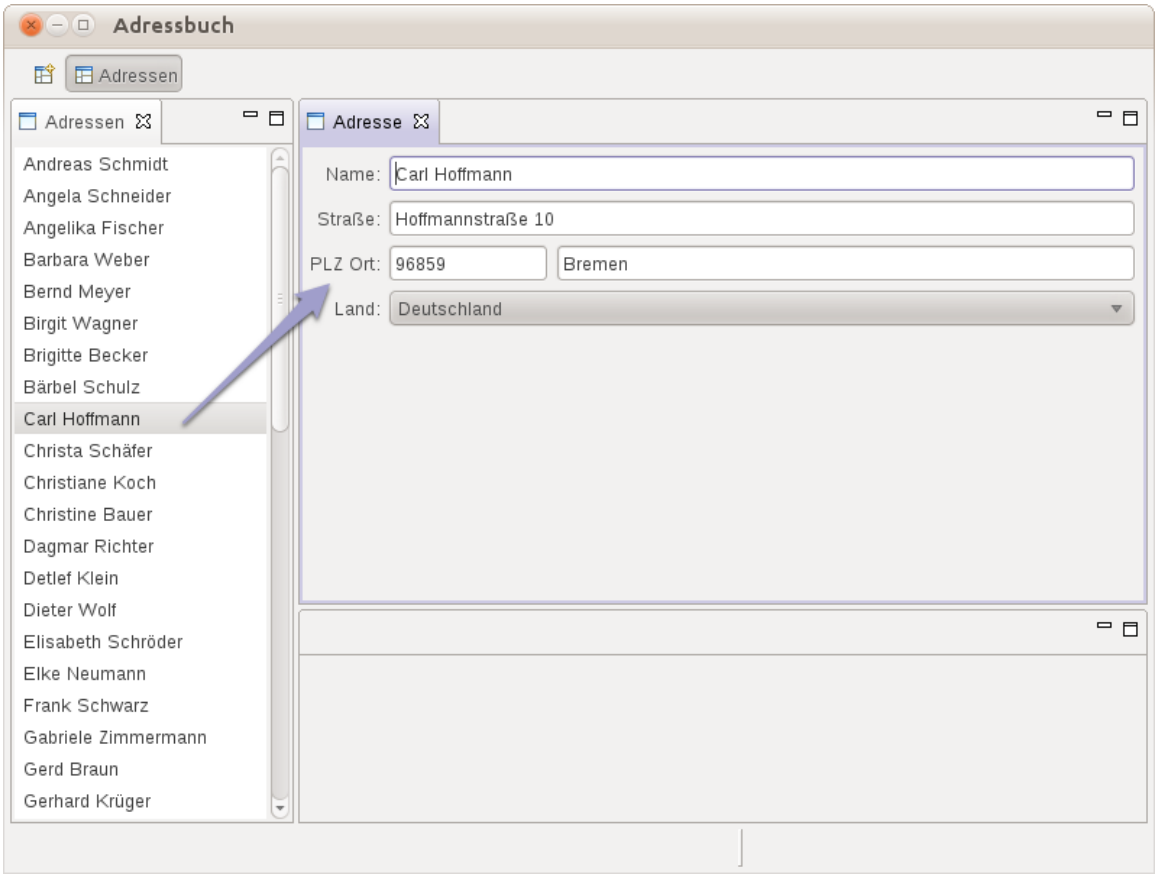
```
    @Override
    public void selectionChanged(IWorkbenchPart part,
            ISelection selection) {
        if (!(selection instanceof IStructuredSelection))
            return;
        Iterator iterator = ((IStructuredSelection) selection).iterator();
        while (iterator.hasNext()) {
            Object object = iterator.next();
            if (object instanceof Address) {
                // handle Address selection event
            }
        }

    }
});
```

So können Views voneinander entkoppelt werden, sie müssen sich nicht direkt kennen, sondern interagieren unabhängig voneinander über den *SelectionService* der Workbench. Dies hat auch den Vorteil, dass dieser Mechanismus erweiterbar ist, d.h. es können später weitere Views hinzukommen, die ebenfalls eine solche Selektionen bereitstellen oder auch auf solche Selektionen reagieren können:



TUTORIAL 14.1

Adressliste und Adressanzeige koppeln



☐ Registrieren Sie den *TableViewer* im Adresslisten-View als *SelectionProvider* bei der Workbench:

```
getSite().setSelectionProvider(tableViewer);
```

☐ Erstellen Sie in dem View mit der Adressmaske eine neue Methode *setAddress*, die alle Eingabefelder mit den Daten einer Adresse befüllt. Die Länderauswahl können Sie dabei setzen, indem Sie dem ComboViewer eine Selektion setzen. Zum Beispiel:

```
private void setAddress(Address address) {
    txtName.setText(address.getName());
    // ...
    countryComboViewer.setSelection(new StructuredSelection(address.getCountry()))
}
```

- ☐ Registrieren Sie in dem View mit der Adressmaske einen SelectionListener bei dem Workbench-*SelectionService*. Implementieren Sie diesen so, dass *setAddress* aufgerufen wird, sofern die Selektion eine Adresse enthält.

- ☐ Überprüfen Sie, dass in **allen** View-Klassen die Methode *setFocus* so implementiert ist, das einem Steuerelement im View der Focus zugewiesen wird, da die Selektion sonst bei der späteren Einführung von Kontextmenüs nicht korrekt behandelt wird.

- ☐ Führen Sie alle Tests inkl. *testSelectionMasterDetail* aus.

KAPITEL 15

Commands

Seit Eclipse 3.3 besteht mit dem Command-Framework ein einheitlicher, auf Extension-Points basierender Mechanismus, mit dem Plug-ins Kommandos deklarieren und diese in Menüs oder Toolbars der Anwendung einfügen können.

Dabei wird durch die deklarative Herangehensweise Flexibilität und Erweiterbarkeit erreicht. So können Plug-ins beispielsweise bestehende Menüstrukturen erweitern oder zu einem vorhandenen Command ein neues Verhalten in einem bestimmten Kontext hinzufügen.

Ein weiterer Vorteil der Commands besteht darin, dass Menüs und Toolbars von der Workbench angezeigt werden können, ohne Klassen aus dem jeweiligen Plug-in zu laden, da alle zur Anzeige benötigten Informationen im *plugin.xml* enthalten sind. Dies macht sich insbesondere bei großen Anwendungen mit sehr vielen Commands hinsichtlich der Startzeit bemerkbar.

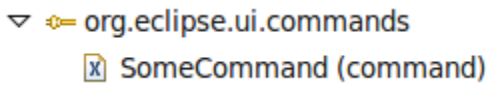
Die Commands lösen das alte *Action*-Konzept der Eclipse Workbench ab. Sie werden in vielen Dokumenten und Webseiten noch Verweise auf dieses Konzept finden. Da die Commands seit Eclipse 3.4 ein vollwertiger Ersatz für die alten Konzepte, APIs und Extension Points sind und die Actions weithin als veraltetet betrachtet werden, werden hier nur noch die neuen Commands behandelt.

Commands

Ein Command definiert eine ausführbare Aktion wie “Speichern”, “Schließen” oder “Aktualisieren”, nicht jedoch das Verhalten dieser Aktion. Gängige Standard-Commands werden von dem Plug-in *org.eclipse.ui* bereitgestellt, z.B.:

- *org.eclipse.ui.file... save, saveAll, close, closeAll, refresh, exit*
- *org.eclipse.ui.edit... undo, redo, cut, copy, paste, delete*

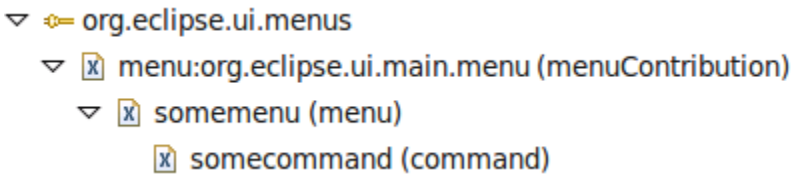
Eigene Commands können mit einer Extension zu *org.eclipse.ui.commands* definiert werden:




```
<extension point="org.eclipse.ui.commands">
    <command id="com.example.somecommand" name="SomeCommand"/>
</extension>
```

Commands in Menüs einfügen

Über den Extension Point *org.eclipse.ui.menus* können Menüstrukturen unter der Verwendung zuvor erstellter Commands definiert werden:



Über eine *menuContribution* wird einer bestehenden Menüstruktur etwas hinzugefügt. Das Attribut *locationURI* gibt dabei die Stelle an, an der die neuen Einträge eingefügt werden sollen. Die entsprechenden Orte sind in der Klasse *org.eclipse.ui.menus.MenuUtil* spezifiziert:

- *menu:<menu-id>* Menü der Anwendung
- *toolbar:<toolbar-id>* Tool-Bar der Anwendung
- *popup:<popupmenu-id>* Popup-Menüs der Anwendung
- *menu:org.eclipse.ui.main.menu* Oberste Ebene des Hauptmenüs
- *toolbar:org.eclipse.ui.main.toolbar* Oberste Ebene der Toolbar
- *popup:org.eclipse.ui.popup.any* Alle Popup-Menüs der Anwendung

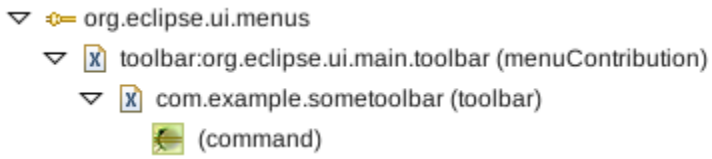
Optional können Sie mit *?after=menuId* oder *?before=menuId* auf die Reihenfolge der Menüpunkte Einfluss nehmen. In eine solche *menuContribution* können Sie direkt Untermenüs oder Commands einfügen:


```
<extension point="org.eclipse.ui.menus">
    <menuContribution locationURI="menu:org.eclipse.ui.main.menu">
        <menu id="com.example.somemenu" label="SomeMenu">
            <command commandId="com.example.somecommand"/>
        </menu>
    </menuContribution>
</extension>
```

Commands in die Toolbar einfügen

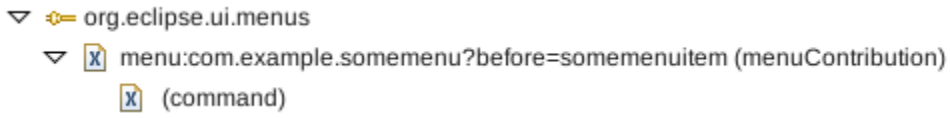
Die Verwendung von Commands in der Toolbar erfolgt analog zu Menüs über den Extension Point *org.eclipse.ui.menus*. Als *locationURI* gibt man jedoch *toolbar:org.eclipse.ui.main.toolbar* an, statt *menu* wird ein *toolbar*-Element verwendet:

```
<extension point="org.eclipse.ui.menus">
    <menuContribution locationURI="toolbar:org.eclipse.ui.main.toolbar">
        <toolbar id="com.example.sometoolbar">
            <command commandId="com.example.somecommand"/>
        </toolbar>
    </menuContribution>
</extension>
```

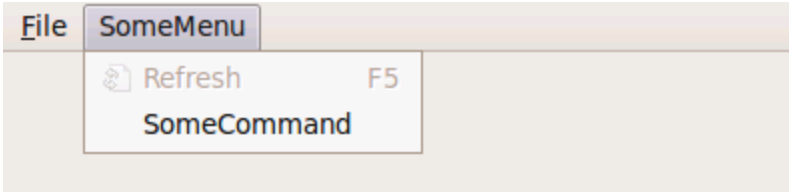


Commands in selbstdefinierte Menüs einfügen

Auch selbstdefinierten Menüs können nachträglich mit einer *menuContribution* Commands oder Untermenüs hinzugefügt werden. Dazu wird die *locationURI menu:[menuid]* verwendet. Möchten wir im zuvor erzeugten Menü *com.example.somemenu* noch einen weiteren Eintrag ergänzen, könnten wir dies auch über eine separate Contribution erreichen. Diesen Weg beschreitet man vor allem, wenn man ein bestehendes Menü aus einem anderen Plug-in um weitere Menüeinträge ergänzen möchte:



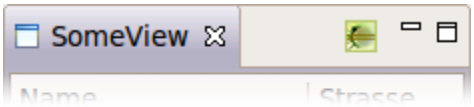

```
<extension point="org.eclipse.ui.menus">
    <menuContribution locationURI="menu:com.example.somemenu?before=somemenuitem">
        <command commandId="org.eclipse.ui.file.refresh"/>
    </menuContribution>
</extension>
```



Commands zu Views hinzufügen

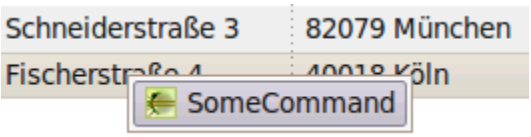
Über die *locationURI toolbar:<viewId>* können Sie einen Command zu einem View hinzufügen:

```
<menuContribution locationURI="toolbar:com.example.someview">
   <command
         commandId="com.example.somecommand"
         icon="icons/alt_window_16.gif"
         style="push">
   </command>
</menuContribution>
```



Commands in Kontextmenüs einfügen

Um einen Command in ein Kontextmenü hinzuzufügen, sollten Sie das Menü mit einem speziellen Separator an der Stelle, an der später hinzugefügte Commands erscheinen sollen, versehen:




```
MenuManager menuManager = new MenuManager();
table.setMenu(menuManager.createContextMenu(table));
```

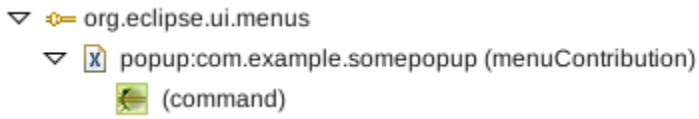
Zudem muss das Kontextmenü bei der Workbench registriert werden. Dies geschieht über die *Site* des *View*- oder *EditorParts*. Die Angabe des ersten Parameters ID ist optional. Wird sie weggelassen, wird das Menü mit der ID des Parts registriert:

```
getSite().registerContextMenu("com.example.somepopup", menuManager, tableViewer);
```

Handelt es sich bei dem Control um einen JFace Viewer, sollte dieser der *Site* als *SelectionProvider* gesetzt werden, damit die Commands im Popup-Menü auf die aktuelle Selektion reagieren können:

```
getSite().setSelectionProvider(tableViewer);
```

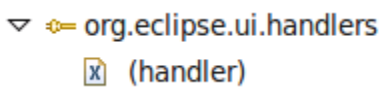
Nun können Sie dem Popup-Menü über die URI *popup:<menuId>* Commands hinzufügen:




```
<extension point="org.eclipse.ui.menus">
    <menuContribution locationURI="popup:com.example.somepopup">
        <command
            commandId="com.example.somecommand"
            icon="icons/alt_window_16.gif"
            style="push">
        </command>
    </menuContribution>
</extension>
```

Handler

Das Verhalten von Commands wird separat durch Handler definiert. Über den Extension Point *org.eclipse.ui.handlers* kann zu einem Command eine Handler-Klasse registriert werden, die für die Ausführung des Commands zuständig ist:



```
<extension point="org.eclipse.ui.handlers">
    <handler
        commandId="com.example.somecommand"
        class="com.example.example.SomeCommandHandler">
    </handler>
</extension>
```

Handler sollten von *org.eclipse.core.commands.AbstractHandler* erben und die *execute*-Methode implementieren. Über statische Methoden auf der *HandlerUtil*-Klasse können Sie in einer Handler-Implementierung auf die Workbench zugreifen:

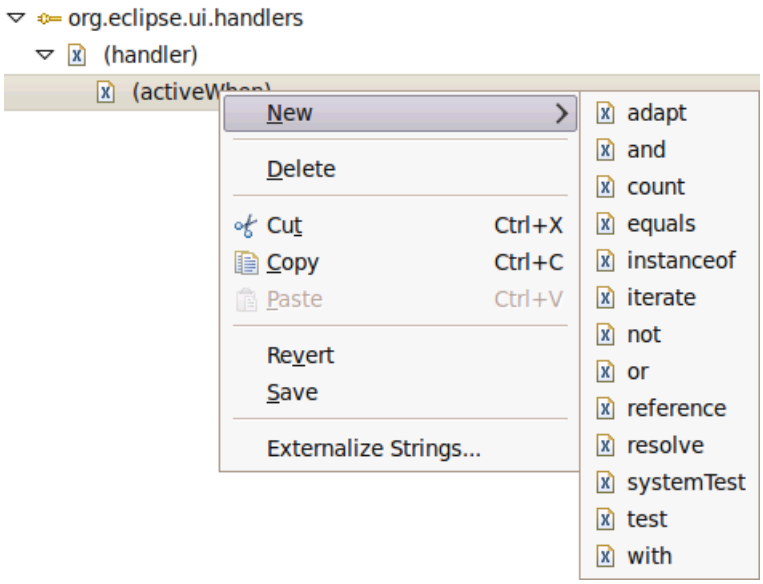
```
public class SomeCommandHandler extends AbstractHandler {

    public Object execute(ExecutionEvent event) throws ExecutionException {
        // Handle command here, Workbench can be accessed using HandlerUtil
        // Return value is reserved for future changes
        return null;
    }

}
```

Häufig benötigte *HandlerUtil*-Methoden sind:

- *getActiveShell*
- *getActiveWorkbenchWindow*
- *getActiveEditor, getActiveEditorId*
- *getActivePart, getActivePartId*
- *getActiveSite*
- *getCurrentSelection*

Handler bedingt aktivieren

Mit einem *activeWhen*-Element in der Handler-Contribution können Bedingungen festgelegt werden, unter denen ein Handler aktiv sein soll:



Ist kein Handler für einen Command aktiv, wird der Command deaktiviert, d.h. der Menüeintrag oder Toolbar-Button erscheint ausgegraut. Sind mehrere Handler für einen Command aktiv, greift der spezifischere Handler.

Für diese Ausdrücke stehen eine Reihe von Variablen zur Verfügung, mit denen Sie auf die Workbench-Umgebung reagieren können:

- *selection*
- *activePartId*
- *activeEditorId*
- *activeWorkbenchWindow.activePerspective*
- *activeFocusControl*, *activeFocusControlId*

Eine Übersicht der Variablen finden Sie unter Variables and the Command Framework sowie in der Klasse *ISources*.

Um die Funktionsweise zu verdeutlichen im Folgenden einige Beispiele für typische Ausdrücke zur Aktivierung von Command-Handlern:

> Handler abhängig vom aktivem WorkbenchPart aktivieren:

```
<activeWhen>
    <with variable="activePartId">
        <equals value="com.example.view"/>
```

> Handler abhängig von der Selektion aktivieren:

```
<activeWhen>
    <with variable="selection">
        <iterate ifEmpty="false" operator="or">
            <instanceof value="com.example.SomeObject"/>
```

> Commands abhängig von der geöffneten Perspektive einblenden:

```
<activeWhen>
    <with variable="activeWorkbenchWindow.activePerspective">
        <equals value="com.example.perspective"/>
```

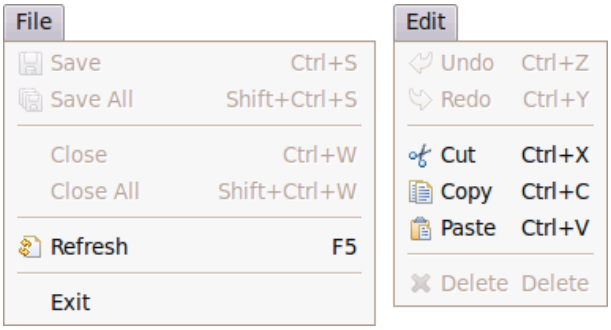
Weitere Informationen

- Eclipse Wiki: Platform Command Framework
  http://wiki.eclipse.org/Platform_Command_Framework
- Eclipse Wiki: Menu Contributions
  http://wiki.eclipse.org/Menu_Contributions
- Eclipse Commands with Eclipse 3.5, Tutorial von Lars Vogel
  http://www.vogella.de/articles/EclipseCommands/article.html
- Eclipse Tips: Commands, Blog-Post von Prakash G.R.
  http://blog.eclipse-tips.com/2009/01/commands-part-1-actions-vs-commands.html
- Working with the menus extension point, Blog-Post von Tonny Madsen
  http://blog.rcp-company.com/2007/06/working-with-menus-extension-point.html
- Eclipse Wiki: Menus Extension Mapping (action vs. command extension points)
  http://wiki.eclipse.org/Menus_Extension_Mapping
- Eclipse Wiki: Command Core Expressions
  http://wiki.eclipse.org/Command_Core_Expressions
- Making extensible GUI with org.eclipse.ui.menus
  http://jacekonthings.blogspot.com/2011/07/making-extensible-gui-with-help-of.html
- Rezepte: Commands programmatisch ausführen (Seite 189)
- Rezepte: Expressions erweitern mit Property-Testern (Seite 190)
- Rezepte: Expressions um eigene Variablen erweitern mit Source Providern (Seite 192)
- Rezepte: Dynamisches aktualisieren mit IElementUpdater (Seite 194)

TUTORIAL 15.1

Menü und Toolbar befüllen

- ☐ Erstellen Sie im Adressbuch-Plug-in eine Extension zu *org.eclipse.ui.menus*.
- ☐ Definieren Sie eine *menuContribution* zu der *locationUri menu:org.eclipse.ui.main.menu*.
- ☐ Erstellen Sie ein Menü “Datei” und binden Sie die Standard-Commands *org.eclipse.ui.file.save*, *saveAll*, *close*, *closeAll*, *refresh*, *exit* ein. Erstellen Sie ein Menü “Bearbeiten” und binden Sie die Standard-Commands *org.eclipse.ui.edit.undo*, *redo*, *cut*, *copy*, *paste*, *delete* ein:



- ☐ Aktivieren Sie die *CoolBar* in *ApplicationWorkbenchWindowAdvisor#preWindowOpen* mit *configurer.setShowCoolBar(true)*.
- ☐ Fügen Sie der Anwendung eine zweite *menuContribution* zu *toolbar:org.eclipse.ui.main.toolbar* hinzu, um der Anwendung neue Toolbars hinzuzufügen.
- ☐ Legen Sie eine Toolbar mit den Commands *Save* und *Refresh* darin an:

- ☐ Führen Sie alle Tests inkl. *testToolbar* aus.

TUTORIAL 15.2

Adressliste aktualisieren

- ☐ Erstellen Sie eine Extension zu *org.eclipse.ui.handlers*. Fügen Sie einen neuen Handler für den *Refresh*-Command hinzu. Achten Sie beim Anlegen der Handler-Klasse darauf, *org.eclipse.core.commands.AbstractHandler* als Superklasse anzugeben.

- ☐ Implementieren Sie den Handler so, dass die Adresslisten-View neu geladen wird, sofern diese gerade aktiv ist. Mit *HandlerUtil#getActivePart* gelangen Sie an das aktive View. Verwenden Sie die zuvor definierte View-Methode *refresh*, um die Adressliste neu vom Adressbuch-Service zu laden:

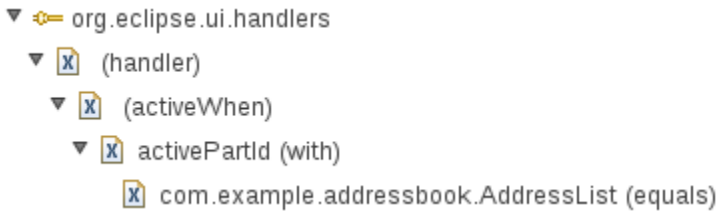
```
public class RefreshHandler extends AbstractHandler {

    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        IWorkbenchPart part = HandlerUtil.getActivePart(event);
        if (part instanceof AddressList) {
            ((AddressList) part).refresh();
        }

        return null;
    }

}
```

- ☐ Führen Sie alle Tests inkl. *testAddressListRefresh* aus.

TUTORIAL 15.3

Refresh-Command kontextabhängig aktivieren

- ☐ Fügen Sie dem Handler für den *Refresh*-Command ein *activeWhen*-Element hinzu, so dass der Handler nur dann greift, wenn die Adresslisten-View aktiv ist:



- ☐ Führen Sie alle Tests inkl. *testRefreshEnabledOnlyForAddressListView* aus.

TUTORIAL 15.4

Karten-Plug-in um Commands erweitern

- ☐ Vergeben Sie im Adressbuch-Plug-in die ID *file* für das Datei-Menü, damit andere Plug-ins Erweiterungen in dieses Menü einfügen können.
- ☐ Erstellen Sie im dem Karten-Plug-in eine Extension zu *org.eclipse.ui.commands* und fügen einen Command *Show Map* mit der ID *com.example.addressbook.maps.show* hinzu.
- ☐ Erstellen Sie im Karten-Plug-in eine Extension zu *org.eclipse.ui.menus* und fügen Sie den soeben definierten Command mit einer *menuContribution* in das bestehende Datei-Menü (*locationURI menu:file*) ein.
- ☐ Erstellen Sie im Karten-Plug-in einen neuen Handler für den *Show Map*-Command. Implementieren Sie diesen so, dass das Karten-View geöffnet wird:

```
public class ShowMapHandler extends AbstractHandler {

    public Object execute(ExecutionEvent event) throws ExecutionException {
        try {
            IWorkbenchPage page = HandlerUtil.getActiveWorkbenchWindow(event)
                    .getActivePage();
            page.showView(MapsIds.VIEW_ID_MAP);
        } catch (PartInitException e) {
            throw new ExecutionException(e.getMessage(), e);
        }
        return null;
    }

}
```

- ☐ Setzen Sie in der *perspectiveExtension* im Karten-Plug-in das Attribut *visible* für das View auf *false*, so dass die Karte initial nicht angezeigt wird.

TUTORIAL 15.5

Adresse löschen im Kontextmenü

- ☐ Erlauben Sie Mehrfachselektionen in der Adressliste, indem Sie das Style-Bit *SWT.MULTI* beim Erzeugen von *Table* bzw. *TableViewer* angeben.

- ☐ Erstellen und registrieren Sie ein Kontextmenü in der Implementierung des Adresslisten-Views:

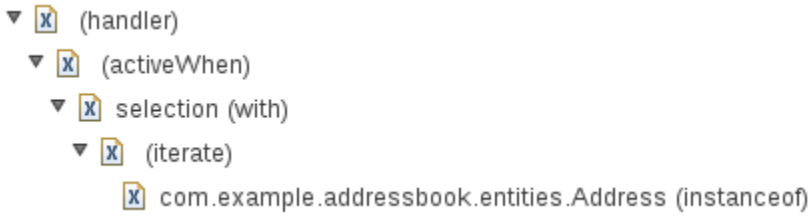
```
MenuManager menuManager = new MenuManager();
tableViewer.getTable().setMenu(menuManager.createContextMenu(tableViewer.getTable()));
getSite().registerContextMenu(menuManager, tableViewer);
getSite().setSelectionProvider(tableViewer);
```

- ☐ Fügen Sie den *Delete*-Command mit einer *menuContribution* zu der *locationURI* *popup:com.example.addressbook.AddressList* in das Kontextmenü ein.

- ☐ Definieren Sie einen neuen Handler für den *Delete*-Command, der die ausgewählten Adressen löscht. Mit *HandlerUtil.getCurrentSelection(event)* gelangen Sie in der Handler-Implementierung an die aktuelle Selektion. Der *AddressService* erlaubt das Löschen von Adressen:

```
public Object execute(ExecutionEvent event) throws ExecutionException {
    ISelection selection = HandlerUtil.getCurrentSelection(event);
    if (!(selection instanceof IStructuredSelection))
        return null;

    for (Iterator<?> it = ((IStructuredSelection)selection).iterator(); it.hasNext();) {
        Object obj = it.next();
        if (obj instanceof Address) {
            AddressBookServices.getAddressService().deleteAddress(((Address) obj).getId());
        }
    }

    return null;
}
```

- ☐ Mit einem *activeWhen*-Ausdruck können Sie den Löschen-Handler nur dann aktivieren, wenn mindestens eine Adresse ausgewählt ist:



- ☐ Beobachten Sie im *AddressList*-View mit einem *AddressChangeListener* den *IAddressService* und aktualisieren Sie das View automatisch, wenn sich Adressen ändern:

```
private IAddressChangeListener addressChangeListener = new IAddressChangeListener() {

    @Override
    public void addressesChanged() {
        refresh();
    }

};

public void createPartControl(Composite parent) {
    // ...
    AddressBookServices.getAddressService()
        .addAddressChangeListener(addressChangeListener);
}

public void dispose() {
    AddressBookServices.getAddressService()
        .removeAddressChangeListener(addressChangeListener);
    super.dispose();
}
```

- ☐ Führen Sie alle Tests inkl. *testDeleteAddress* aus.

KAPITEL 16

Dialoge und Wizards

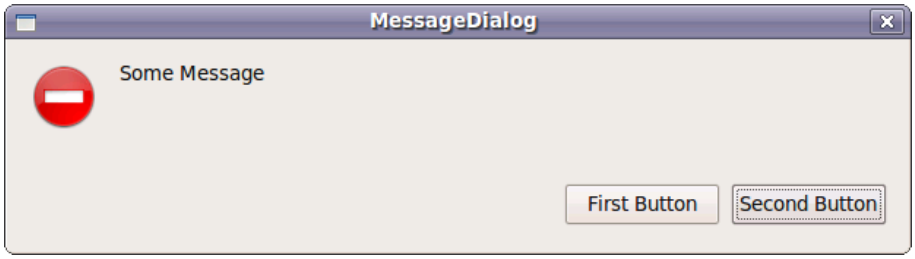
SWT Systemdialoge

SWT stellt für alle Systemdialoge entsprechende Klassen bereit:

- *MessageBox*
- *FileDialog*
- *DirectoryDialog*
- *ColorDialog*
- *FontDialog*
- *PrintDialog*

JFace Nachrichtendialoge

Ein JFace *MessageDialog* zeigt dem Benutzer ähnlich zur SWT *MessageBox* eine Nachricht an. Es wird jedoch kein Systemdialog verwendet. So können z.B. eigene Buttons hinzugefügt werden:




```
MessageDialog messageDialog = new MessageDialog(parentShell, "MessageDialog", null,
        "Some Message", MessageDialog.ERROR,
        new String[] { "First Button", "Second Button" }, 1);
if (messageDialog.open() == 1) {
    System.out.println("Second Button was clicked.");
}
```

Für Standard-Dialoge stehen auf *MessageDialog* statische Methodenaufrufe zur Verfügung:

```
MessageDialog.openConfirm(parentShell, title, message)
MessageDialog.openError(parentShell, title, message)
MessageDialog.openInformation(parentShell, title, message)
MessageDialog.openQuestion(parentShell, title, message)
MessageDialog.openWarning(parentShell, title, message)
```

In Eclipse werden häufig *Status*-Objekte eingesetzt, um Fehlerzustände zu kommunizieren. Möchten Sie dem Benutzer einen solchen Status anzeigen, können Sie die Klasse *ErrorDialog* verwenden:

```
Status status = new Status(IStatus.ERROR, Activator.PLUGIN_ID, "Status message");
ErrorDialog.openError(parentShell, "ErrorDialog", "Message", status);
```

JFace Dialoge

Neben Standarddialogen finden Sie in JFace auch Basisklassen für eigene Dialoge. Die einfachste Basisklasse ist *Dialog*, ein modaler Dialog mit einer Buttonzeile unterhalb des Inhaltsbereiches:

Um einen solchen Dialog zu implementieren, leiten Sie eine Klasse von *Dialog* ab und implementieren die Methode *createDialogArea*, um den Dialog mit Inhalten zu befüllen. Wichtig zu beachten ist dabei: Der Dialog selbst wird mit einem *GridLayout* gelayoutet und daher wird erwartet, dass dem zurückgegebene Control ein *GridData*-Objekt zugewiesen ist.

Dazu kann man *super.createDialogArea* aufrufen und das zurückgegebene Composite mit eigenen Inhalten befüllen:

```
Dialog dialog = new Dialog(parentShell) {
    @Override
    protected Control createDialogArea(Composite parent) {
        Composite composite = (Composite) super.createDialogArea(parent);
        Label label = new Label(composite, SWT.NONE);
        label.setText("Dialog Area");
        return composite;
    }

    @Override
    protected Point getInitialSize() {
        return new Point(400, 300);
    }

    @Override
    protected void configureShell(Shell newShell) {
        super.configureShell(newShell);
        newShell.setText("Dialog");
    }
};

if (dialog.open() == Dialog.OK)
    System.out.println("User clicked ok.");
```

JFace TitleAreaDialog

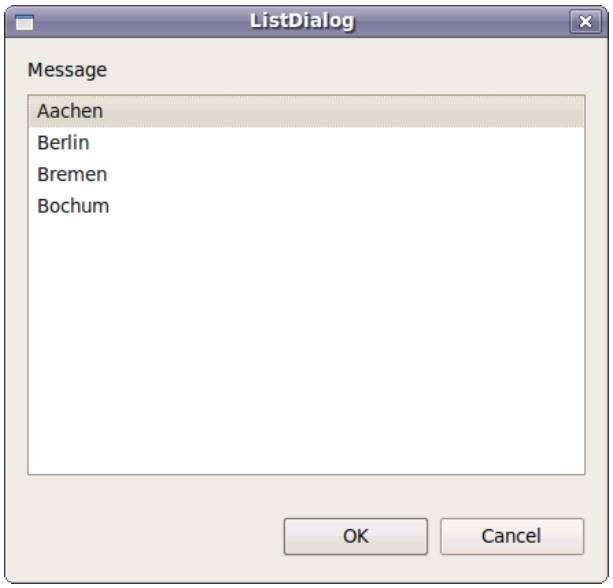
Die Klasse *TitleAreaDialog* stellt einen Dialog mit Titelbereich mit Text und Symbol bereit:

Die Implementierung eines solchen Dialogs erfolgt analog zur *Dialog*-Klasse:

```
TitleAreaDialog titleAreaDialog = new TitleAreaDialog(parentShell) {

    @Override
    protected Control createDialogArea(Composite parent) {
        LocalResourceManager resources
                = new LocalResourceManager(JFaceResources.getResources(), getShell());

        setTitle("Title");
        setErrorMessage("Error message");
        ImageDescriptor title = Activator.getImageDescriptor("/icons/imagetitle.png");
        setTitleImage(resources.createImage(title));

        Composite composite = (Composite) super.createDialogArea(parent);
        Label label = new Label(composite, SWT.NONE);
        label.setText("Dialog Area");
        return composite;
    }

};
```

JFace Auswahldialoge

JFace stellt einige Basisklassen bereit, die Sie verwenden können, um Auswahldialoge zu realisieren. Die einfachste Variante ist der *ListDialog*, der intern einen JFace *ListViewer* verwendet, um eine Auswahlliste zu präsentieren:



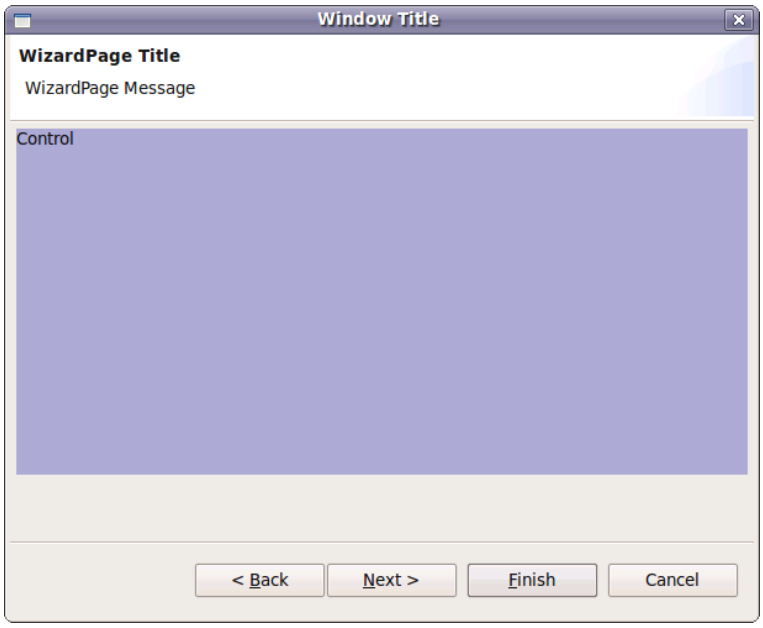
Die Inhalte werden dabei analog zu einem *ListViewer* gesetzt:

```
ListDialog listDialog = new ListDialog(parentShell);
listDialog.setTitle("ListDialog");
listDialog.setMessage("Message");
listDialog.setContentProvider(ArrayContentProvider.getInstance());
listDialog.setLabelProvider(new LabelProvider());
listDialog.setInput(someList);
if (listDialog.open() == Dialog.OK) {
    System.out.println("Selected objects: " + Arrays.toString(listDialog.getResult()));
}
```

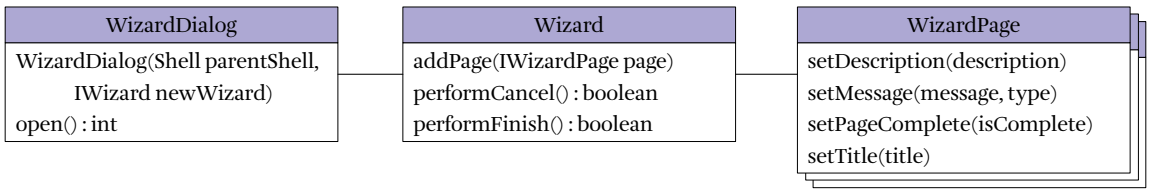
Analog dazu stehen mit *ElementTreeSelectionDialog* und *CheckedTreeSelectionDialog* Dialogklassen bereit, die eine Einfach- oder Mehrfachauswahl aus einem Baum ermöglichen.

JFace Wizards

JFace Wizards sind Basisklassen für Assistenten, die den Anwender durch eine Reihe von Schritten führen:



Zur Erstellung eines Wizards werden folgende Klassen benötigt:



Für jede Seite des Assistenten implementieren Sie eine *WizardPage*-Klasse:

```
public class SomeWizardPage extends WizardPage {

    protected SomeWizardPage() {
        super("Some wizard Page");
        setTitle("WizardPage Title");
        setMessage("WizardPage Message");
    }

    public void createControl(Composite parent) {

        // createControl verlangt es, genau ein Control zu erstellen
        // und dieses mit setControl zu setzen
```

```
        Label label = new Label(parent, SWT.NONE);
        setControl(label);

        // Hier würde composite mit eigenen UI Elementen befüllt

        // Mit setPageComplete(boolean) könnte man hier die Aktivierung
        // der nächsten Wizard-Seite in Abhängigkeit vom UI erlauben oder verbieten

    }

}
```

Für den Wizard selbst, also die Gesamtmenge aller Seiten, implementieren Sie die Klasse *Wizard*:

```
public class SomeWizard extends Wizard {

    public SomeWizard() {
        setWindowTitle("Window Title");
        addPage(new FirstWizardPage());
        addPage(new SecondWizardPage());
        // ...
    }

    @Override
    public boolean performFinish() {
        // true zurückgeben wenn Wizard erfolgreich abgeschlossen wurde
        return true;
    }

}
```

Einen solchen Wizard können Sie nun mit *WizardDialog* verwenden:

```
WizardDialog wizardDialog = new WizardDialog(parentShell, new SomeWizard());
wizardDialog.open();
```

Eclipse UI: Wizards für Neu, Im- und Export

Neben den JFace *Wizards* stellt Ihnen die Eclipse Workbench drei Extension Points bereit, mit denen Wizards für Neu, Im- und Export registriert werden können:

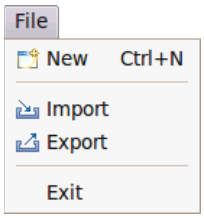
- *org.eclipse.ui.newWizards*
- *org.eclipse.ui.importWizards*

> *org.eclipse.ui.exportWizards*

Dabei ist eine *Wizard*-Klasse mit einem zusätzlichen Interface (*INewWizard*, *IImportWizard* bzw. *IExportWizard*) anzugeben, z.B.:

```
<extension point="org.eclipse.ui.newWizards">
   <wizard
         class="com.example.SomeNewWizard"
         id="com.example.SomeNewWizard"
         name="Neue Adresse">
   </wizard>
</extension>
```

Ein so registrierter *Wizard* wird dem Benutzer über entsprechende Workbench-Commands angeboten:



Command-Id
org.eclipse.ui.newWizard
org.eclipse.ui.file.import
org.eclipse.ui.file.export

Weitere Informationen

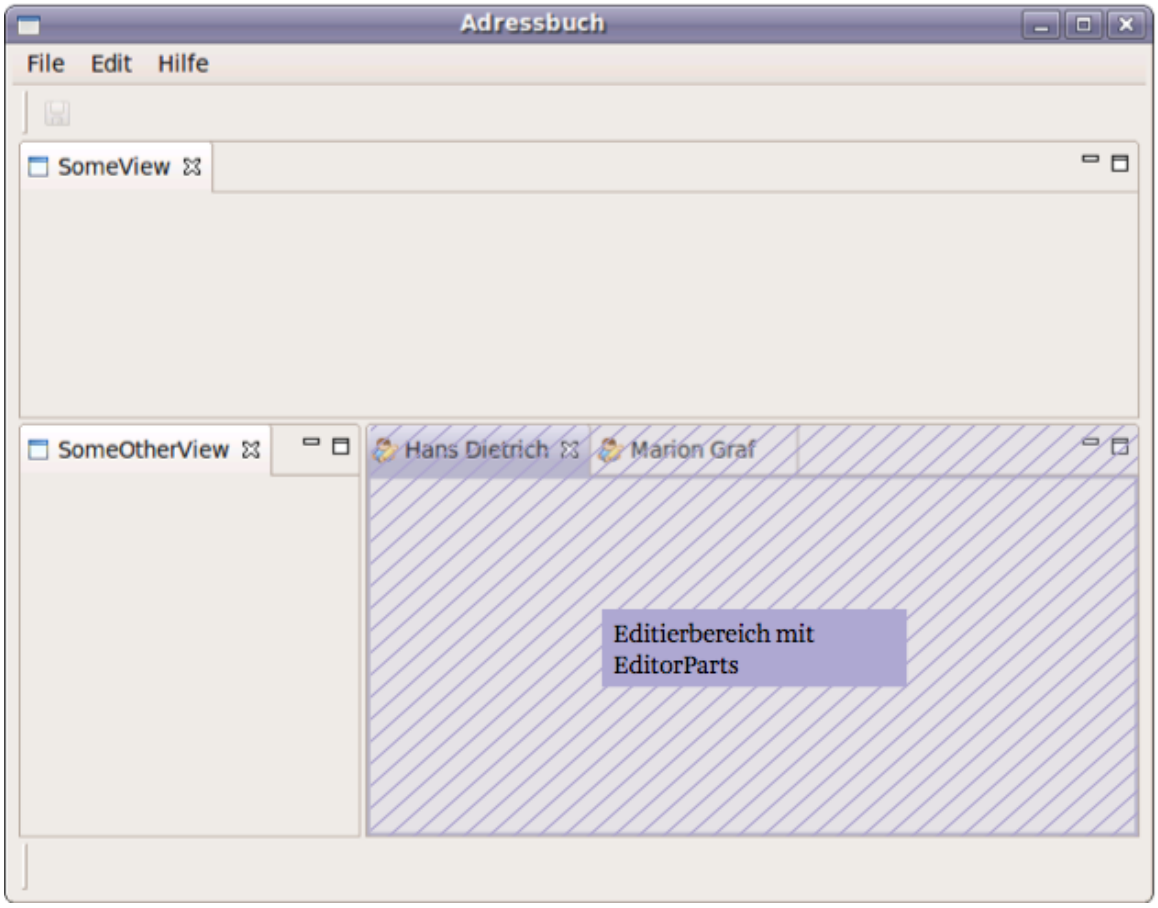
> JFace Wizard FAQ
http://www.ralfebert.de/blog/eclipsercp/wizard_faq/

KAPITEL 17

Editoren

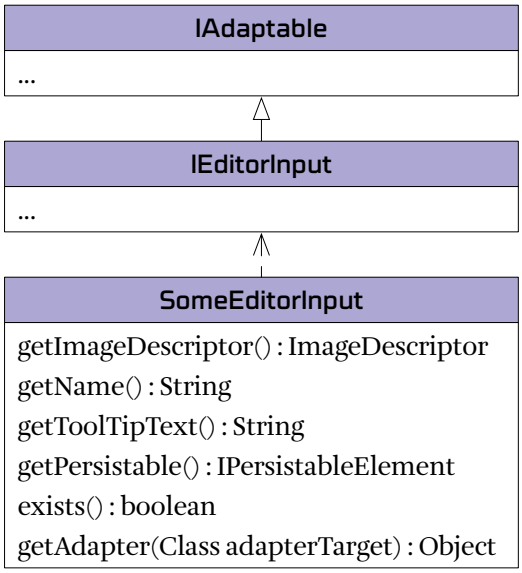
In Eclipse RCP-Anwendungen werden Dokumente, die vom Benutzer geöffnet, bearbeitet und wieder gespeichert werden können, mit Editoren abgebildet. Dies bedingt weder eine Datei im Dateisystem noch dass es sich dabei um einen Texteditor handeln muss - die Texteditoren der Eclipse IDE sind lediglich eine spezielle Variante eines Editors. Ein Editor könnte beispielsweise eine Eingabemaske oder eine Grafik zum Gegenstand haben. Die Daten könnten in einer Datenbank persistiert werden. Dies obliegt dem Entwickler bei der Implementierung des Editors.

Alle Editoren werden im Editierbereich der Anwendung geöffnet. Diesen Bereich teilen sich alle Perspektiven, er kann lediglich ein- und ausgeblendet werden:



EditorInput

Ein *EditorInput*-Objekt beschreibt eine Referenz auf die Daten, die der Editor bearbeitet. Bei einer Datei im Dateisystem würde das *EditorInput*-Objekt den Dateinamen beinhalten, bei einem Datenbankobjekt den Primärschlüssel. Für dieses Objekt ist das Interface *IEditorInput* zu implementieren:



Für einen Editor, dem ein Datenbank-Objekt *SomeObject* zugrunde liegt, könnte das Editor-Input-Objekt folgendermaßen implementiert werden:

```
/**
 * SomeEditorInput beschreibt eine Referenz auf SomeObject
 */
public class SomeEditorInput implements IEditorInput {

    // Minimale Informationen, mit der das zu bearbeitende Objekt beschrieben
    // und referenziert werden kann
    private int id;

    public SomeEditorInput(int id) {
        super();
        this.id = id;
    }

    public int getId() {
        return id;
    }
```

```
public ImageDescriptor getImageDescriptor() {
    return Activator.getImageDescriptor("/icons/someicon.png");
}

public String getName() {
    // Text für den Benutzer, wird initial im Reiter des Editors angezeigt
    return "Some Document " + id;
}

public String getToolTipText() {
    return getName();
}

public IPersistableElement getPersistable() {
    // getPersistable muss nur implementiert werden, wenn das Editor-Inputobjekt
    // über mehrere Anwendungssessions hinweg gelten soll, z.B. wenn eine
    // "Zuletzt geöffnete Dokumente"-Liste verwendet wird.
    return null;
}

public boolean exists() {
    // Ggf. prüfen ob Objekt noch existiert und false zurückgeben wenn nicht.
    // Vor allem relevant im Zusammenspiel mit getPersistable.
    return true;
}

public Object getAdapter(Class adapterTarget) {
    // Editor-Input-Objekte können optional adaptierbar gestaltet werden
    return null;
}

// equals und hashCode müssen implementiert werden, damit nur ein
// Editor für dasselbe Dokumente geöffnet werden kann

public int hashCode() {
    return id;
}

public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null || getClass() != obj.getClass())
```

```
            return false;
        return (id == ((SomeEditorInput) obj).getId());
    }

}
```

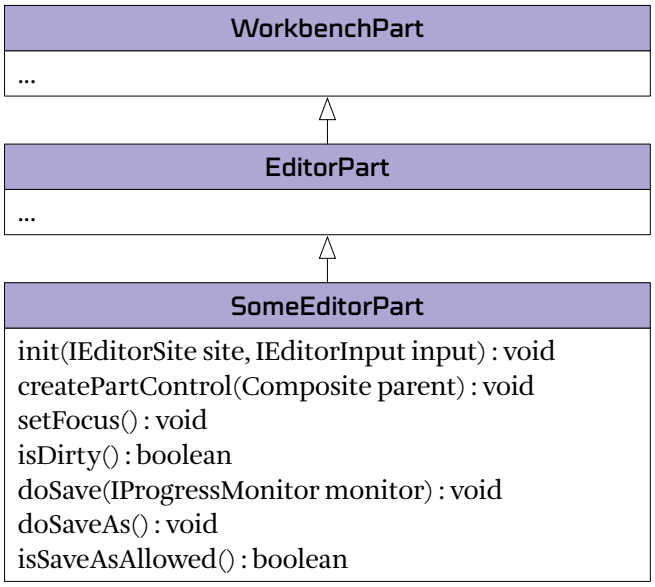
Editor

Editoren werden der Workbench über den Extension Point *org.eclipse.ui.editors* bekannt gegeben:

```
<extension point="org.eclipse.ui.editors">
    <editor
        id="com.example.SomeEditorPart"
        class="com.example.SomeEditorPart"
        name="Some Object Editor">
    </editor>
</extension>
```

Hier ist die implementierende Klasse für den Editor anzugeben. Diese muss von *EditorPart* erben:

EditorPart-Klasse



Der Editor ist so zu implementieren, dass die Daten anhand des EditorInput-Objektes geladen werden und entsprechend angezeigt werden. Beispielsweise:

```
public class SomeEditorPart extends EditorPart {

    public void init(IEditorSite site, IEditorInput input) throws PartInitException {
        // Sobald der Editor erzeugt wird, wird die init-Methode aufgerufen.
        // EditorPart verlangt, dass man die hereingereichte EditorSite und das
        // EditorInput-Objekt setzt. In dieser Methode könnte man auch
        // Nicht-UI bezogene Initialisierungen erledigen.
        setSite(site);
        setInput(input);

        // Sicherstellen, das das Input-Objekt vom erwarteten Typ ist
        Assert.isTrue(input instanceof SomeEditorInput,
                "Input object needs to be SomeEditorInput!");
    }

    public void createPartControl(Composite parent) {
        // Analog zu ViewParts sind in createPartControl die UI-Inhalte des
        // Editors zu erzeugen
        // ...

        // Titel des Editors kann gesetzt werden mit:
        setPartName(/* ... */);
    }

    public void setFocus() {
        // setFocus muss implementiert werden: einem Control im Part
        // den Eingabefokus geben!
        someControl.setFocus();
    }

    public boolean isDirty() {
        // Die Workbench erledigt das Handling des Speichern und die
        // Anzeige des aktuellen Änderungsstatus des Editors. Dazu muss isDirty
        // Auskunft darüber geben, ob der Editor ungesicherte Änderungen enthält
        // (= dirty ist). Dies ist gemäß der Inhalte des Editors herauszufinden und
        // zurückzugeben.
        return /* ... */;
    }

    private void setDirty(boolean dirty) {
```

```
        this.dirty = dirty;
        firePropertyChange(IEditorPart.PROP_DIRTY);
    }

    public void doSave(IProgressMonitor monitor) {
        // doSave wird aufgerufen, sobald der Benutzer den Speichern-Command
        // auslöst und sollte die Änderungen des Editors persistieren.
    }

    public void doSaveAs() {
        throw new UnsupportedOperationException();
    }

    public boolean isSaveAsAllowed() {
        return false;
    }

    // TIPP: getEditorInput überschreiben und ein konkretisiertes
    // EditorInput-Objekt zurückliefern
    public SomeEditorInput getEditorInput() {
        return (SomeEditorInput) super.getEditorInput();
    }
}
```

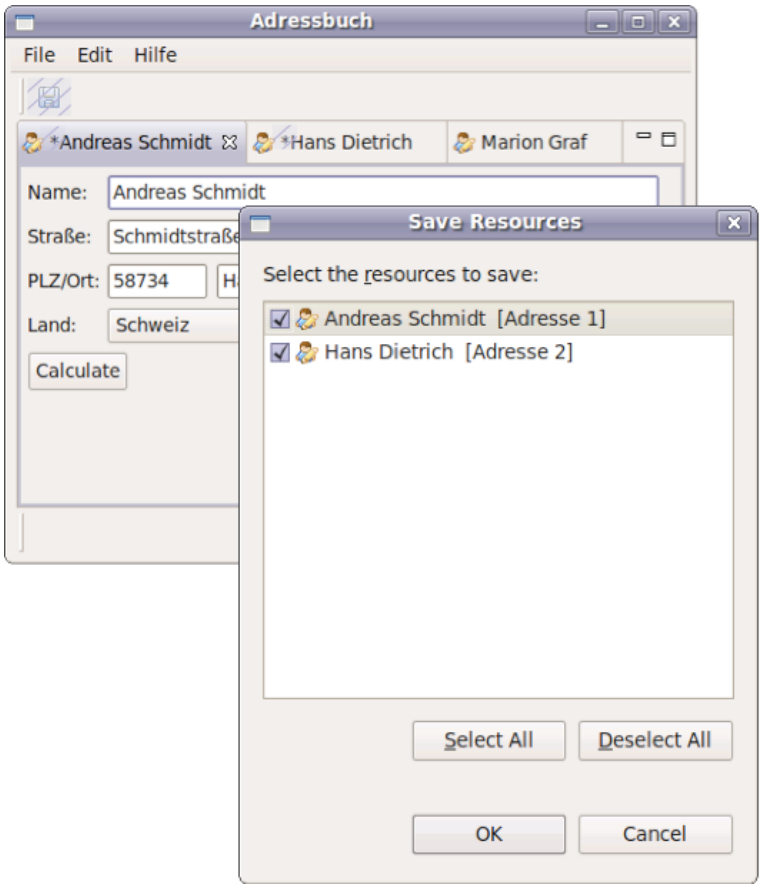
Speichern der Editor-Inhalte

Die Methode *isDirty* ist so zu implementieren, dass sie jederzeit Auskunft darüber gibt, ob ein Editor ungesicherte Änderungen enthält. Damit dieser Status korrekt angezeigt werden kann, muss eine Benachrichtigung erfolgen, sobald sich dieser *dirty*-Status ändert. Dies kann z.B. im Zusammenspiel mit den bearbeitenden UI-Elementen erfolgen:

```
public class SomeEditorPart extends EditorPart {
    private boolean dirty;

    public void createPartControl(Composite parent) {
        // ...
        someTextField.addModifyListener(new ModifyListener() {
            public void modifyText(ModifyEvent e) {
                setDirty(true);
            }
        });
    }
```

```
    public boolean isDirty() {
        return dirty;
    }

    public void setDirty(boolean dirty) {
        this.dirty = dirty;
        firePropertyChange(IEditorPart.PROP_DIRTY);
    }
}
```

Die Workbench erledigt dabei auch die Sicherheitsabfragen, sobald der Benutzer einen Editor mit ungesicherten Änderungen schließen möchte:



Das Speichern wird über den Command *org.eclipse.ui.file.save* ausgelöst. Durch den Standard-Handler für den *save*-Command wird die Methode *doSave* des Editors aufgerufen.

Editoren öffnen, Workbench APIs für Editoren

Um einen Editor zu öffnen, rufen Sie *openEditor()* auf der Workbench-Page mit dem Editor-Input-Objekt und der Editor-ID auf:

```
// Der Zugriff auf Workbench-Page ist situationsabhängig
// Variante 1) in Views / Editoren
page = getSite().getPage();
// Variante 2) in Command-Handlern
page = HandlerUtil.getActiveWorkbenchWindow(event).getActivePage();
// Variante 3) globaler Zugriff
page = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();

// Editor öffnen:
page.openEditor(new SomeEditorInput(someId), SOME_EDITOR_ID);
```

Weitere Methoden für das Handling von Editoren finden Sie ebenfalls auf der Workbench-Page:

```
// Alle Editoren schließen, optional mit Bestätigung bei ungesicherten Änderungen
page.closeAllEditors(true);

// Editoren suchen
IEditorReference[] editorRefs = page.findEditors(someEditorInput, TestEditor.EDITOR_ID,
        IWorkbenchPage.MATCH_INPUT | IWorkbenchPage.MATCH_ID)

// Alle Editoren
IEditorReference[] editorRefs = page.getEditorReferences();

// Editoren schließen, optional mit Bestätigung bei ungesicherten Änderungen
page.closeEditors(editorRefs);
```

Weitere Informationen

- User Interface Guidelines - Editors
  http://wiki.eclipse.org/User_Interface_Guidelines#Editors
- Eclipse RCP Recipes: Creating Custom Eclipse Editors
  http://adamosloizou.blogspot.com/2008/02/eclipse-rcp-recipes-creating-custom.html

TUTORIAL 17.1

Adress-Editor erstellen

Anpassungen im Adressbuch:

- ☐ Im Folgenden wird das Kontextmenü für die Adressliste im Adressbuch-Plug-in benötigt. Wenn Sie es nicht bereits im Commands-Kapitel angelegt haben, holen Sie dies bitte nach, siehe Adresse löschen im Kontextmenü (Seite 121).

Plug-in erstellen:

- ☐ Erstellen Sie ein neues, leeres Plug-in *com.example.addressbook.editor*. Achten Sie dabei darauf, dass *No* für *Would you like to create a rich client application* ausgewählt ist, um eine Erweiterung für die vorhandene Anwendung zu erstellen. Hinweis: Im Folgenden wird ein Plug-in für das Adressbuch entwickelt, welches das Adressbuch um Editierfunktionalität erweitert. Alle folgenden Schritte erfolgen daher in dem soeben erstellten Editor-Plug-in.

Kontextmenü erweitern:

- ☐ Definieren Sie einen neuen Command “Adresse öffnen”.
- ☐ Fügen Sie den Command mit einer Extension zu *org.eclipse.ui.menus* in das Kontextmenüs für Adressen ein.

Produkt aktualisieren und Anwendung starten:

- ☐ Starten Sie die Anwendung mit dem Editor-Plug-in, indem Sie das Editor-Plug-in in Ihrem Produkt unter *Dependencies* hinzufügen und das Produkt mit *Launch an Eclipse application* starten (Hinweis: Dadurch wird die Startkonfiguration gem. der Angaben in der Produktkonfiguration aktualisiert).

Editor erstellen:

- ☐ Definieren Sie in dem neuen Plug-in eine Extension zu *org.eclipse.ui.editors* und definieren Sie einen neuen Adress-Editor.
- ☐ Erstellen Sie eine *AddressEditorInput*-Klasse, die eine Referenz auf eine Adresse beschreibt (Adressen haben eine eindeutige *int*-ID).

Handler für “Adresse öffnen”:

- ☐ Definieren Sie in dem neuen Plug-in einen Handler für den “Adresse öffnen”-Command und legen Sie eine Handler-Klasse an.

- ☐ Implementieren Sie den Handler für den Adresse öffnen-Command so, der Adresseditor für alle ausgewählten Adressen geöffnet wird.

Editor implementieren:

- ☐ Implementieren Sie die Methoden *init* und *createPartControl* im *EditorPart* so, dass die Adresse geladen und angezeigt wird. Kopieren Sie dazu die bereits erstellte Adresseingabemaske.
- ☐ Implementieren Sie die Methode *setFocus* im EditorPart, so dass Sie hier einem Control den Eingabefokus geben (sonst kommt es zu Problemen mit dem Fokus/Selektion beim Wechsel zwischen Tabs).
- ☐ Erweitern Sie den Editor, so dass Adressen bearbeitet und gespeichert werden können.
- ☐ Stellen Sie sicher, dass nur ein Editor für dieselbe Adresse geöffnet werden kann.

Tests:

- ☐ Führen Sie alle Tests inkl. *testEditorsUnique* aus.
- ☐ Stellen Sie sicher, dass Ihre Anwendung auch ohne das Plug-in zur Adressbearbeitung funktioniert, indem Sie das Plug-in temporär in der Startkonfiguration abhaken und die Anwendung starten.

KAPITEL 18

Features, Target Platform

Plug-ins gruppieren: Features

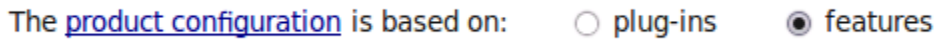
Bei größeren Anwendungen wird die Menge der Plug-ins schnell unübersichtlich. Abhilfe schaffen hier sog. *Features*. Ein Feature besteht neben einigen Angaben wie Name, Versionsnummer aus einer Auflistung von Plug-ins:

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="com.example.somefeature"
      label="Some Feature"
      version="1.0.0">

   <plugin id="com.example.someplugin"/>
   <plugin id="com.example.someotherplugin"/>

</feature>
```

Ein eigenes Feature können Sie mit *File* › *New* › *Other* › *Plug-in Development* › *Feature Project* anlegen. Häufig werden alle Anwendungs-Plug-ins in einem Feature abgelegt, bei größeren Projekten macht auch die Unterteilung in überschaubare Unter-Features Sinn.

Das standardmäßig vorhandene Feature *org.eclipse.rcp* stellt Ihnen alle Plug-ins der RCP-Plattform bereit.

In einer Produktkonfiguration können Sie konfigurieren, dass das Produkt Feature-basiert sein soll. Sie stellen dann lediglich Features statt Plug-ins zusammen, wodurch die Produktkonfiguration deutlich übersichtlicher wird:

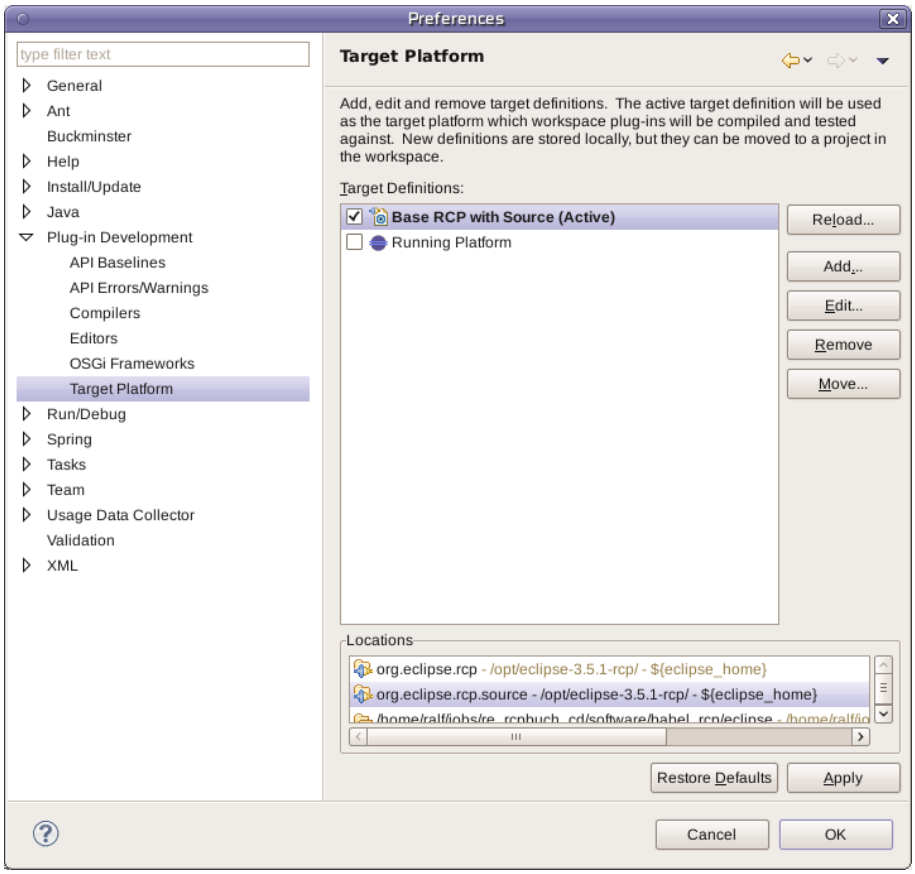


Target Platform

Wie wir gesehen haben, besteht eine RCP Anwendung aus einer Menge Plug-ins, die die Eclipse Plattform erweitern. Ohne die Plug-ins der Plattform könnten sie nicht funktionieren. Woher kommen diese Plattform-Plug-ins?

Werfen wir einen Blick in eine Startkonfiguration: Eigene Plug-ins stehen unter *Workspace*. Die Plug-ins der Eclipse Plattform werden unter *Target Platform* aufgelistet:

Die *Target Platform* definiert die Menge von Plug-ins, auf der eine RCP Anwendung entwickelt und ausgeführt wird. Eine Target Plattform besteht aus einem Ordner mit Plug-ins. Alle Plug-ins in diesem Ordner können bei der Anwendungsentwicklung verwendet werden. Die Target Plattform wird in den Eclipse-Einstellungen unter *Plug-in Development* festgelegt:



Standardmäßig wird die Eclipse IDE selbst als Target Plattform verwendet. Diese enthält jedoch viel mehr Plug-ins als für eine RCP-Anwendung notwendig und sinnvoll sind. Es ist daher sehr empfehlenswert, die Zielplattform von der IDE zu entkoppeln, also eine reduzierte Target Plattform für die RCP-Anwendung zu verwenden, die nur die benötigten Plug-ins enthält:

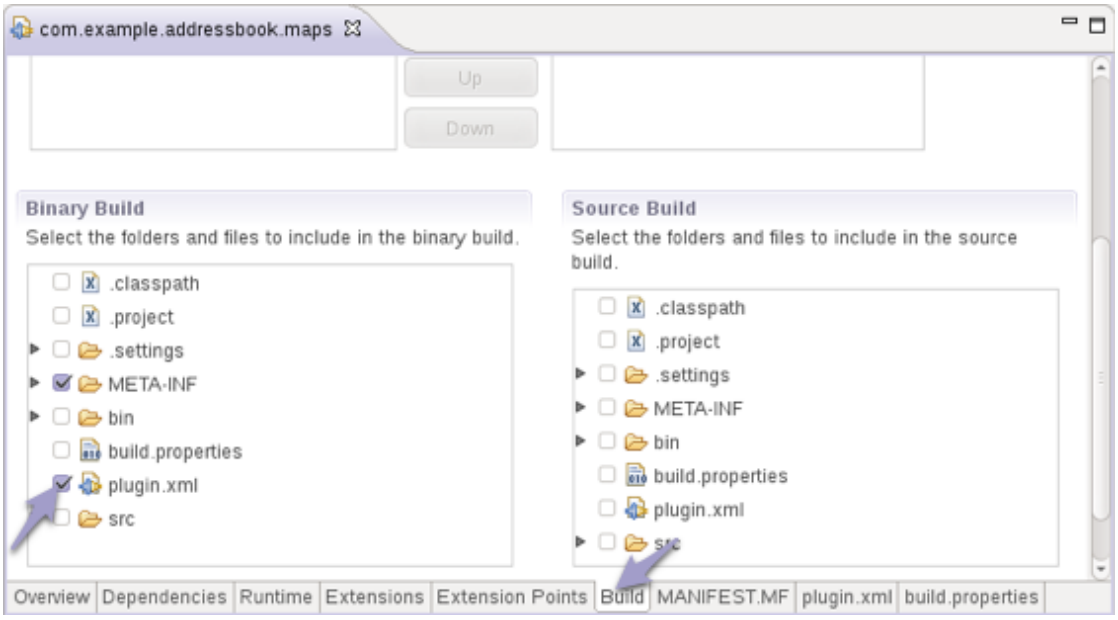
- Alle Entwickler verwenden so dieselbe, exakt definierte Zielplattform.
- Die Version der Plug-ins ist unabhängig von der IDE-Version und Entwickler können neuere Versionen der Eclipse IDE verwenden, auch wenn die Anwendung selbst noch nicht aktualisiert werden soll.
- Es können keine unerwünschten Abhängigkeiten zu IDE-Plug-ins eingeführt werden.
- Häufig benötigen RCP-Anwendung zusätzliche Plug-ins, die nicht in der IDE installiert werden sollen - beispielsweise BIRT (Reporting Framework) oder Plug-ins für andere Betriebssysteme (Eclipse Delta Pack).

Eine minimale Target Plattform für die RCP-Entwicklung finden Sie unter der Bezeichnung “RCP SDK” auf der Eclipse-Downloadseite.

TUTORIAL 18.1

Feature-basierte Produktkonfiguration

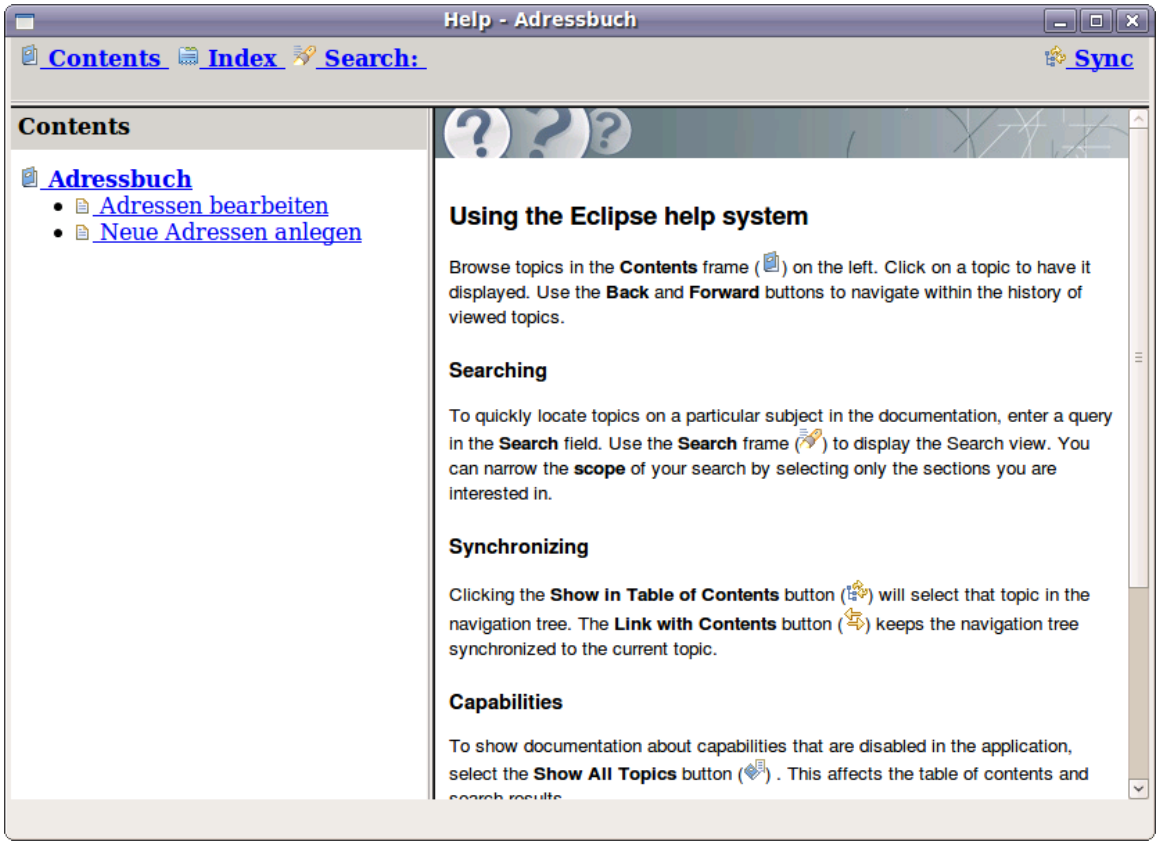
- ☐ Erstellen Sie mit *File* › *New* › *Other* › *Plug-in Development* › *Feature Project* ein neues Feature-Projekt *com.example.addressbook.feature*.
- ☐ Fügen Sie alle *com.example.addressbook.** Plug-ins hinzu.
- ☐ Wählen Sie im Produkt unter *Overview feature-based*.
- ☐ Fügen Sie im Produkt unter *Dependencies* das soeben erstellte Adressbuch-Feature sowie die Features *org.eclipse.rcp* und *org.eclipse.rcp.nl_de* hinzu.
- ☐ Starten Sie die Anwendung im Produkt.
- ☐ Exportieren Sie das Produkt.
- ☐ Testen Sie, das in der exportierten Anwendung das Karten- und Editor-Plug-in funktionieren. Falls nicht, prüfen Sie, ob die Datei *plugin.xml* in den Build dieser Plug-ins eingeht:



KAPITEL 19

Eclipse Hilfe

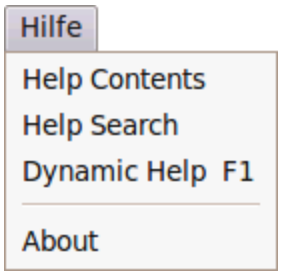
Eclipse RCP stellt ein Hilfesystem zur Einbindung in eigene Anwendungen bereit. Die Anzeige der Hilfeinhalte erfolgt dabei analog zur Eclipse IDE in einem Browser-Fenster, welches aus der Anwendung heraus geöffnet wird:



Die Hilfe-Funktionalität wird von dem Feature *org.eclipse.help* bereitgestellt. Dabei ist zu beachten, dass zusätzlich das Plug-in *org.eclipse.ui.forms* benötigt wird, welches nicht in dem Hilfe-Feature enthalten ist (siehe Bug 294541: Feature org.eclipse.rcp should contain org.eclipse.ui.forms).

Zum Aufrufen der Hilfe können folgende Standard-Commands verwendet werden:

- Inhaltsverzeichnis: *org.eclipse.ui.help.displayHelp*
- Suche: *org.eclipse.ui.help.helpSearch*
- Kontextsensitive Hilfe “Dynamic Help”: *org.eclipse.ui.help.dynamicHelp*



Hilfethemen hinzufügen

Die Hilfetexte werden idealerweise in separaten Hilfe-Plug-ins abgelegt. Für solche Hilfe-Plug-ins gibt es im *Plug-in Project* Assistenten die Vorlage *Plug-in with sample help content*.

Hilfethemen werden über den Extension Point *org.eclipse.help.toc* registriert:

```
<extension point="org.eclipse.help.toc">
   <toc file="someToc.xml" primary="true"/>
   <toc file="toc.xml"/>
</extension>
```

Die *toc.xml*-Dateien beschreiben das Inhaltsverzeichnis der Hilfeinhalte:

```
<?xml version="1.0" encoding="UTF-8"?>
<toc label="Table of Contents">
    <topic label="Some Topic" href="html/sometopic.html"/>
</toc>
```

Die referenzierten Inhalte sind HTML-Dateien, die mit allen HTML-Mitteln formatiert werden können.

Kontextsensitive Hilfe

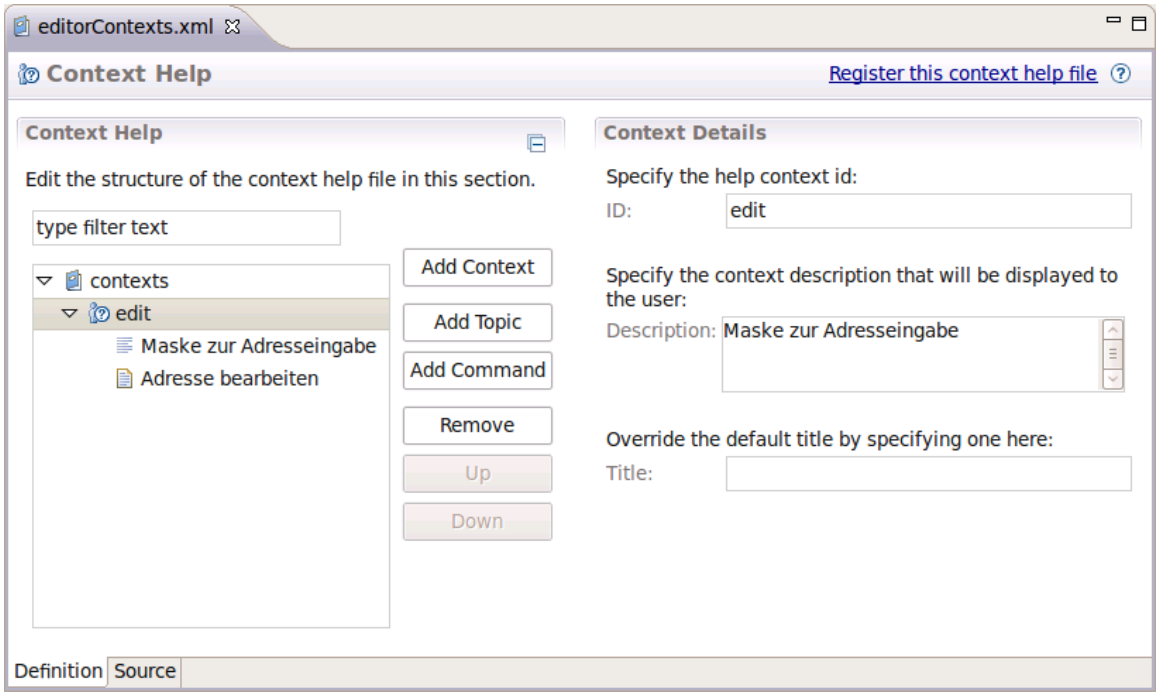
Um dem Anwender eine kontextsensitive Unterstützung anzubieten, müssen zunächst Kontexte für die Anwendung definiert werden. Dies geschieht mit dem Extension Point *org.eclipse.help.contexts*. Mit diesem deklarieren Sie einen Verweis auf eine XML-Datei, in der die Hilfekontexte und -texte für ein Plug-in definiert sind:

```
<extension point="org.eclipse.help.contexts">
    <contexts file="contextsOfSomePlugin.xml" plugin="com.example.someplugin"/>
</extension>
```

Diese Datei hat folgende Struktur:

```
<contexts>
    <context id="message">
              <description>This is the sample context-sensitive help.</description>
              <topic href="html/gettingstarted/subtopic.html" label="Subtopic" />
    </context>
</contexts>
```

Die angegebene XML-Datei mit den Hilfekontexten wird in einem speziellen Editor bearbeitet:



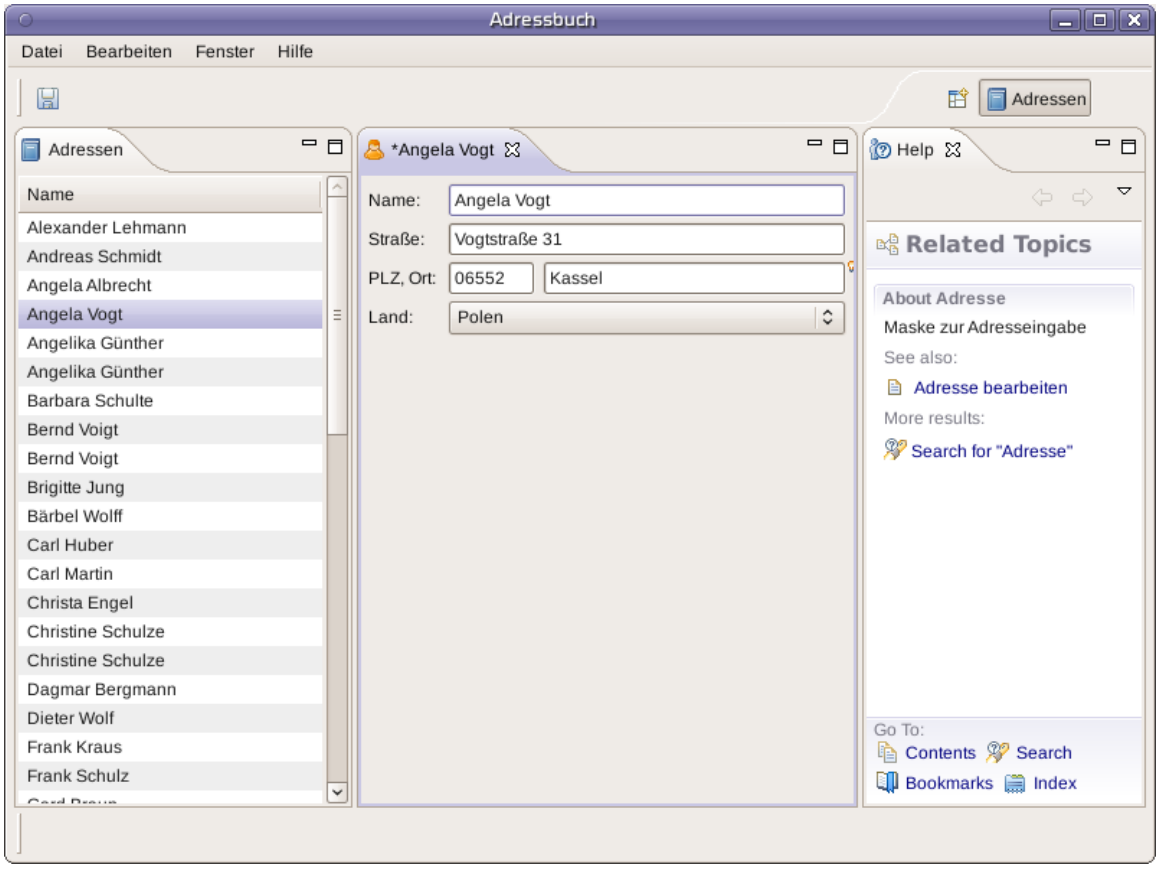
Sie können nun über das Hilfe-System beliebigen SWT-Controls einen Hilfekontext zuordnen:

```
PlatformUI.getWorkbench().getHelpSystem()
        .setHelp(someControl, SomePluginConstants.SOME_HELP_CONTEXT);
```

Für die IDs deklarieren Sie am besten Konstanten in einer separaten Klasse in dem verwendenden Plug-in:

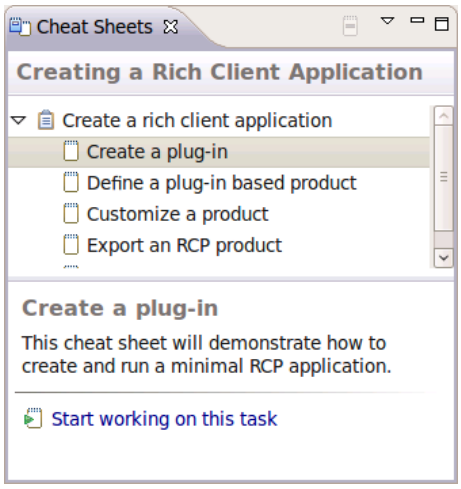
```
public class SomePlugin {

    public static final String PLUGIN_ID = "com.example.someplugin"
    public static final String HELP_CONTEXT_EXAMPLE = PLUGIN_ID + ".edit";

}
```

Aktiviert der Benutzer *Dynamic Help* (Sie könnten dafür den Tastenkürzel F1 vergeben) werden die Hilfethemen zu dem zugeordneten Kontext angezeigt:



Weitere Hilfe-Features

Mit *Cheat Sheets* können dem Benutzer Schritt-für-Schritt Anleitungen in der Anwendung selbst angezeigt werden:



Active Help ermöglicht es, aus Hilfetexten heraus Aktionen in der Anwendung auszulösen, also Java-Code ausgelöst durch einen Link auszuführen. So können Aktionen in der Anwendung in Hilfetexten verlinkt werden.

Weitere Informationen

- User Assistance Overview
  http://www.eclipse.org/eclipse/platform-ua/overview.html
- Adding Help Support to a Rich Client Platform Application
  http://www.eclipse.org/articles/article.php?file=Article-AddingHelpToRCP/index.html
- Dynamic User Assistance in Eclipse-Based Applications
  http://www.eclipse.org/articles/article.php?file=Article-DynamicCSH/index.html
- Dynamic Context Help
  http://richclientplatform.blogspot.com/2008/04/dynamic-context-help.html
- Writing Documentation and Help for Eclipse Projects and Plug-ins
  http://www.keycontent.org/tiki-index.php?page=Eclipse%20Help%20System
- Cheat Sheet Editor
  http://richclientplatform.blogspot.com/2007/09/cheat-sheet-editor.html

TUTORIAL 19.1

Hilfe einbinden

> Fügen Sie im Produkt das Feature *org.eclipse.help* hinzu.

> Fügen Sie im Anwendungs-Feature *com.example.addressbook.feature* das Plug-in *org.eclipse.ui.forms* hinzu.

> Binden Sie die Hilfe-Commands *org.eclipse.ui.help.helpContents*, *org.eclipse.ui.help.helpSearch* und *org.eclipse.ui.help.dynamicHelp* in das Anwendungsmenü ein.

> Erstellen Sie mit *File* › *New* › *Plug-in project* ein Plug-in *com.example.addressbook.help*. Verwenden Sie das Template *Plug-in with sample help content*.

> Fügen Sie das soeben erstellte Plug-in Ihrem Feature hinzu.

> Starten Sie die Anwendung im Produkt über *Launch an Eclipse application* und testen Sie das Hilfesystem.

KAPITEL 20

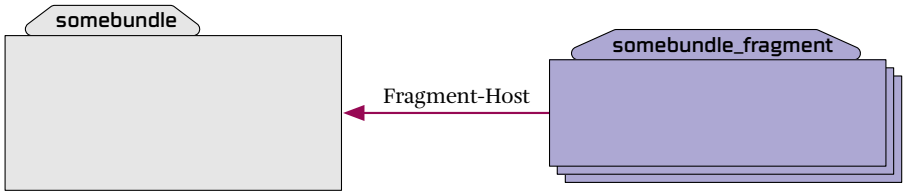
Mehrsprachige Anwendungen

Eclipse RCP-Anwendungen können über die Auslagerung der Texte in Property-Dateien (*Java Resource Bundles*) übersetzt werden. Je Sprache ist dabei eine Property-Datei mit Texten für die jeweilige Sprache bzw. das jeweilige Locale anzulegen. Die Auflösung der Texte erfolgt gemäß der Anwendungssprache mit Fallback auf allgemeinere *Locales*. Es empfiehlt sich daher, eine Standardsprache zu definieren und die Texte für diese in einer Property-Datei ohne Locale-Angabe abzulegen:

```
messages.properties         -> Standard-Sprache (i.d.R. Englisch)
messages_de.properties      -> Alle deutschen Übersetzungen
messages_de_AT.properties   -> Länderspezifische Besonderheiten Österreich
messages_fr.properties      -> Französische Übersetzungen
```

Fragmente

Die Properties-Dateien mit den Übersetzungen können im Plug-in selbst oder in sog. *Fragments* abgelegt werden. Fragmente stellen eine optionale Ergänzung zu einem Host-Plug-in dar. Zur Laufzeit sind die Inhalte der Fragmente so sichtbar, als wären Sie im Host-Plug-in selbst abgelegt. So können Plug-ins nachträglich mit zusätzlichen Ressourcen wie Property-Dateien ergänzt werden.



Die Verwendung von Fragments bietet sich vor allem an, um die Übersetzungen von den Plug-ins der Anwendung separat zu halten oder mehrere, unabhängige Sprachpakete anzubieten.

Plug-in-Dateien übersetzen

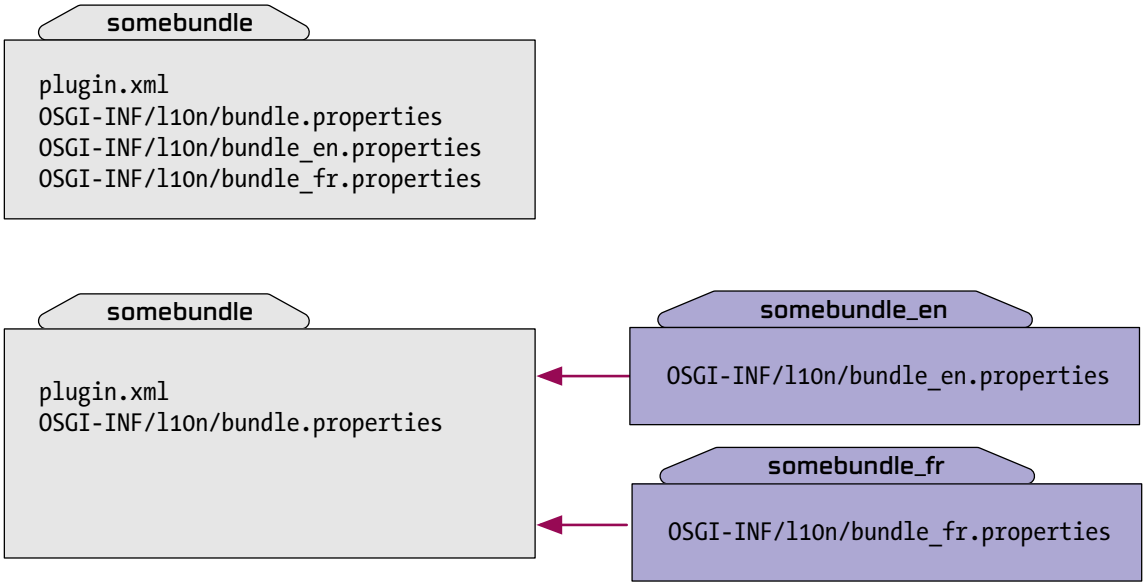
Texte in den *MANIFEST.MF* und *plugin.xml*-Dateien können über *%someKey*-Platzhalter übersetzt werden:

```
<extension point="org.eclipse.ui.perspectives">
    <perspective name="%somePerspectiveName">
        <!-- ... -->
    </perspective>
</extension>
```

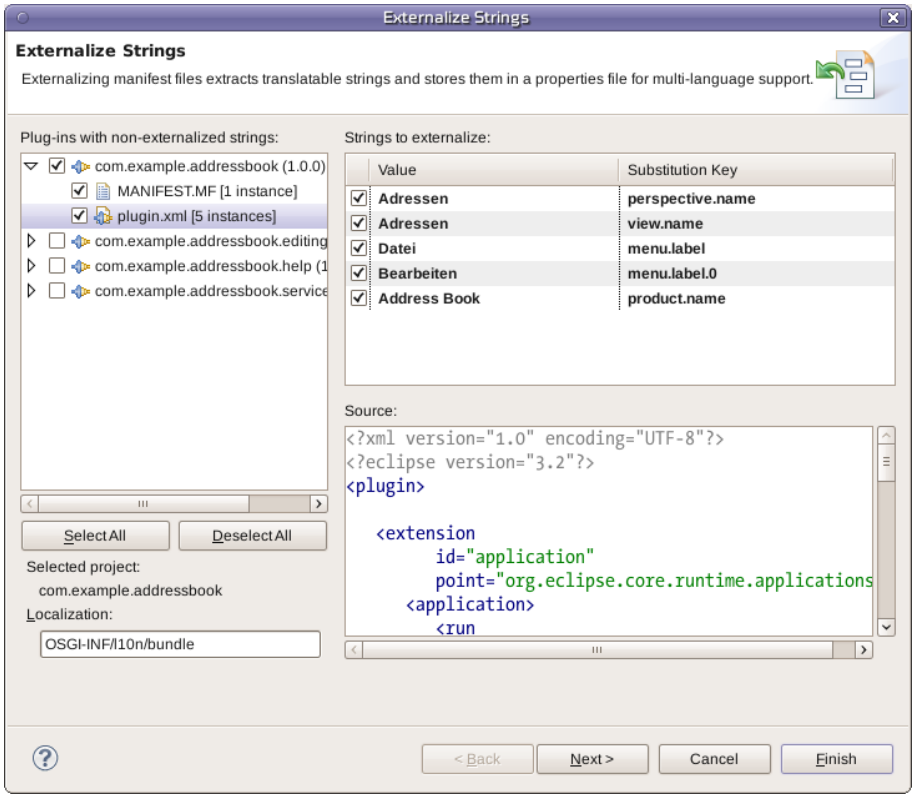
Seit Eclipse 3.5 werden die Übersetzungen für diese Keys standardmäßig in einem *Resource Bundle* unter *OSGI-INF/l10n/bundle.properties* abgelegt. Das Resource Bundle kann sich auch an einem anderen Ort befinden, dann muss im Manifest mit *Bundle-Localization* ein Pfad spezifiziert werden:

```
Bundle-Localization: OSGI-INF/l10n/bundle
```

Die Resource Bundle-Dateien können sich entweder im Plug-in selbst oder in optionalen Fragmenten befinden:



Eclipse PDE stellt zum Extrahieren der Texte aus *plugin.xml*-Dateien mit *PDE Tools › Externalize Strings* einen praktischen Assistenten bereit:



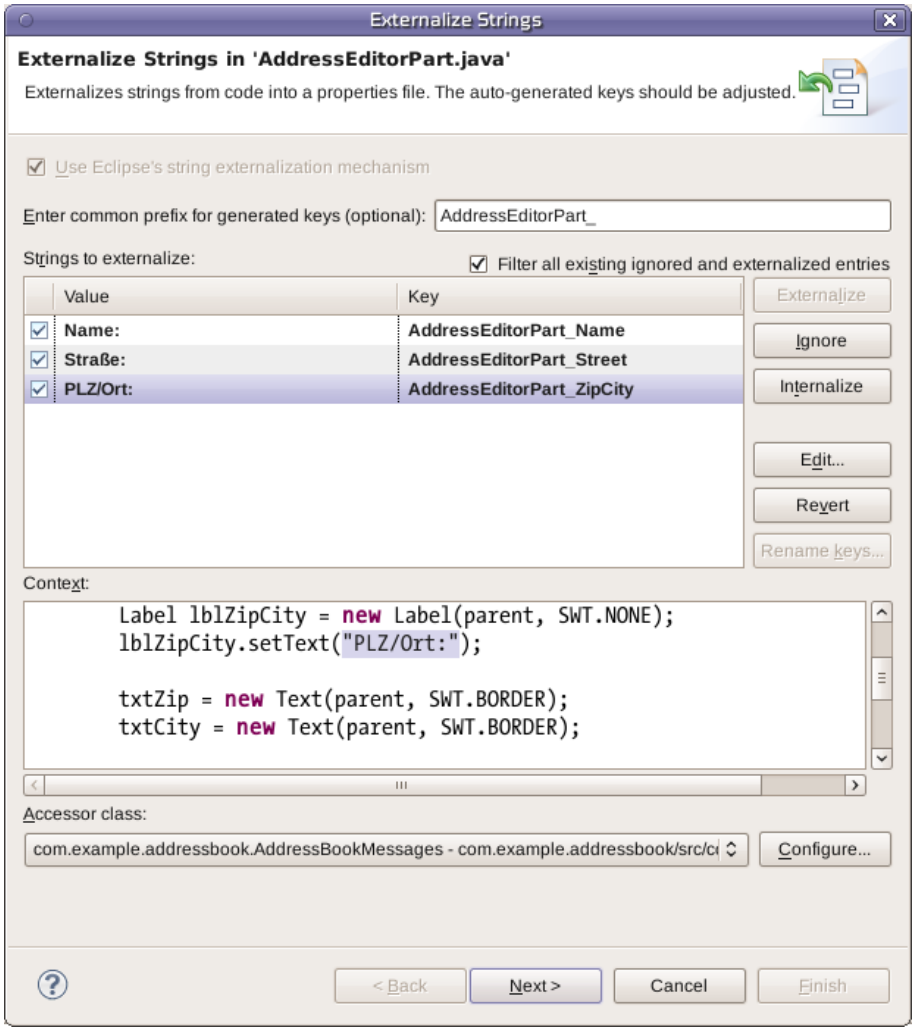
Strings im Code übersetzen

Strings aus *Resource Bundles* erreichen Sie über den Eclipse-eigenen Übersetzungsmechanismus *NLS*. Dazu werden Klassen mit statischen Konstanten für den Zugriff auf die Texte generiert:

```
public class Messages extends NLS {

    private static final String BUNDLE_NAME = "com.example.somebundle.messages";

    public static String SomeString;
    public static String SomeOtherString;

    static {
        // initialize resource bundle
        NLS.initializeMessages(BUNDLE_NAME, Messages.class);
    }

}
```

Die *Resource Bundles* werden dabei anhand des *BUNDLE_NAME* aufgelöst. Diese Angabe bezieht sich nicht auf OSGi Bundles - es handelt sich um die Angabe eines Packages + Dateiname. Obige Klasse würde im Klassenpfad des Plug-ins nach *com/example/somebundle/messages*.properties* suchen.

Auch diese Texte können mit *Source* › *Externalize Strings* automatisch extrahiert werden:



Verwendung von Platzhaltern in Übersetzungen

In den Resource-Bundle-Texten können Platzhalter verwendet werden:

```
SomeMessage=\"{0}\" is the message!
SomeOtherMessage={0} is {1}!
YetAnotherOtherMessage={0} and {1} are {2}!
```

Die entsprechenden Texte werden durch Aufruf von *NLS.bind* eingesetzt:

```
message = NLS.bind(Messages.SomeMessage, obj)
message = NLS.bind(Messages.SomeOtherMessage, obj1, obj2);
message = NLS.bind(Messages.YetAnotherOtherMessage, new Object[] {obj1, obj2, obj3});
```

Texte aus der Plattform übersetzen

Für alle Texte der Plattform-Plug-ins stellt das *Eclipse Babel*-Projekt Übersetzungen bereit, die der Target Platform hinzugefügt werden können, um die Plattform selbst zu übersetzen.

Sprache festlegen

Die Sprache wird beim Start der Anwendung einmalig festgelegt. In den Standard-Programmargumenten der Startkonfiguration wird die Sprache auf die Sprache der IDE konfiguriert:

```
-os ${target.os} -ws ${target.ws} -arch ${target.arch} -nl ${target.nl}
```

Für *target.nl* kann ein beliebiges Java-Locale angegeben werden, um die Anwendung in einer bestimmten Sprache zu testen. Erfolgt keine *Locale*-Angabe mit den Programmargumenten, wird das Standard-Java-*Locale* verwendet, welches von Java automatisch plattformabhängig aus den Ländereinstellungen des Betriebssystems ermittelt wird.

Weitere Informationen

- Eclipse RCP: Mehrsprachigkeit (Mitschnitt Live-Demo)
  http://vimeo.com/21847847
- How to Internationalize your Eclipse Plug-In
  http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html
- Eclipse Babel Project
  http://www.eclipse.org/babel/
- WindowBuilder Internationalization (i18n) / Localization
  http://code.google.com/intl/de-DE/javadevtools/wbpro/features/internationalization.html
- Eclipse ResourceBundle Editor
  http://sourceforge.net/projects/eclipse-rbe/
- Internationalization and RCP
  http://eclipsesource.com/blogs/2008/07/23/tip-internationalization-and-rcp/
- Bug 294406: Provide Babel translation features for RCP features
  http://bugs.eclipse.org/294406
- Dynamic Language Switcher for Eclipse RCP Apps
  http://www.toedter.com/blog/?p=19
- ResourceBundle (Java 2 Platform SE 5.0)
  http://download.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html

TUTORIAL 20.1

Anwendung übersetzen

- ☐ Fügen Sie Ihrer Produktkonfiguration das Feature *org.eclipse.rcp.nl_de* hinzu (dieses Feature wurde manuell zusammengestellt und enthält die für die RCP-Plattform benötigten Übersetzungsfragmente aus dem Babel-Projekt).
- ☐ Extrahieren Sie die Texte aus dem *plugin.xml* des Editor-Plug-ins mittels *PDE Tools* › *Externalize Strings*.
- ☐ Extrahieren Sie die Texte aus dem Code des Adress-Editors mittels *Source* › *Externalize Strings*.
- ☐ Passen Sie die generierten Properties-Dateien so an, dass englische Übersetzungen in **.properties* und die deutschen Übersetzungen in **_de.properties* liegen.
- ☐ Starten Sie die Anwendung einmal in Deutsch und einmal in Englisch, indem Sie den *-nl* Parameter in der Startkonfiguration auf *de_DE* bzw. *en_US* setzen.
- ☐ Exportieren Sie die Anwendung und testen Sie die korrekte Mehrsprachigkeit der exportierten Anwendung.

Häufige Probleme:

Übersetzungen werden nicht angezeigt: Eclipse cacht Übersetzungen intern. Inkrementieren Sie die Versionsnummer des Plug-ins mit den Übersetzungen oder löschen Sie die *Configuration Area* der Anwendung (*Startkonfiguration* › *Configuration* › *Clear the configuration area before launching*).

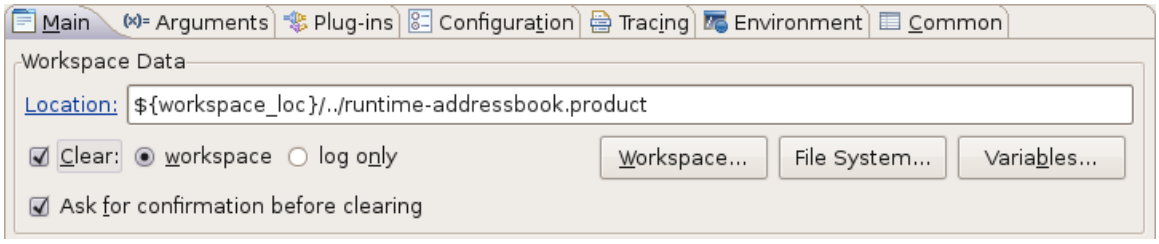
Übersetzungen werden beim Start des exportieren Produkts nicht angezeigt: Prüfen Sie, dass die Übersetzungsdateien im *build.properties* angegeben sind und in dem exportierten JAR vorhanden sind.

KAPITEL 21

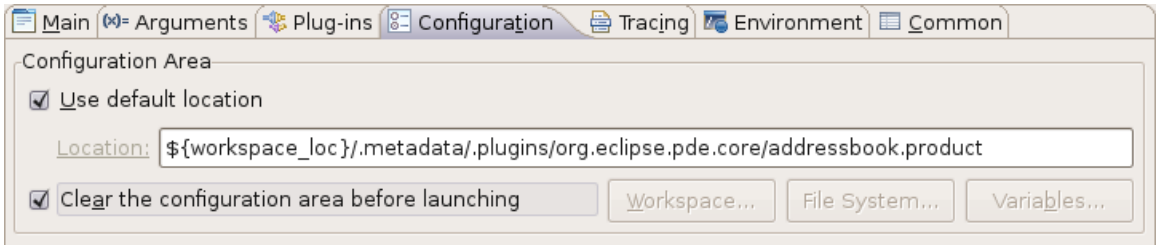
Einstellungen und Konfiguration

Eclipse-Anwendungen haben zwei Ordner, in denen Dateien zur Laufzeit gespeichert werden:

Der **Workspace** ist der zentraler Ordner für Benutzerdateien. Jede Anwendung wird mit einem Workspace-Ordner gestartet, der von der Eclipse Plattform verwaltet wird. Zur Laufzeit wird der Ordner über die Eigenschaft *osgi.instance.area* bzw. über das Kommandozeilenargument *-data* konfiguriert. Standardmäßig liegt er unter *workspace* neben dem Launcher-Binary. Beim Start aus der IDE können Sie den Ordner in der *Run Configuration* konfigurieren und beim Start der Anwendung zurücksetzen lassen:



Neben dem *Workspace* für Benutzerdateien existiert für Plug-ins der Laufzeitumgebung und Eclipse Runtime noch die **Configuration Area**. Diese ist als interner Ordner der Eclipse Plattform zu betrachten. Hier werden vor allem Metadaten, Caches sowie interne Konfigurationsdateien (z.B. die Startkonfiguration *config.ini*) abgelegt. Der Ordner wird über *osgi.configuration.area* bzw. *-configuration* konfiguriert und liegt standardmäßig im Anwendungsordner unter *configuration*. Beim Start aus der IDE können Sie den Ordner in der *Run Configuration* konfigurieren und beim Start der Anwendung zurücksetzen lassen:



Zustand der Workbench speichern

Die Eclipse Workbench bietet die Möglichkeit, den Zustand der Workbench automatisch zu speichern. Dabei werden z.B. die Größe des Workbench-Fensters, das Layout aller geöffneten Perspektiven sowie die zuletzt aktive Perspektive gespeichert und beim nächsten Start wiederhergestellt. Um dieses Feature zu aktivieren, ergänzen Sie den *WorkbenchAdvisor* der Anwendung um die Konfigurationsoption *saveAndRestore*:

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    public void initialize(IWorkbenchConfigurer config) {
        config.setSaveAndRestore(true);
    }

}
```

Diese Zustandsinformationen werden in sogenannten *Memento*-Objekten abgelegt. Ein *Memento* repräsentiert den Zustand eines Objektes in einer persistierbaren Hierarchie von *Key-Value*-Paaren. Dieses *Memento* kann um weitere Informationen ergänzen werden, indem die Methoden *saveState* und *restoreState* der *WorkbenchAdvisor* bzw. *WorkbenchWindowAdvisor*-Klasse implementiert werden. Zum Beispiel:

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    @Override
    public void initialize(IWorkbenchConfigurer configurer) {
        configurer.setSaveAndRestore(true);
    }

    @Override
    public IStatus saveState(IMemento memento) {
        memento.createChild("myApp").putString(
            "lastOpenedDate", DateFormat.getDateTimeInstance().format(new Date()));
        return super.saveState(memento);
    }

    @Override
    public IStatus restoreState(IMemento memento) {
        if (memento != null) {
            IMemento myAppMemento = memento.getChild("myApp");
            if (myAppMemento != null)
                System.out.println("Last opened on: " +
                    myAppMemento.getString("lastOpenedDate"));
```

```
        }
        return super.restoreState(memento);
    }
}
```

Die Memento-Objekte werden im Anwendungs-Workspace unter *.metadata/.plugins/org.eclipse.ui.workbench* persistiert.

Nach der Aktivierung von *saveAndRestore* gilt es zu beachten, dass das Layout einer Perspektive gespeichert wird, sobald diese einmal geöffnet wurde. Änderungen an der *PerspectiveFactory* bzw. Contributions über *perspectiveExtensions* wirken daher erst nach einem Zurücksetzen der Perspektive. Dies kann direkt in der Anwendung mit dem Command "Perspektive zurücksetzen" (*org.eclipse.ui.window.resetPerspective*) oder über ein Zurücksetzen des gesamten Workspace in der Startkonfiguration geschehen.

Zustand von Views persistieren

Wird der *saveAndRestore*-Mechanismus der Workbench aktiviert, können auch Views durch Überschreiben der Methoden *saveState* und *init* Einstellungen in einem Memento-Objekt ablegen:

```
public class SomeViewPart extends ViewPart {

    private static final String MEMENTO_SOME_SETTING = "SomeSetting";

    @Override
    public void saveState(IMemento memento) {
        memento.putString(MEMENTO_SOMETHING, "abc");
    }

    @Override
    public void init(IViewSite site, IMemento memento) throws PartInitException {
        super.init(site, memento);
        if (memento != null) {
            String someSetting = memento.getString(MEMENTO_SOME_SETTING));
        }
    }

}
```

Eigene Dateien je Plug-in ablegen

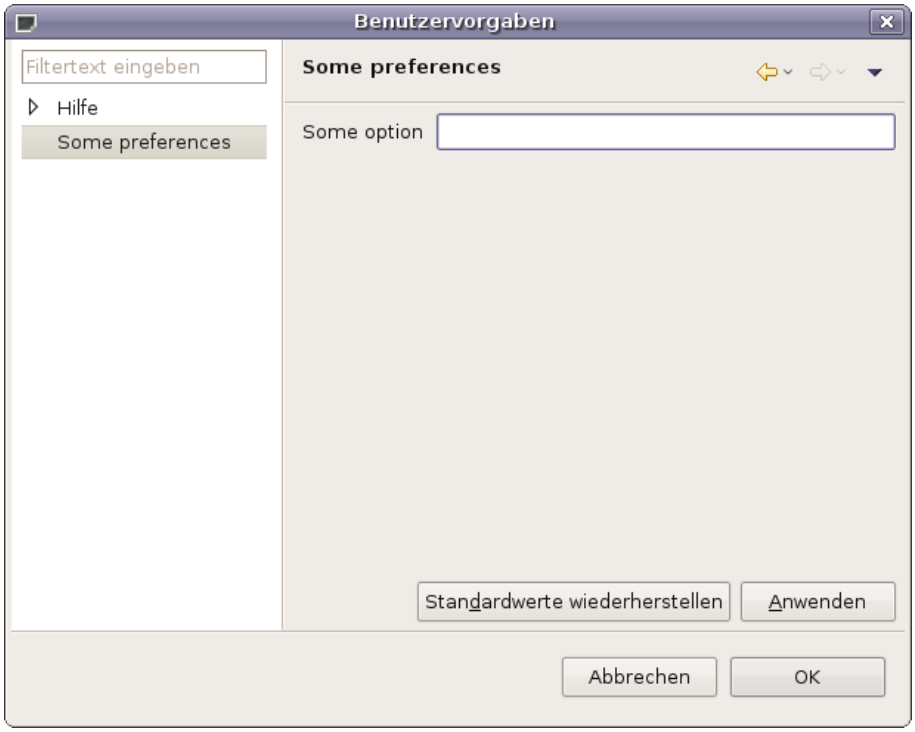
Zur Ablage von eigenen Dateien in einem Plug-in kann über den *Activator* ein *State*-Ordner erreicht werden. Dieser bietet sich insbesondere an, wenn ein Plug-in seinen Zustand in einem eigenen Datenformat persistieren möchte. Die Daten werden im *workspace* unter *.metadata/.plugins/<bundle-id>/* abgelegt.

```
IPath stateLocation = Activator.getDefault().getStateLocation();
File stateFile = stateLocation.append("state.txt").toFile();
PrintWriter writer = new PrintWriter(stateFile);
writer.println("some state");
writer.close();
```

Einstellungen

Plug-in-spezifische Konfigurationsoptionen können über den Eclipse-eigenen *Preference Store* verwaltet werden. Diese Optionen werden im Workspace unter *.metadata/.plugins/org.eclipse.core.runtime/.settings/<bundle-id>.prefs* abgelegt. Diese Variante bietet sich für Konfigurationsoptionen an, die vom Benutzer über den Einstellungsdialog der Anwendung konfiguriert werden sollen.

Der Eclipse-Einstellungsdialog kann der Anwendung mit dem Standard-Command *org.eclipse.ui.window.preferences* hinzugefügt werden:



Über den Extension Point *org.eclipse.ui.preferencePages* können Plug-ins diesen Dialog um eigene Einstellungsseiten ergänzen:

```
<extension point="org.eclipse.ui.preferencePages">
   <page
         class="com.example.SomePreferencePage"
         id="com.example.SomePreferencePage"
         name="Some preferences">
   </page>
</extension>
```

Neben eigenen Implementierungen des Interfaces *IWorkbenchPreferencePage* steht die Basisklasse *FieldEditorPreferencePage* zur Verfügung, die über *FieldEditor*-Objekte die Bearbeitung von Einstellungsoptionen abstrahiert, z.B.:

```
public class SomePreferencePage extends FieldEditorPreferencePage
    implements IWorkbenchPreferencePage {

    public final static String SOME_OPTION = "SomeOption";

    public SomePreferencesPage() {
        // Layout
```

```
        super(GRID);
    }

    @Override
    protected void createFieldEditors() {
        // Felder der Einstellungsseite hinzufügen
        addField(new StringFieldEditor(SOME_OPTION, "Some Option", getFieldEditorParent()));
    }

    @Override
    public void init(IWorkbench workbench) {
        // Preference Store des eigenen Plug-ins setzen
        setPreferenceStore(Activator.getDefault().getPreferenceStore());
    }

}
```

Folgende Editor-Typen stehen dabei zur Verfügung: *BooleanFieldEditor*, *IntegerFieldEditor*, *StringFieldEditor*, *RadioGroupFieldEditor*, *ColorFieldEditor*, *FontFieldEditor*, *DirectoryFieldEditor*, *FileFieldEditor* und *PathEditor*

Der programmatische Zugriff auf diese Einstellungen erfolgt über den *PreferenceStore* des jeweiligen Plug-ins:

```
IPreferenceStore preferenceStore = Activator.getDefault().getPreferenceStore();
String value = preferenceStore.getString(SomePreferencePage.SOME_OPTION);
```

Es ist empfehlenswert, Veränderungen an den Eigenschaften zu beobachten, um auf Änderungen an den Einstellungen zur Laufzeit reagieren zu können:

```
preferenceStore.addPropertyChangeListener(new IPropertyChangeListener() {

    @Override
    public void propertyChange(PropertyChangeEvent event) {
        if (SomePreferencePage.SOME_OPTION.equals(event.getProperty())) {
            // handle change from event.getOldValue() to event.getNewValue()
        }
    }
});
```

Vordefinierte Preference Pages

Die Klasse *org.eclipse.ui.ExtensionFactory* definiert unter anderem fertige Einstellungsseiten, die in eigene Anwendungen eingebunden werden können, z.B.:

```
<extension point="org.eclipse.ui.preferencePages">
    <page
        class="org.eclipse.ui.ExtensionFactory:newKeysPreferencePage"
        id="newKeysPreferencePage"
        name="Keys"/>
</extension>
```

Weitere Informationen

- Eclipse runtime options
  http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/misc/runtime-options.html
- Eclipse Preferences - Tutorial
  http://www.vogella.de/articles/EclipsePreferences/article.html
- Getting Started with Preference Pages
  http://eclipsepluginsite.com/preference-pages.html
- Re-using the Eclipse proxy preference settings
  http://www.vogella.de/blog/2009/12/08/eclipse-rcp-proxy-preference/

TUTORIAL 21.1

Layout persistieren und konfigurierbarer Kartendienst

- ☐ Aktivieren Sie die *saveAndRestore*-Funktionalität der Anwendung und testen Sie, ob das Layout der Workbench auch nach dem Neustart der Anwendung beibehalten wird.
- ☐ Ändern Sie eine Perspektive und testen Sie, ob sich die Änderung in der Anwendung auswirkt.
- ☐ Ergänzen Sie in der Menüleiste den Command *org.eclipse.ui.window.resetPerspective*, mit dem Perspektiven der Anwendung in den Ursprungszustand zurückgesetzt werden können.
- ☐ Ergänzen Sie in der Menüleiste den Command *org.eclipse.ui.window.preferences* zur Anzeige der Anwendungseinstellungen.
- ☐ Erweitern Sie in das Karten-Plug-in über eine Extension zu *org.eclipse.ui.preferencePages* eine Einstellungsseite “Karte”.
- ☐ Implementieren Sie die *Preference Page* so, dass sie eine Konfigurationsoption “Kartendienst” mit den Auswahlmöglichkeiten “Google Maps”, “Bing” und “OpenStreetMap” anbietet.

KAPITEL 22

JFace Data Binding

JFace Data Binding stellt einen Framework bereit, um Werte aneinander zu binden. Dies kann man in RCP-Anwendungen verwenden, um Eigenschaften von Modellobjekten mit entsprechenden GUI-Elementen zu verknüpfen. Bindet man z.B. eine Eigenschaft eines Modellobjektes an die Text-Eigenschaft eines SWT-Text-Steuerelementes, synchronisiert JFace Data Binding diese Werte automatisch. Wird der Text im Textfeld verändert, wird auch die Eigenschaft des Modellobjektes aktualisiert (und umgekehrt). Dabei anfallende Aufgaben wie Konvertierung, Validierung und Beachtung von Thread-Zugriffsregeln werden dabei von dem Framework erledigt.

Objekte beobachten: Properties und Observables

Ein *Observable* ist eine Abstraktion für etwas, das sich beobachten lässt, d.h. bei Veränderung ein Ereignis auslöst. Solche *Observables* werden über Property-Objekte erzeugt. Bei der Beobachtung von Java-Objekten hat man die Wahl zwischen *BeansObservables* und *PojoObservables* (*Plain Old Java Object*). Ein Bean meint hier ein Objekt, welches *PropertyChangeSupport* gemäß der Java Bean Spezifikation unterstützt und bei Änderungen an Eigenschaften ein Ereignis auslöst, z.B.:

```
public class Person {

    private PropertyChangeSupport changes = new PropertyChangeSupport(this);

    private String name;
    private String vorname;

    public String getName() { return name; }
    public String getVorname() { return name; }

    public void setName(String name) {
        changes.firePropertyChange("name", this.name, this.name = name);
    }

    public void setVorname(String vorname) {
        changes.firePropertyChange("vorname", this.vorname, this.vorname = vorname);
    }

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        changes.addPropertyChangeListener(listener);
    }
```

```
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        changes.removePropertyChangeListener(listener);
    }

}
```

Für eine Eigenschaft eines solchen Objektes kann mittels der Klasse *BeanProperties* ein beobachtbarer Wert, ein *IObservableValue*, erzeugt werden:

```
IObservableValue nameObservable = BeanProperties.value("name").observe(person);
```

Um ein Objekt, welches *PropertyChangeSupport* nicht implementiert, zu binden, kann die Klasse *PojoProperties* verwendet werden. Bedingung dabei ist jedoch, dass das Objekt *ausschließlich* durch JFace Data Binding manipuliert wird:

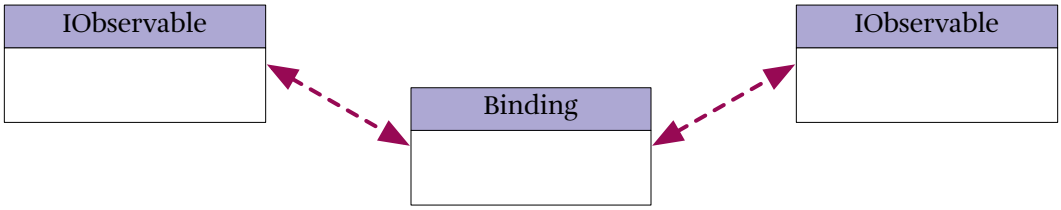
```
IObservableValue nameObservable = PojoProperties.value("name").observe(person);
```

Eigenschaften von SWT-Steuerelementen können mit *WidgetProperties* beobachtet werden:

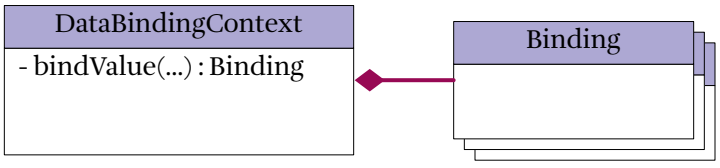
```
Text textWidget = new Text(parent, SWT.BORDER);
IObservableValue nameTextObservable = WidgetProperties.text(SWT.Modify).observe(textWidget);
```

Binding

Mit einem *Binding* werden zwei *Observables* miteinander verbunden und sind fortan automatisch synchronisiert:



Für die Erzeugung und Verwaltung von *Bindings* ist ein *DataBindingContext* verantwortlich:



Die *Observables* aus dem obigen Beispiel könnten folgendermaßen gebunden werden:

```
DataBindingContext context = new DataBindingContext();
context.bindValue(nameTextObservable, nameObservable);
context.bindValue(vornameTextObservable, vornameObservable);
```

Dabei ist die Reihenfolge der *Observable*-Parameter zu beachten. Das erste *Observable* ist das *Target*, das zweite das *Model*:

```
context.bindValue(target, model);
```

Initial synchronisiert JFace Data Binding den Wert vom *Model* zum *Target*, d.h. der initiale Zustand des Targets wird verworfen.

Update-Strategien

Über *UpdateValueStrategy*-Objekte kann der Weg vom Modell ins Target-Observable (und umgekehrt) konfiguriert werden. Diese können bei der Erzeugung des Bindings angegeben werden:

```
context.bindValue(targetObservable, modelObservable, targetToModel, modelToTarget);
```

Dabei kann eine *updatePolicy* spezifiziert werden, die den Zeitpunkt der Aktualisierung des Zielobjektes bestimmt:

- *UpdateValueStrategy.POLICY_NEVER*: Observable wird nicht beobachtet und es erfolgt keine Aktualisierung des Ziel-Observable.
- *UpdateValueStrategy.POLICY_ON_REQUEST*: Aktualisierung erfolgt nur manuell durch Aufruf von *updateModelToTarget* oder *updateTargetToModel* auf dem Binding bzw. *updateModels* oder *updateTargets* auf dem *DataBindingContext*
- *UpdateValueStrategy.POLICY_CONVERT*: Konvertierung von Werten erfolgt immer, Aktualisierung der Objekte analog zu *POLICY_ON_REQUEST*
- *UpdateValueStrategy.POLICY_UPDATE* (Standard): Konvertierung, Validierung und Aktualisierung des Zielobjektes erfolgt bei jeder Änderung des Modellobjektes.

Beispiel: Um die Bearbeitung ein Feldes für die Bearbeitung im GUI zu sperren, würde man den Weg von UI-Observable (*Target*) zum Modell-Observable mit *UpdateValueStrategy.POLICY_NEVER* versehen:

```
context.bindValue(dateText, dateProperty,
    new UpdateValueStrategy(UpdateValueStrategy.POLICY_NEVER),
    new UpdateValueStrategy(UpdateValueStrategy.POLICY_UPDATE));
```

Konvertierung

Die Werte der gebundenen *Observables* können auf dem Weg von Modell zu Target und Target zu Modell konvertiert werden. Dazu ist der *UpdateValueStrategy* mit *setConverter* ein Konverter zu setzen. Dies ist insbesondere notwendig, wenn die gebundenen Werte nicht den gleichen Typ haben und daher eine manuelle Typkonvertierung erfolgen muss. Für simple Konvertierungen wie *String* <-> *int* verwendet JFace Data Binding automatisch einen passenden Konverter.

Eine Datumskonvertierung kann z.B. folgendermaßen realisiert werden:

```
UpdateValueStrategy modelToTarget = new UpdateValueStrategy();
UpdateValueStrategy targetToModel = new UpdateValueStrategy();
modelToTarget.setConverter(new Converter(String.class, Date.class) {

    public Object convert(Object fromObject) {
        return SimpleDateFormat.getDateInstance().parse(String.valueOf(fromObject));
    }

});

context.bindValue(dateText, dateProperty,
    targetToModel, modelToTarget);
```

Validierung

Über die *UpdateValueStrategy* kann auch ein Validator für die Wege zwischen Modell und Zielobjekt festgelegt werden. Die Validierung kann dabei zu drei verschiedenen Zeitpunkten erfolgen:

- *setAfterGetValidator*: Der unkonvertierte Wert aus dem Quellobjekt wird validiert.
- *setAfterConvertValidator*: Der konvertierte Wert wird direkt nach der Konvertierung validiert.
- *setBeforeSetValidator*: Der konvertierte Wert wird validiert, bevor er dem Zielobjekt gesetzt wird. Hier sollte vor allem komplexere Validierungslogik platziert werden, die bei einem einfachen *afterConvert* Validierungsfehler nicht ausgeführt werden soll.

Die Rückmeldung der *Validator*-Klasse erfolgt über Status-Objekte, die mit *ValidationStatus* erzeugt werden können:

```
UpdateValueStrategy targetToModel = new UpdateValueStrategy();
targetToModel.setAfterGetValidator(new IValidator() {

    public IStatus validate(Object value) {
```

```
        if (String.valueOf(value).length() > 10)
            return ValidationStatus.error("Maximal 10 Zeichen!");
        else
            return ValidationStatus.ok();
    }

});
```

Felder mit Validierungsfehlern dekorieren

Einem gebundenem Eingabefeld kann mit der Klasse *ControlDecorationSupport* eine Dekoration hinzugefügt werden, die automatisch den aktuellen Validierungsstatus anzeigt:

Beim erstellen der Dekoration ist lediglich das *Binding* anzugeben, die beteiligten Steuerelemente werden automatisch ermittelt:

```
Binding binding = context.bindValue(someTextField, someProperty, uiToModel, modelToUi);
ControlDecorationSupport.create(binding, SWT.TOP | SWT.RIGHT);
```

Feldübergreifende Validierungsregeln

Eine Validierungsregel kann sich auch über mehrere Felder erstrecken. Dazu verwendet man einen *MultiValidator*, der bei der Validierung mehrere Observables berücksichtigt und den *DataBindingContext* um das Validierungsergebnis ergänzt:

```
MultiValidator datesInOrder = new MultiValidator() {

    @Override
    protected IStatus validate() {
        Date startDate = (Date) startDateObservable.getValue();
        Date endDate = (Date) endDateObservable.getValue();
        return (endDate.after(startDate)) ? ValidationStatus.ok() : ValidationStatus
                .error("End date needs to be after start date.");
```

```
    }
};

context.addValidationStatusProvider(datesInOrder);
```

Wichtig ist dabei, dass die Abhängigkeiten des Validators (d.h. welche Änderungen eine Neu-Validierung notwendig machen) automatisch durch die Beobachtung der Zugriffe auf die Target-Observables in der *validate*-Methode ermittelt werden. Daher sollte das Holen dieser Observables nicht von Bedingungen abhängig sein, die sich zur Lebenszeit des Validators ändern können.

Beispiele zur feldübergreifenden Validierung:

> Snippet021MultiFieldValidation
> *http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.jface.examples.databinding/src/org/eclipse/jface/examples/databinding/snippets/Snippet021MultiFieldValidation.java?view=markup*

Realms

Unter Umständen muss der Zugriff auf die beobachteten Objekte oder die Logik zur Validierung in einem bestimmten Thread erfolgen. JFace Data Binding bildet diese Notwendigkeit über *Realms* ab. Ein *Realm* definiert einen Kontext, aus dem heraus ein Zugriff auf Objekte erfolgen soll.

Sie können z.B. beim Erzeugen von *Observables* einen solchen Realm angeben:

```
BeansObservables.observeValue(realm, bean, propertyName)
```

Über die angegebene Realm-Implementierung könnten Sie den Zugriff auf das Bean in einem bestimmten Kontext ausführen lassen. Wird kein Realm angegeben, wird der *Default-Realm* verwendet. Der Default-Realm kann explizit spezifiziert werden, indem das Binding in einem *Realm.runWithDefault()*-Block durchgeführt wird:

```
Realm.runWithDefault(defaultRealm, new Runnable() {

    public void run() {
        DataBindingContext context = new DataBindingContext();
        // binding logic
    }
}
```

Eine RCP-Anwendung wird immer mit dem SWT-Realm als *Default-Realm* ausgeführt, alle Data Binding Zugriffe erfolgen daher standardmäßig auf dem UI-Thread.

Viewer binden

Mit der Klasse *ViewerSupport* können *JFace Viewer* an Listen gebunden werden. Dabei wird sowohl die Liste selbst beobachtet (das Hinzufügen oder Entfernen von Elementen führt zur Aktualisierung der Tabellenansicht) als auch die Objekt-Attribute selbst:

```
IObservableList list = new WritableList(someList, SomeObject.class);
ViewerSupport.bind(someTableViewer, list,
        BeanProperties.values(new String[] { "name", "street", "country.name" }));
```

Um die Selektion eines *Viewers* zu beobachten, stehen in der Klasse *ViewersObservables* Methoden zur Verfügung, mit dem entsprechende Observables erzeugt werden können:

```
IObservableValue selection = ViewerProperties.singleSelection().observe(someViewer)
```

Weitere Informationen

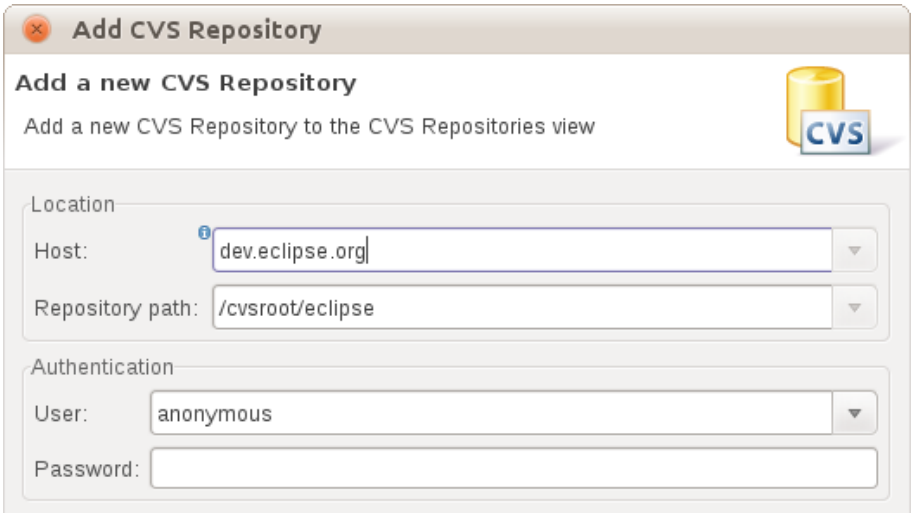
- JFace Data Binding
  http://wiki.eclipse.org/index.php/JFace_Data_Binding
- org.eclipse.jface.examples.databinding Snippets
  http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.jface.examples.databinding/src/org/eclipse/jface/examples/databinding/snippets/
- Eclipse DataBinding für die Kommunikation zwischen Modell und GUI
  http://it-republik.de/jaxenter/artikel/Eclipse-DataBinding-fuer-die-Kommunikation-zwischen-Modell-und-GUI-1353.html
- Eclipse JFace Databinding - Tutorial
  http://www.vogella.de/articles/EclipseDataBinding/article.html
- Tip: Validation with a MultiValidator
  http://eclipsesource.com/blogs/2009/02/27/databinding-crossvalidation-with-a-multivalidator/
- Databinding: A Custom Observable for a Widget
  http://eclipsesource.com/blogs/2009/02/03/databinding-a-custom-observable-for-your-widget/
- Eclipse Databinding + Validation + Decoration
  http://www.toedter.com/blog/?p=36
- Binden von JFace-Viewern: Snippet017TableViewerWithDerivedColumns
  http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.jface.examples.databinding/src/org/eclipse/jface/examples/databinding/snippets/Snippet017TableViewerWithDerivedColumns.java?view=markup

TUTORIAL 22.1

JFace Data Binding Snippets auschecken

In dem Projekt *org.eclipse.jface.examples.databinding* finden sich viele Beispiel-Snippets zur korrekten Verwendung von *JFace Data Binding*. Sie können das Projekt mit folgenden Schritten aus dem Eclipse CVS-Repository auschecken:

- ☐ Wechseln Sie in die *CVS Repository Exploring*-Perspektive.
- ☐ Legen Sie ein neues *Repository :pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse* an:



- ☐ Checken Sie unter *HEAD* das Projekt *org.eclipse.jface.examples.databinding* aus.
- ☐ Führen Sie *org.eclipse.jface.examples.databinding.snippets.Snippet000HelloWorld* mit *Run as › Java application* aus.

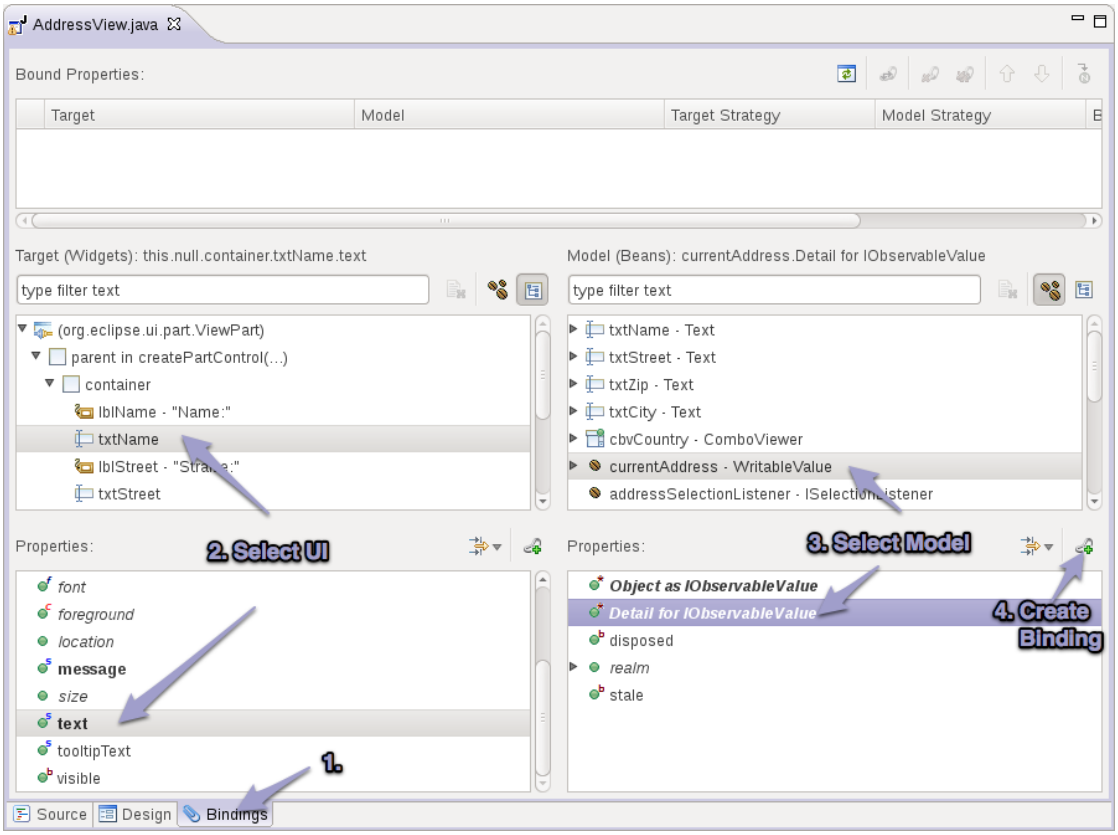
TUTORIAL 22.2

Adressmaske binden

- ☐ Aktivieren Sie in den Eclipse Preferences unter *WindowBuilder > SWT > Code Generation > JFace Data Binding* die Option *Generate observables code for version 1.3 (over properties)*.
- ☐ Fügen Sie dem Plug-in *com.example.addressbook* folgende Abhängigkeiten hinzu, um JFace Data Binding zu verwenden:
  - ☐ *org.eclipse.core.databinding*
  - ☐ *org.eclipse.core.databinding.beans*
  - ☐ *org.eclipse.core.databinding.property*
  - ☐ *org.eclipse.jface.databinding*.
- ☐ Deklarieren Sie in der Adressmasken-View eine Instanzvariable *currentAddress* vom Typ *WritableValue*. Diese wird die jeweils ausgewählte Adresse beinhalten. Ändern Sie die Methode *setAddress* so ab, dass *currentAddress* gesetzt wird, statt wie bisher die Widgets zu befüllen:

```
public class AddressViewPart extends ViewPart {

    private WritableValue currentAddress = new WritableValue();

    /* ... */

    protected void setAddress(Address address) {
        this.currentAddress.setValue(address);
    }

}
```

☐ Binden Sie im WindowBuilder alle Steuerelemente an die jeweiligen Eigenschaften von *currentAddress*:



☐ Führen Sie die UI-Tests aus.

KAPITEL 23

OSGi Services

OSGi Bundles können *Service* publizieren. Services sind normale Java-Objekte, die unter einem Interface in der *OSGi Service-Registry* angemeldet werden. Andere Bundles können nach diesen Services suchen und sie verwenden. Auch hier kommt die dynamische Natur von OSGi zum Tragen - Services können jederzeit kommen und gehen, z.B. wenn das bereitstellende Bundle gestoppt wird. Konsequenterweise bedeutet dies, dass verwendende Bundles auf diese Ereignisse reagieren müssen.

Im Eclipse RCP-Framework selbst spielen OSGi Services nur eine untergeordnete Rolle, das Framework macht eher selten Gebrauch von diesem Mechanismus. OSGi Services können jedoch problemlos in eigenen Bundles zur Implementierung von Anwendungsfunktionalität verwendet werden.

Publizieren von Services

Das Publizieren von Services geschieht in der Regel beim Starten des Bundles im *BundleActivator*. Dabei wird für den Klassennamen eines Java-Interfaces die entsprechende Implementierung registriert:

```
public class Activator implements BundleActivator {

    public void start(BundleContext context) throws Exception {
        context.registerService(ISomeService.class.getName(), new SomeServiceImpl(), null);
    }

    // ...

}
```

Verwendung von Services

Das Abfragen von Services erfolgt wiederum über den *BundleContext*:

```
ServiceReference serviceReference = bundleContext.getServiceReference(ISomeService.class.getName());
ISomeService service = (ISomeService) bundleContext.getService(serviceReference);
```

OSGi zählt die Verwendung von Services. Daher ist es wichtig, den Service wieder freizugeben, sobald er nicht mehr benötigt wird:

```
bundleContext.ungetService(serviceReference);
```

Eine solche Verwendung von Services wird der Dynamik des OSGi-Programmiermodells jedoch nicht gerecht. Denn Services können jederzeit kommen und gehen und Bundles sollten auf diese Ereignisse reagieren. Das Verfolgen von Services kann über *ServiceListener* erfolgen.

Meist wird jedoch ein *ServiceTracker* verwendet, bei dem das Auftauchen und Verschwinden von Services durch die Methoden *addingService* und *removedService* behandelt wird:

```
ServiceTracker tracker = new ServiceTracker(bundleContext,
        ISomeService.class.getName(), null) {

    public Object addingService(ServiceReference reference) {
        ISomeService service = (ISomeService) super.addingService(reference);
        // Handle occuring service here
        return service;
    }

    public void removedService(ServiceReference reference, Object service) {
        // Handle removed service here
        super.removedService(reference, service);
    }

};

tracker.open();
```

Dependency Injection

Die manuelle Verwendung von Services über *getService* bzw. *ServiceTracker* ist nur in wenigen Fällen sinnvoll. Es empfiehlt sich, einen Container zu verwenden, der Services automatisch mit ihren abhängigen Services versorgt (*Dependency Injection*). Services selbst sollten unabhängig von ihrer Laufzeitumgebung sein, d.h. ihre Abhängigkeiten nicht selbst auflösen, sondern davon ausgehen, das diese von der Laufzeitumgebung bereitgestellt werden (*Inversion of Control*). Dazu gibt es verschiedene Ansätze:

> *e4*
> *OSGi Declarative Services*

- *Peaberry*: Extension library for Google Guice that supports dependency injection of dynamic services
- *Spring Dynamic Modules* (DM) bzw. *OSGi Blueprint*
- *Extensions2Services*: Integration layer for Eclipse Extensions to use OSGi Services
- *Riena Core*: Dependency Injection für Services und Extensions

Weitere Informationen

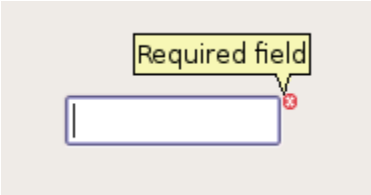
- OSGi Service Platform Core Specification, Chapter 5: Service Layer
- OSGi Service Platform Service Compendium, 112 Declarative Services Specification, 701 Service Tracker Specification
- Die OSGi Service Plattform, Kapitel 6: OSGi Services, Kapitel 7: Umgang mit dynamischen Services sowie Kapitel 12: Declarative Services

KAPITEL 24

Rezepte

SWT: Dekoration von Controls

Mit der JFace-Klasse *ControlDecoration* können SWT-Controls mit einem Symbol versehen werden:



Die Dekoration erfolgt dabei um das eigentliche Control herum und ist unabhängig vom jeweiligen Layout. Man sollte lediglich sicherstellen, dass genügend Platz zur Anzeige des Symbols zur Verfügung steht. Erzeugt wird eine Dekoration mit:

```
ControlDecoration decoration = new ControlDecoration(control, SWT.RIGHT | SWT.TOP);
```

Standard-Symbole für Fehler oder Warnungen werden von *FieldDecorationRegistry* zur Verfügung gestellt:

```
Image errorImage = FieldDecorationRegistry.getDefault()
        .getFieldDecoration(FieldDecorationRegistry.DEC_ERROR).getImage();
decoration.setImage(errorImage);
```

Folgende Konstanten können verwendet werden:

- *DEC_CONTENT_PROPOSAL*
- *DEC_REQUIRED*
- *DEC_ERROR*
- *DEC_WARNING*
- *DEC_INFORMATION*
- *DEC_ERROR_QUICKFIX*

Optional kann ein beschreibender Text eingeblendet werden, sobald der Benutzer mit der Maus über das Symbol fährt:

```
decoration.setDescriptionText("Required field");
decoration.setShowHover(true);
```

Mit den Methoden *show* und *hide* kann die Dekoration ein- oder ausgeblendet werden:

```
decoration.show();
decoration.hide();
```

JFace: Eingabe-Vervollständigung mit FieldAssist

JFace stellt eine komfortable Möglichkeit bereit, Eingaben in Textfeldern automatisch zu vervollständigen:

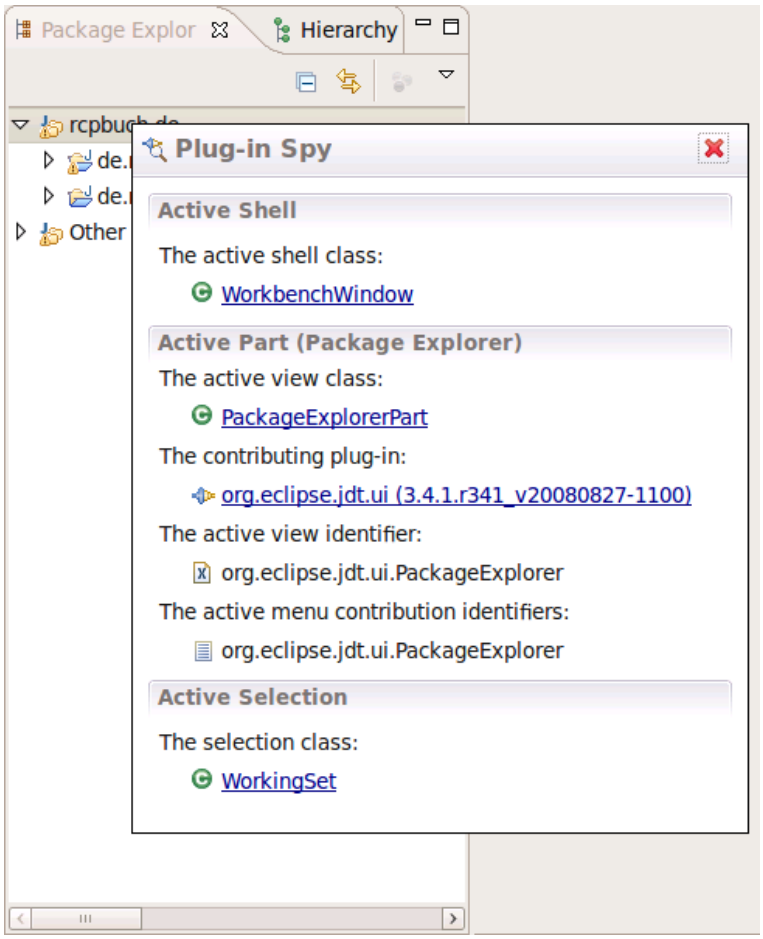
Um die Vervollständigung zu aktivieren, ist eine Instanz von *AutoCompleteField* mit einem bestehendem Text-Feld und einer Liste aller Vorschläge zu erzeugen:

```
String[] cities = new String[] { "Aachen", "Berlin", "Bremen", "Bochum" };
new AutoCompleteField(textControl, new TextContentAdapter(), cities);
```

Im obigen Beispiel wurde zusätzlich eine *ControlDecoration* mit *DEC_CONTENT_PROPOSAL* für das Textfeld erzeugt, um den Anwender auf die Möglichkeit der Eingabevervollständigung hinzuweisen.

Plug-in Spy verwenden, um Eclipse-Komponenten aufzuspüren

Der Eclipse Plug-in Spy erlaubt Ihnen, die Klasse und das Plug-in zu UI-Komponenten in Eclipse zu lokalisieren. Betätigen Sie dazu die Tastenkombination “Alt+Shift+F1” und der Plug-in Spy erscheint mit allen Informationen zum aktuell geöffneten Element:



Um den Plugin-Spy für RCP-Anwendungen zu aktivieren, starten Sie die Anwendung mit dem Plug-in *org.eclipse.pde.runtime*.

Commands programmatisch ausführen

Möchten Sie einen Command programmatisch ausführen, sollten sie den Handler nicht direkt instantiieren, sondern den *HandlerService* dazu verwenden, um den aktuell gültigen Handler für den Command auszuführen:

```
IHandlerService handlerService = (IHandlerService) PlatformUI.getWorkbench()
    .getService(IHandlerService.class);
handlerService.executeCommand("com.example.somecommand", null);
```

Commands: Expressions erweitern mit Property-Testern

Ein *Property Tester* erlaubt es, die Ausdrücke zur Evaluierung von Expressions zu ergänzen. Diese können beispielsweise bei der Aktivierung von Handlern für die Selektion verwendet werden, um Werte des selektierten Objekts abzufragen. Nehmen wir an, wir möchten einen Handler nur dann aktivieren, wenn es sich bei der ausgewählten Adresse um eine Firma handelt.

Dazu müssen wir unserem Plug-in zunächst eine Abhängigkeit zum Plug-in *org.eclipse.core.expressions* hinzufügen, da dieses den Extension Point zur Deklaration eigener *Property Tester* bereitstellt:

```
Require-Bundle: [...],
 org.eclipse.core.expressions
```

Nun können wir einen *Property Tester* deklarieren. Ein *Property Tester* bekommt eine eindeutige ID, eine implementierende Klasse und ist gültig für den angegebenen Objekttyp *type*. Der Name des Properties setzt sich aus einem Namespace und dem Namen zusammen, im folgenden Beispiel "com.example.addressbook.isCompany":

```
<extension point="org.eclipse.core.expressions.propertyTesters">
   <propertyTester
         id="com.example.addressbook.AddressPropertyTester"
         class="com.example.addressbook.AddressPropertyTester"
         type="com.example.addressbook.entities.Address"
         namespace="com.example.addressbook"
         properties="isCompany"/>
</extension>
```

Die Implementierung hat zur Evaluierung des Properties das *receiver*-Objekt (wie im *type*-Attribut deklariert) und den Namen des Properties zur Verfügung. Zusätzlich kann der Nutzer des Properties eine Liste von Argumenten und einen erwarteten Wert angeben, die dem PropertyTester übergeben werden:

```
public class AddressPropertyTester extends PropertyTester {

    @Override
    public boolean test(Object receiver, String property, Object[] args, Object expectedValue) {
        return ((Address) receiver).isCompany();
    }

}
```

Ein solcher Property-Tester kann nun in *test*-Elementen verwendet werden:

```
<test property="com.example.addressbook.isCompany" forcePluginActivation="true"/>
```

Möchten wir beispielsweise einen Menüeintrag nur für Firmen aktivieren, könnten wir dies folgendermaßen erreichen:

```
<extension point="org.eclipse.ui.handlers">
    <handler class="com.example.SomeHandler" commandId="com.example.SomeCommand">
        <activeWhen>
            <with variable="selection">
                <iterate ifEmpty="false" operator="or">
                    <and>
                        <instanceof value="com.example.addressbook.entities.Address"/>
                        <test property="com.example.addressbook.isCompany"/>
                    </and>
                </iterate>
            </with>
        </activeWhen>
    </handler>
</extension>
```

Commands: Expressions mit Source Providern um eigene Variablen erweitern

Mit dem Extension Point *org.eclipse.ui.services* können Sie eigene Variablen zur Verwendung in *with*-Elementen bei der Aktivierung von Commands und Command-Handlern deklarieren. Dazu deklarieren Sie einen *SourceProvider* und spezifizieren die bereitgestellten Variablen. Da die Evaluierung von Variablen nach Prioritäten gewichtet wird (eine Prüfung auf das aktive View mittels *activePartId* ist beispielsweise höher priorisiert als die Prüfung auf die geöffnete Perspektive), müssen Sie mittels *priorityLevel* angeben, wie Ihre eigene Variable priorisiert wird:

```
<extension point="org.eclipse.ui.services">
   <sourceProvider provider="com.example.SomeSourceProvider">
      <variable name="com.example.someVariable" priorityLevel="activePartId"/>
   </sourceProvider>
</extension>
```

Den *SourceProvider* leiten Sie von *AbstractSourceProvider* ab und spezifizieren, welche Variablen bereitgestellt werden und wie diese belegt sind:

```
public class SomeSourceProvider extends AbstractSourceProvider {
    private static final String SOME_VAR = "com.example.someVariable";

    public Map getCurrentState() {
        HashMap<String, String> values = new HashMap<String, String>();
        values.put(SOME_VAR, "someValue");
        return values;
    }

    public String[] getProvidedSourceNames() {
        return new String[] { SOME_VAR };
    }
}
```

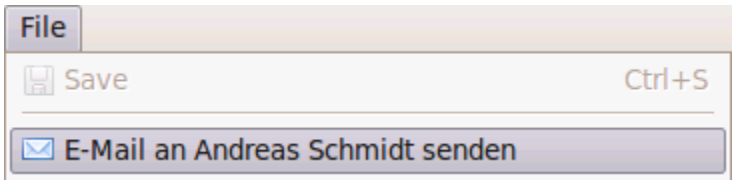
Dabei ist zu beachten, dass *fireSourceChanged* immer dann aufgerufen werden muss, wenn sich der Wert einer Variable verändert hat, da die Eclipse Plattform die Variablen nur bei Bedarf evaluiert.

Eine so deklarierte Variable können Sie in Expressions mit einem *with*-Element abfragen:

```
<extension point="org.eclipse.ui.handlers">
   <handler class="com.example.SomeHandler" commandId="com.example.someCommand">
      <activeWhen>
         <with variable="com.example.someVariable">
            <equals value="someValue"/>
```

Commands: Dynamisches aktualisieren mit IElementUpdater

Es besteht die Möglichkeit, dass ein Handler zugehörige UI-Elemente wie Menüeinträge oder Toolbar-Buttons verändert. Nehmen wir an, Sie möchten den Text eines Menüeintrages entsprechend der aktuellen Selektion ergänzen:



Lassen Sie dazu Ihren Handler das Interface *IElementUpdater* implementieren. In der *updateElement*-Methode können Sie nun das UI-Element manipulieren:

```
public class SendEmailHandler extends AbstractHandler implements IElementUpdater {

    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        //...
    }

    @Override
    public void updateElement(UIElement element, Map parameters) {
        String text = "E-Mail senden";
        ISelection selection = PlatformUI.getWorkbench().getActiveWorkbenchWindow()
                .getSelectionService().getSelection();
        if (!selection.isEmpty() && selection instanceof IStructuredSelection) {
            Object selectedObject = ((IStructuredSelection) selection).getFirstElement();
            if (selectedObject instanceof Address) {
                Address address = ((Address) selectedObject);
                text = "E-Mail an " + address.getName() + " senden";
            }
        }
        element.setText(text);
    }

}
```

Dem *UIElement* können Sie folgende Eigenschaften setzen:

- *text, tooltip*
- *icon, disabledIcon, hoverIcon*
- *checked*

> *dropDownId* (erlaubt den Austausch der Einträge in einem Drop-Down-Menü)

Der Element-Updater wird automatisch ausgeführt, sobald sich die Umgebung verändert. Im obigen Beispiel wird immer, wenn sich die Selektion ändert, auch *SendEmailHandler.updateElement* ausgeführt und der Menüeintrag aktualisiert. Möchten Sie auf Ereignisse außerhalb der Kontrolle von Eclipse reagieren, können Sie programmatisch eine Aktualisierung über den *CommandService* auslösen:

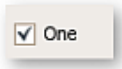
```
ICommandService commandService = (ICommandService) PlatformUI.getWorkbench().getService(
    ICommandService.class);
commandService.refreshElements("com.example.addressbook.email", null);
```

KAPITEL 25

Anhang: Übersicht SWT Widgets

Browser

Button (SWT.ARROW)

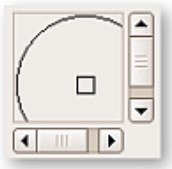


Button (SWT.CHECK)

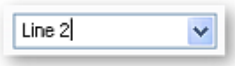
Button (SWT.PUSH)

Button (SWT.RADIO)

Button (SWT.TOGGLE)



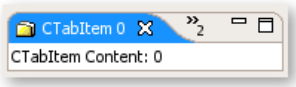
Canvas



Combo

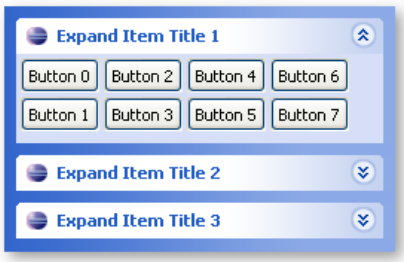
Composite

CoolBar



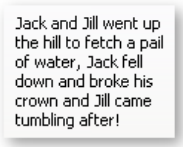
CTabFolder

DateTime



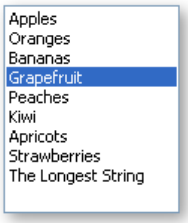
ExpandBar

Group

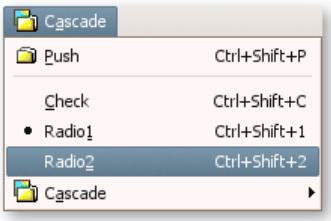


Label

Link

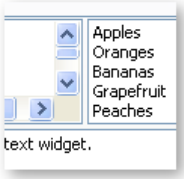


List

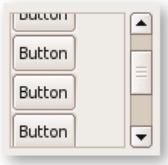


Menu

ProgressBar



Sash

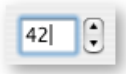


ScrolledComposite

Shell

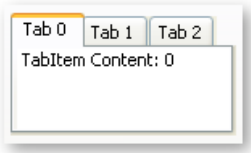
Slider

Scale

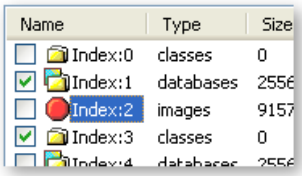


Spinner

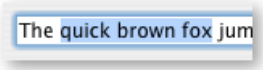
StyledText



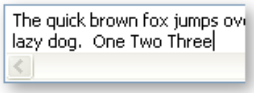
TabFolder



Table

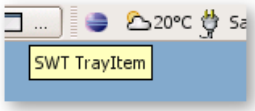


Text (`SWT.SINGLE`)

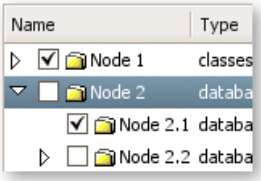


Text (`SWT.MULTI`)

ToolBar



Tray



Tree

KAPITEL 26

Anhang: p2

Making an application updateable

> Create the example project: Create a new plug-in project *com.example.mail*. Choose Eclipse version 3.7 and create a rich client application from the *RCP Mail Template*:

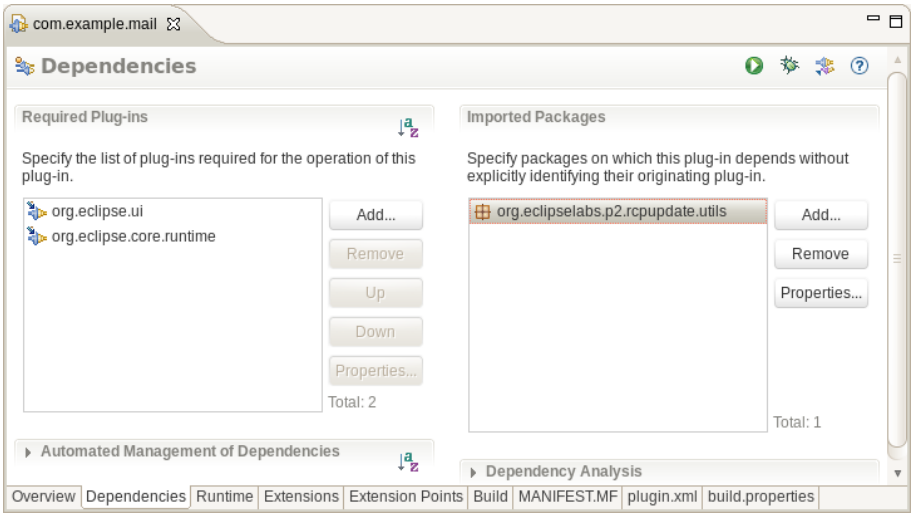
> Download org.eclipselabs.p2.rcpupdate. This project contains a plug-in and a feature to make RCP applications updateable via p2. It's mostly extracted from the Eclipse Wiki: Adding Self-Update to an RCP Application and the RCP Cloud example project from Susan F. McCourt. There are plans to include something like this in Eclipse 3.7, see Bug 281226 - RCP Simple Update UI for more information about this.

> The archive contains two projects, import both into your workspace using *File* › *Import* › *Existing Projects into Workspace* › *Select archive file*.

> Create a new menu contribution in the *plugin.xml* of the *com.example.mail* plug-in and add both the commands *org.eclipselabs.p2.rcpupdate.install* and *org.eclipselabs.p2.rcpupdate.update* to the help menu:

```
<extension point="org.eclipse.ui.menus">
    <menuContribution locationURI="menu:help">
        <command commandId="org.eclipselabs.p2.rcpupdate.install" style="push"/>
        <command commandId="org.eclipselabs.p2.rcpupdate.update" style="push"/>
    </menuContribution>
</extension>
```

> Add a package import to *org.eclipselabs.p2.rcpupdate.utils* to the *com.example.mail* plug-in:

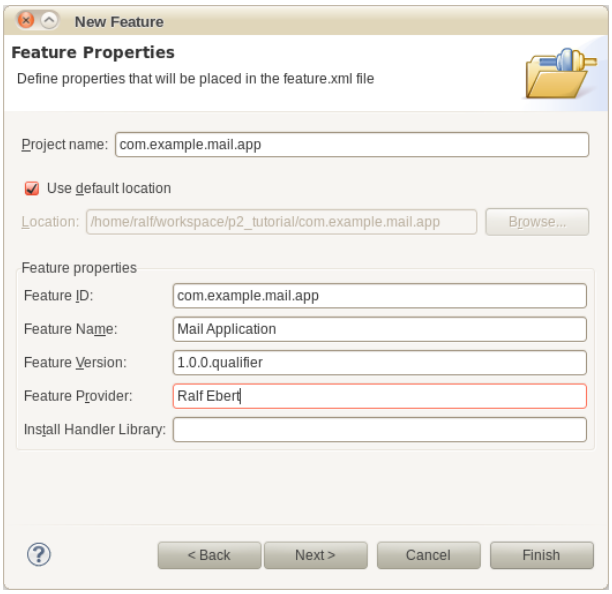


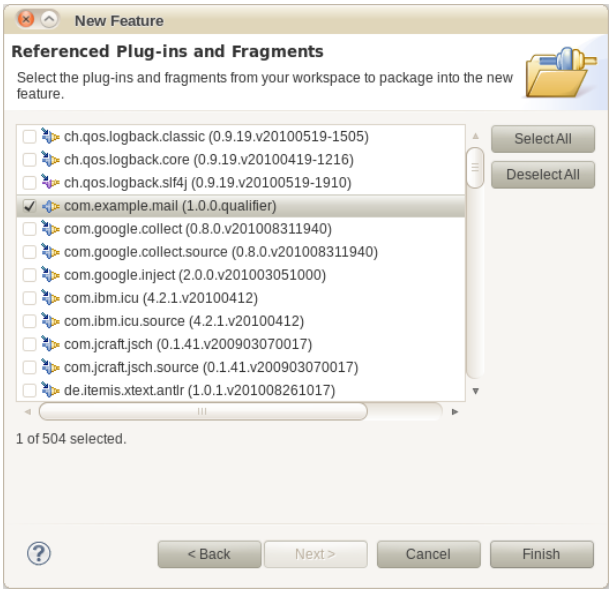
> Include the update check on application startup by calling *P2Util.checkForUpdates()* in your *ApplicationWorkbenchAdvisor* class:

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {

    @Override
    public void preStartup() {
        P2Util.checkForUpdates();
    }

}
```

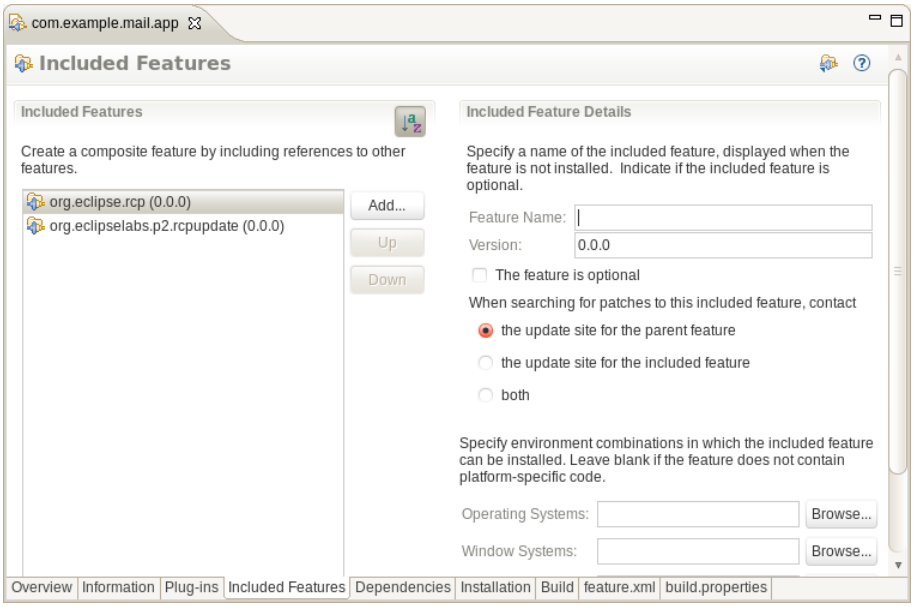
Feature / product setup

> p2 will only install features, so create a new feature *com.example.mail.app* for the mail application. This feature will contain all the main application plug-ins, so add the plug-in *com.example.mail* to this feature:

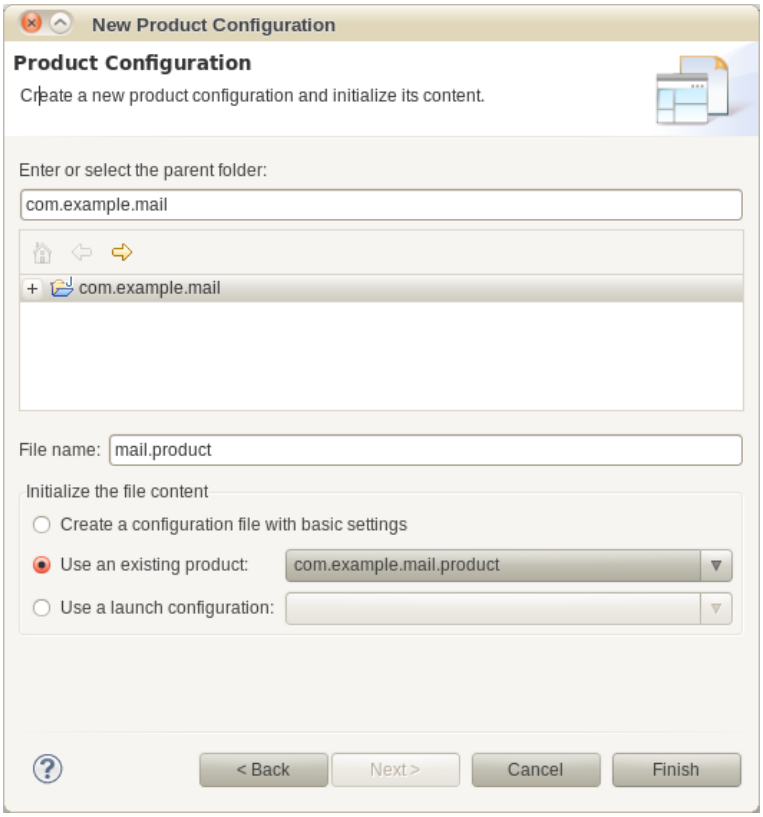




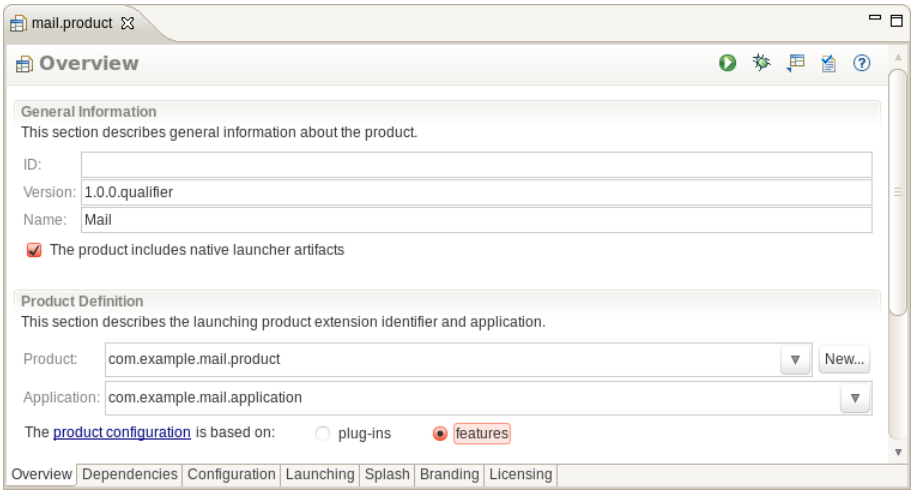
> Add *org.eclipse.rcp* and *org.eclipselabs.p2.rcpupdate* to the list of included features:



> Create a new product named *mail.product* using *File › New › Plug-in Development › Product Configuration*:



> Open the product. Check that it has a version number like '1.0.0.qualifier' and that the ID is empty (an ID will be generated automatically. Please note: If you enter an ID here, make sure it's different from the product ID!). Change the product to be based on features:



> Add the feature *com.example.mail.app* to the product dependencies and delete the version number (this means it should use the newest version available - this is new in Eclipse 3.6, see Bug 279465 - no feature version should imply "0.0.0"):

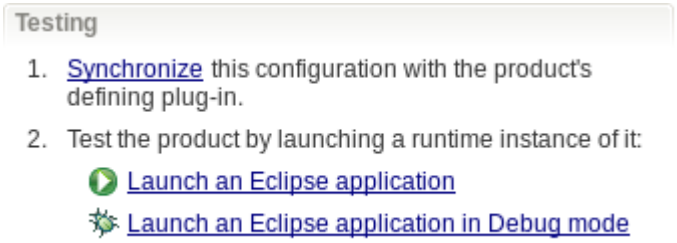
> Note: We will export the application in a minute. I'll refer to the folder we will export to as *export/*. The export will create two subfolders, *export/repository/* containing a p2 repository for updating, and one folder with the application. We need to configure p2 to get the application updates from the repository, so you need to plan ahead the location of the repository folder. If you have a HTTP server somewhere, you can upload the repository and use a *http://* URL; a local *file://* URL will work as well.

> Create a *p2.inf* file in the plug-in project that contains the product and configure either a file:// **or** a http://-repository. Please make sure that you get the syntax right or copy from http://gist.github.com/551240):

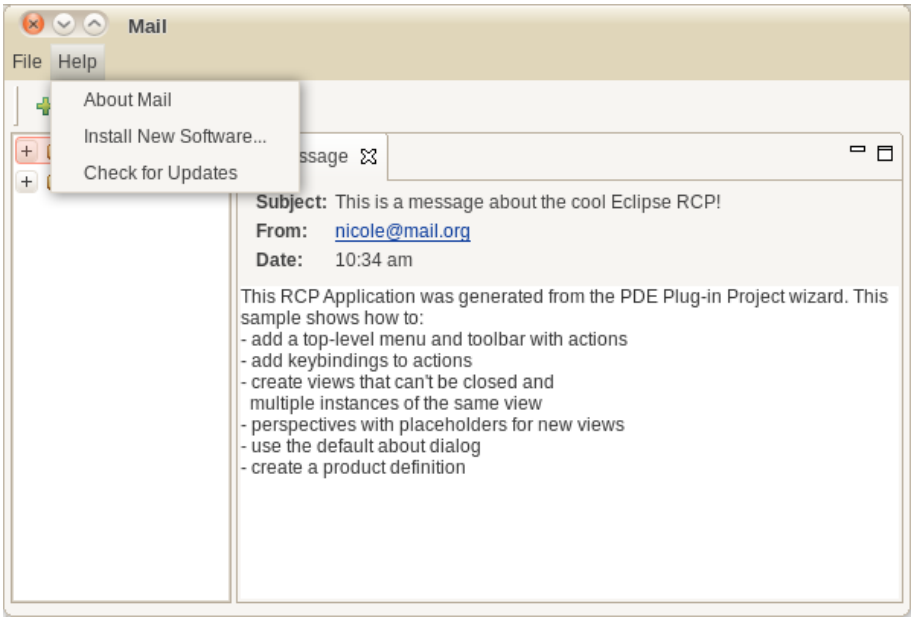
```
instructions.configure=\
  addRepository(type:0,location:file${#58}/c:/export/repository/);\
  addRepository(type:1,location:file${#58}/c:/export/repository/);

instructions.configure=\
  addRepository(type:0,location:http${#58}//localhost:1234/repository/);\
  addRepository(type:1,location:http${#58}//localhost:1234/repository/);
```

(*type* configures the repository type, 0 is a metadata repository, 1 is an artifact repository - you need to configure both. *${#58}* in the URL expresses a colon. You can read more about the p2 metadata instructions here: p2 Touchpoint Instructions)

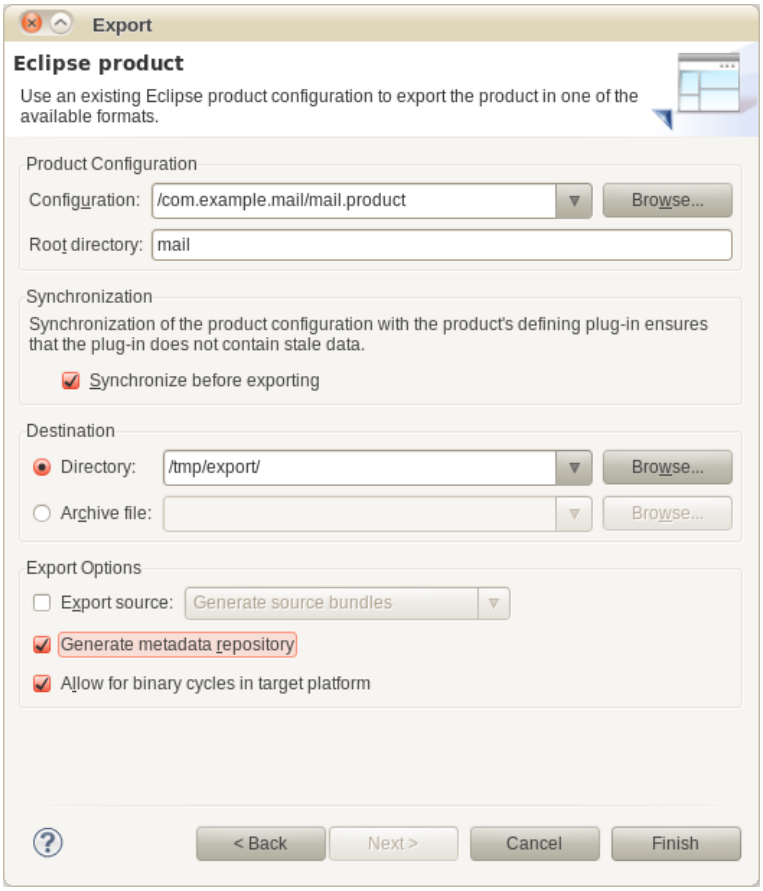
> Launch the product from the product configuration. If you have started the product already, delete the existing run configuration before launching the product again (the changed dependencies are applied to the run configuration **only** when a new run configuration is created):



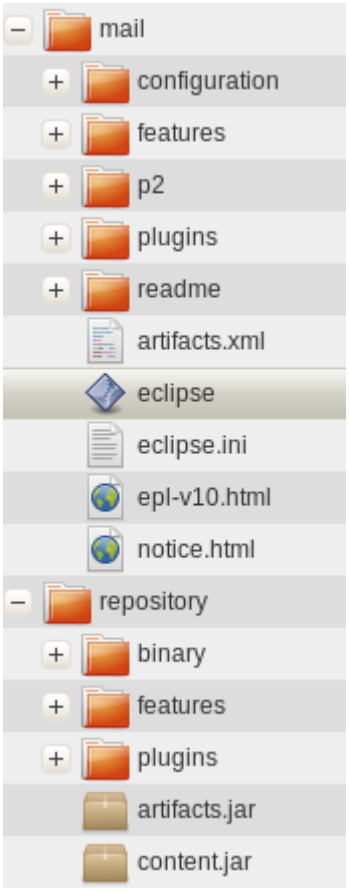
> Check that the application launches and that the menu items are shown correctly (it's normal that they don't work yet, as applications are not managed by p2 when started using a run configuration from Eclipse - by the way, since Eclipse 3.6 there is an option for that, see *Run Configuration › Configuration › Software Installation › Support software installation in the launched application*).



> Export the product using *File* › *Export* › *Eclipse product* to your *export/* folder. Make sure to check *Generate metadata repository*:



> Have a look at the generated *export* folder - you should find a *repository* folder and a folder *mail*. Move *mail* to *mail_user* to simulate installing the app at the user's computer (the folder will cause a conflict upon re-exporting if you don't rename it because the export will only update the repository, but not the product). Launch the exported application:



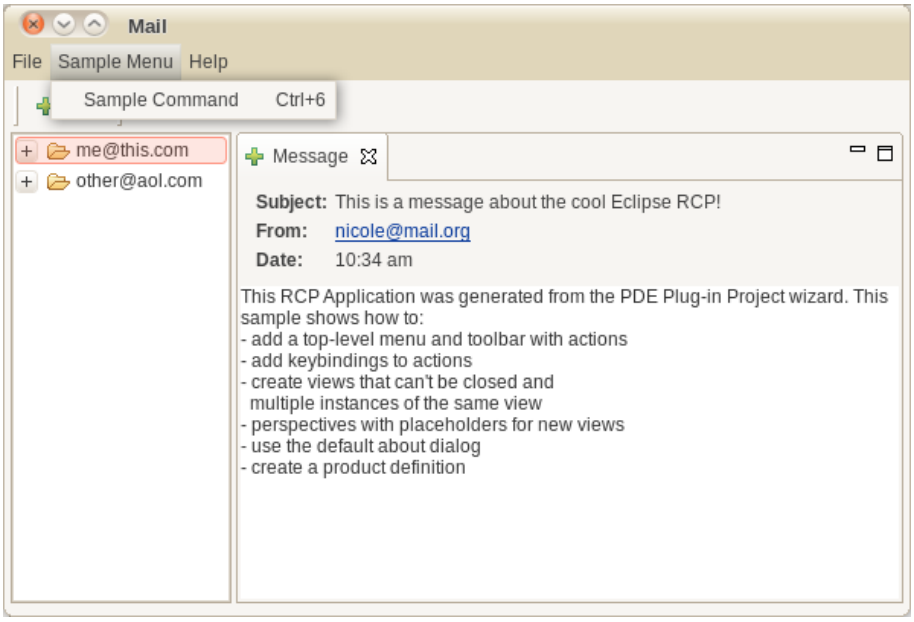
Installing additions using p2

Lets try to install additions to the Mail application.

As an example, let's assume we want to supply users of the mail application with some e-mail protection and safety features (like a spam filter or phising protection). Lets assume we decided that users of the mail application can install these features additionally, they are not included in the default distribution of the Mail application or are even provided by a 3rd party.

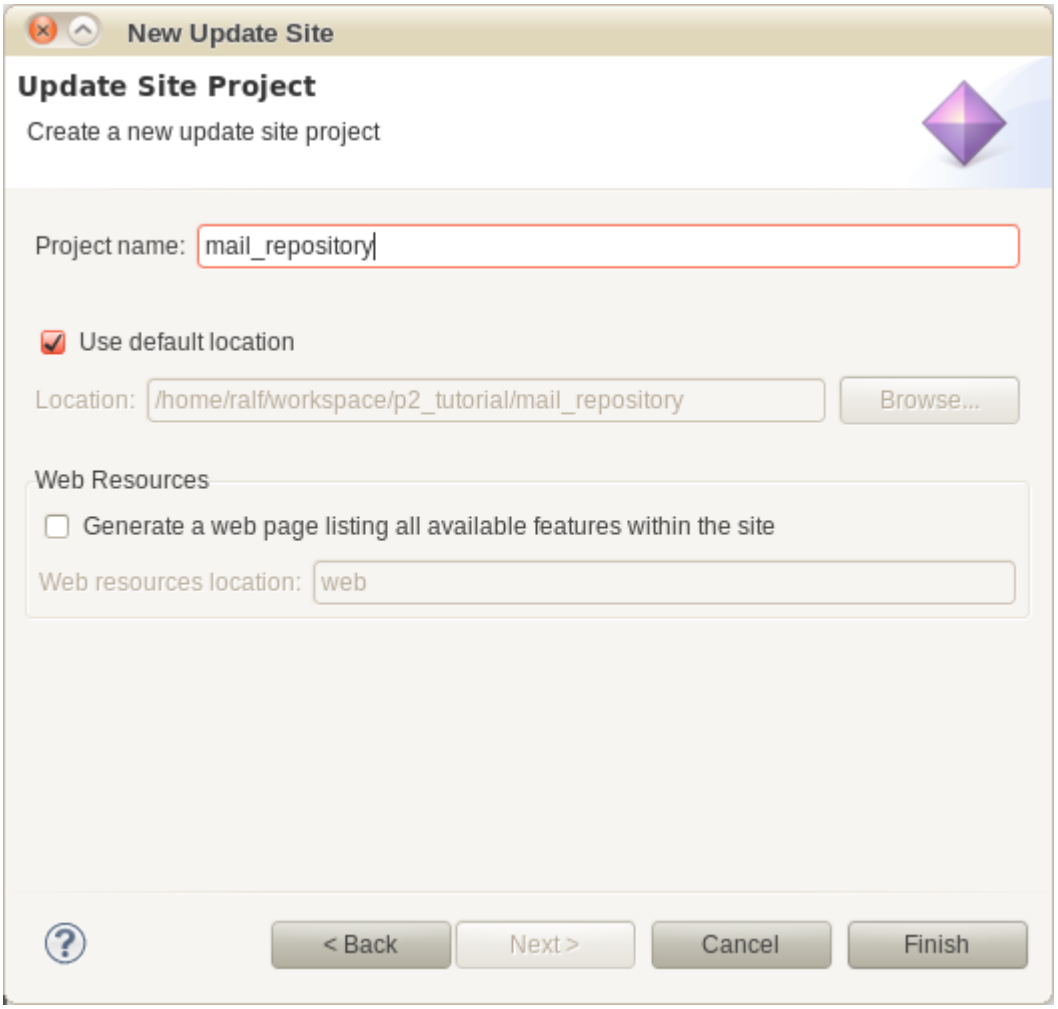
> Create a new plug-in project *com.example.mail.protection.spamfilter*. Don't create a rich client application and choose the template *Hello, World Command* to create a plug-in that adds a single command to the menu and toolbar of the application.

> Add the spam filter plug-in to your run configuration and check that the UI contribution works as expected:



> The p2 update UI only works for installing and updating features. So we have to create a feature that contains the additional plug-in. Create a new feature *com.example.mail.protection* and add the spamfilter plug-in to the feature.

> Create a new update site project *mail_repository*. Optionally click *generate a web page listing*:

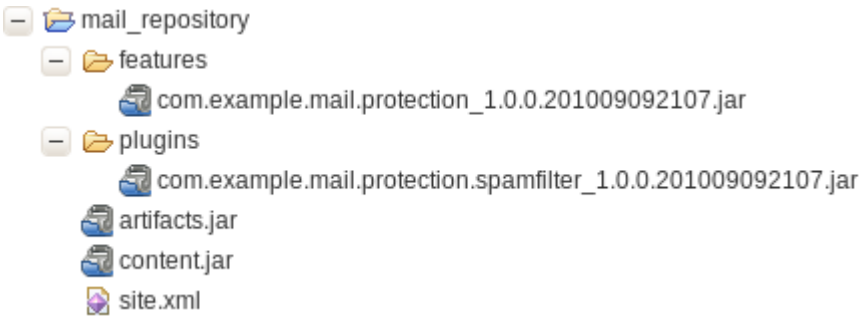


> Create a new category *Mail Additions* and add the *com.example.mail.protection* feature to it:

Note: Features to be installed by the user need to have a category. By default, the p2 UI shows only features that have a category assigned. Features without a category are meant to be technical and not to be installed by the user directly (like the RCP feature). If we don't create a category for the repository, the user would have to uncheck *Group items by category* in the update dialog to see the feature.

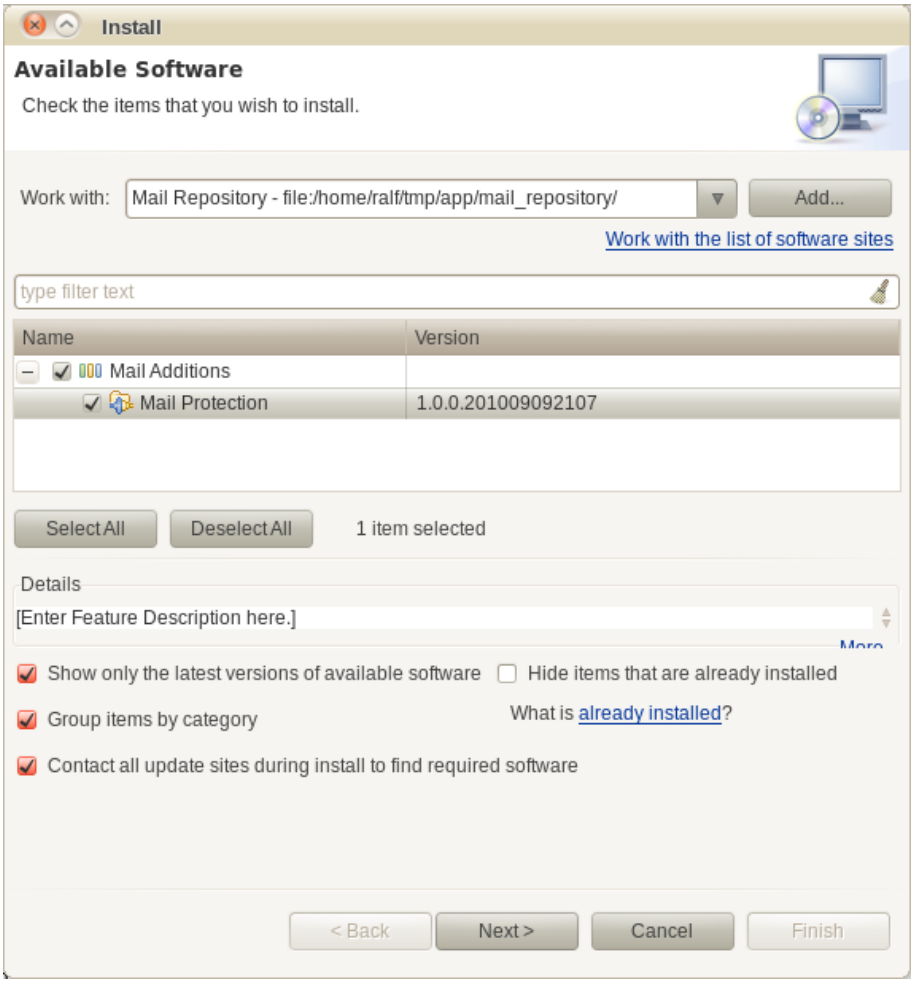
> Click *Build All* to build the feature and add the feature and plug-in to the repository (by the way, we could also use *Export › Deployable features* to do the export, but the *update site project* is a bit more convenient).

> Have a look at the files in the *mail_repository*. This is a *p2 repository/p2 software site* that can be used to install updates and new features. It contains the feature and plug-in jars and metadata about the repository in *artifacts.jar* and *content.jar*:

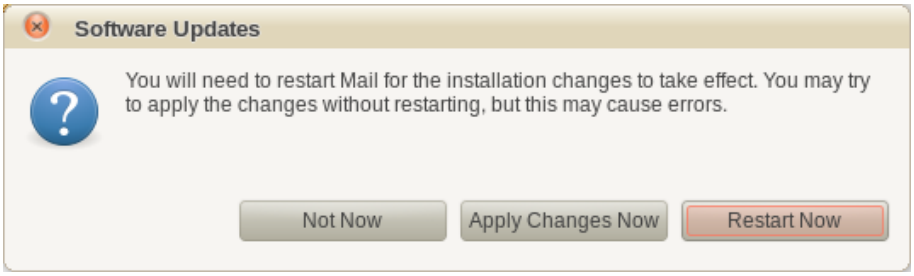


> Optional: upload the repository files to some HTTP server so that they are reachable using a *http://* URL.

> Start the mail application from *mail_user*. Choose *Help › Install New Software...* and enter the repository URL or choose a local file using *Add...*:



> Install the *Mail Protection* feature. You will be asked about installing unsigned bundles and about restarting the application:



Depending on the installed features a restart might be necessary or not - p2 cannot know. If you just added a menu item, you can count on the dynamic nature of the RCP workbench and just go without a restart. You will see the added menu items immediately.

p2 concepts

What just happened is that you used the p2 UI to install additions from a repository. These additions (or more general, everything that can be installed using p2), is called an *installable unit (IU)*. These installable units are contained in so called *metadata repositories* which refer to actual files in an *artifact repository*.

Have a look into the metadata files *artifacts.jar* and *content.jar* to learn about the general nature of p2 repositories. *artifacts.xml* in *artifact.jar* is just a list of all the files (*artifacts*) in the repository and metadata like file size, for example:

```
<artifact classifier='osgi.plug-in' id='com.example.mail' version='1.0.1'>
    <properties size='2'>
        <property name='artifact.size' value='4096'/>
        <property name='download.size' value='103133'/>
    </properties>
</artifact>
```

content.xml in *content.jar* is a list of *installable units* with their name, capabilities and a description of what the unit provides and requires. This is a very general description, as p2 is meant to be able to provision everything, not only features or plug-ins. For example:

```
<unit id='com.example.mail.spamfilter' version='1.0.1'>
    <update id='com.example.mail.spamfilter' range='[0.0.0,1.0.1)' severity='0'/>
    <properties size='2'>
        <property name='org.eclipse.equinox.p2.name' value='Spam Filter'/>
        <property name='org.eclipse.equinox.p2.provider' value='Ralf Ebert'/>
    </properties>
```

```
    <provides size='3'>
        <provided namespace='o.e.equinox.p2.iu' name='com.example.mail.spamfilter' version='1.0.1'/>
        <provided namespace='osgi.plug-in' name='com.example.mail.spamfilter' version='1.0.1'/>
        <provided namespace='o.e.equinox.p2.eclipse.type' name='plug-in' version='1.0.0'/>
    </provides>
    <requires size='2'>
        <required namespace='osgi.plug-in' name='org.eclipse.ui' range='0.0.0'/>
        <required namespace='osgi.plug-in' name='org.eclipse.core.runtime' range='0.0.0'/>
    </requires>
    <artifacts size='1'>
        <artifact classifier='osgi.plug-in' id='com.example.mail.spamfilter' version='1.0.1'/>
    </artifacts>
    <touchpoint id='org.eclipse.equinox.p2.osgi' version='1.0.0'/>
        <touchpointData size='1'>
            <instructions size='1'>
                <instruction key='manifest'>
                    <!-- manifest omitted here -->
                </instruction>
            </instructions>
        </touchpointData>
    </touchpoint>
</unit>
```

The idea behind these metadata repositories is that p2 can reason (like resolving dependencies) without downloading the actual artifacts. So when PDE exports deployable features or plug-ins, all the plug-in dependencies are written down in the metadata repository in this very general format.

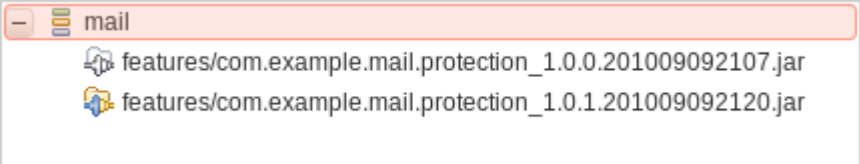
The installation of installable units is a fairly complex process that is conducted by several p2 components. There is a p2 planner component that reasons about the necessary steps to perform a p2 operation like an installation. These steps are carried out by the p2 engine. Planner and engine are directed by the *director* component. Mostly you can use p2 as a black box, but sometimes it's required to dig deeper into these concepts - have a look at the p2 concepts to learn more about the general p2 architecture.

This additional complexity yields some interesting features. For example, a p2 installation is capable to revert itself to previous installation states (see Chris Aniszczyks blog post for more information about this: Reverting Changes in an Eclipse Installation using p2)

Updating the feature

Let's assume we improved the spam filter plug-in to combat the latest developments from the spam industry and want to provide our users with an update:

- Do some visible change to the spam filter plug-in, like changing some text.
- Increment the version number of the plug-in and the feature. Hint: Have a look at the Eclipse version numbering scheme to learn how version numbers for Eclipse projects are handled.
- Add the feature again to the category of the update site project and click *Build*. This will **add** the new plug-in/feature to the existing repository (you could also delete the old one, but the general recommendation is to keep track of all published versions in one repository and never delete something that has already been published):



- Click *Help* › *Check for Update* in the application or restart the application to update to the newest version (you might have to restart because of p2 caching the metadata).

Updating the application

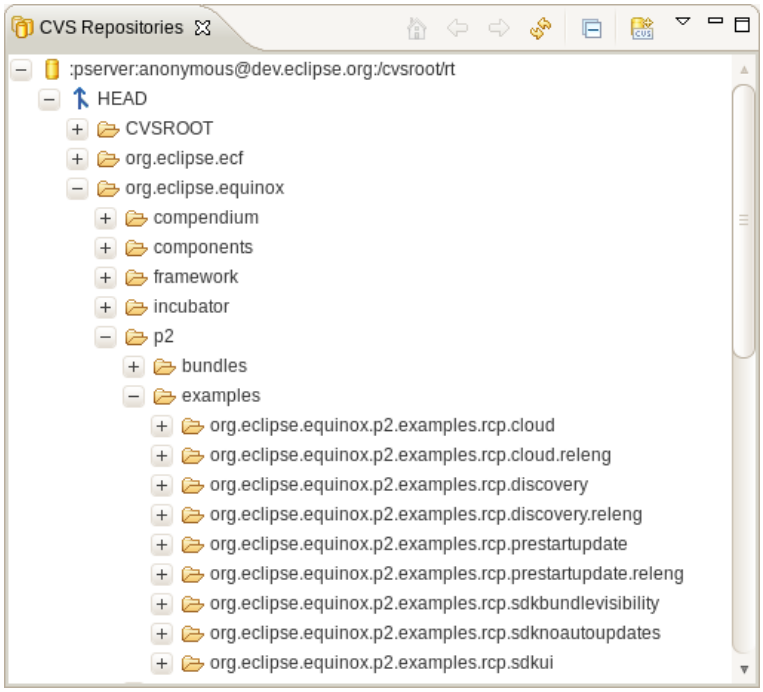
For now we just provided additions and updates for these additions. But how about the main application? Let's see how this can be updated using p2.

- Do some visible change in the mail plug-in so that you can check if the product was updated correctly (like adding some String to a label in the *View* class).
- Increment the version number of your plug-in, the feature and the product (for real products you should organize some strategy that makes sure that version numbers are incremented when doing releases).
- Warning: I told you to rename *export/mail* to *mail_user* before. This is important now. You should have *export/repository*, but not *export/mail*. Keeping the repository is ok, the export will update it, but if the mail application from the previous export would still exist there, you would get a conflict.
- Export the deployable product again. The repository will get updated with the new versions.

> Click *Help* › *Check for Update* in the application or restart the application to update to the newest version.

Checking out the examples

I highly recommend to have a look at the p2 examples. You can check them out from the Eclipse Runtime Project at *:pserver:anonymous@dev.eclipse.org:/cvsroot/rt*. Just paste the connect string into the CVS repository view and go to *org.eclipse.equinox/p2/examples*:



More information

- Eclipse Wiki: Adding Self-Update to an RCP Application
- *RCP Cloud Examples*

  ```
  CVS Repository :pserver:anonymous@dev.eclipse.org:/cvsroot/rt
  Project org.eclipse.equinox/p2/examples/org.eclipse.equinox.p2.examples.rcp.cloud
  ```

- Bug 281226: RCP Simple Update UI
- org.eclipselabs.p2.rcpupdate
- Eclipse Wiki: p2
- Eclipse Wiki: p2 concepts
- Reverting Changes in an Eclipse Installation using p2
- Equinox p2 cures Eclipse plug-in headaches
- Building p2 RCP products in Eclipse 3.5M6
- Example Headless build for a RCP product with p2

TUTORIAL 26.1

Anwendung updatefähig machen

- ☐ Starten Sie Ihr Produkt in Eclipse mit *Launch an Eclipse application* und exportieren Sie es, um sicherzustellen, dass das Produkt lauffähig ist.
- ☐ Laden Sie von *http://github.com/ralfebert/org.eclipselabs.p2.rcpupdate* das Plug-in und Feature für das Update von RCP-Anwendungen mit p2.
- ☐ Importieren Sie beide Projekte mit *File > Import > Existing Projects into Workspace* aus dem Archiv in Ihren Workspace.
- ☐ Fügen Sie die Commands *org.eclipse.ui.window.resetPerspective*, *org.eclipselabs.p2.rcpupdate.install* und *org.eclipselabs.p2.rcpupdate.update* in das Anwendungsmenü ein.
- ☐ Fügen Sie dem Manifest des Adressbuch-Plug-ins eine Abhängigkeit auf *org.eclipselabs.p2.rcpupdate.utils* hinzu.
- ☐ Überschreiben Sie die *preStartup*-Methode Ihrer WorkbenchAdvisor-Klasse und fügen Sie den Update-Check-Aufruf *P2Util.checkForUpdates()* ein.
- ☐ Entfernen Sie das Karten-Plug-in aus Ihrem Anwendungsfeature.
- ☐ Fügen Sie im Produkt unter *Dependencies* das Feature *org.eclipselabs.p2.rcpupdate* hinzu und entfernen Sie die Versionsnummern (keine Angabe bedeutet: neueste Version wird verwendet).
- ☐ Geben Sie im Produkt die Versionsnummer '1.0.0.qualifier' an (ID und Name sind optionale Metadaten für das p2-Repository. Achtung: Wenn eine ID vergeben wird, muss sich diese von der Produkt-ID unterscheiden!)
- ☐ Erstellen Sie einen neuen Ordner *export*. Hinweis: Der Export wird unter *export/addressbook* die Anwendung ablegen und unter *export/repository* ein p2-Repository, mit dem die Anwendung geupdatet werden kann.

- ☐ Fügen Sie dem Bundle, welches das Produkt enthält, eine Textdatei *p2.inf* hinzu und konfigurieren Sie folgendermaßen den Pfad zu dem *export/repository*-Ordner (siehe http://gist.github.com/551240. Wenn Sie einen Webserver zur Verfügung haben können Sie alternativ die *http://*-Variante verwenden):

```
instructions.configure=\
  addRepository(type:0,location:file${#58}/c:/export/repository/);\
  addRepository(type:1,location:file${#58}/c:/export/repository/);

instructions.configure=\
  addRepository(type:0,location:http${#58}//localhost:1234/repository/);\
  addRepository(type:1,location:http${#58}//localhost:1234/repository/);
```

- ☐ Starten Sie das Produkt testweise aus der IDE, indem Sie die vorhandene Startkonfiguration löschen und das Produkt mit "Launch an Eclipse Application" aus dem Produkt heraus starten. Prüfen Sie, dass die neuen Menüeintrage korrekt angezeigt werden.

- ☐ Exportieren Sie Ihr Produkt in den *export*-Ordner. Wählen Sie dabei die Option *Generate metadata repository*.

- ☐ Verschieben(!) Sie die exportierte Anwendung *export/addressbook* in einen Ordner *addressbook_user*, um eine Installation beim Benutzer zu simulieren.

Karten-Plug-in installieren und updaten:

- ☐ Erstellen Sie ein neues Feature *com.example.addressbook.maps.feature* mit dem Namen *Karten*. Nehmen Sie darin nur das Karten-Plug-in auf.

- ☐ Erstellen Sie mit *File > New > Update Site Project* ein neues Update-Site-Projekt namens *addressbook_additions*. Fügen Sie eine Kategorie und das Karten-Feature hinzu.

- ☐ Bauen Sie das Repository mit *Build All*.

- ☐ Starten Sie die Anwendung aus *addressbook_user* und installieren Sie die Kartenfunktionalität per p2 aus dem *addressbook_additions* Repository.

- ☐ Nehmen Sie eine sichtbare Änderung an dem Karten-Plug-in vor (z.B. veränderter Text).

- ☐ Fügen Sie das Feature erneut in das Update-Site-Projekt hinzu und aktualisieren Sie das Repository mit *Build All*.

- ☐ Prüfen Sie, dass im Repository nun die alte und die neue Version des Plug-ins und des Features vorliegen.

- ☐ Starten Sie das Adressbuch und installieren Sie mit dem Update-Kommando die neue Version (ggf. neu starten).

Anwendungs-Update:

- ☐ Nehmen Sie eine sichtbare Änderung in einem der Adressbuch-Bundles vor (z.B. ein Reiter-Text).
- ☐ Exportieren Sie die Anwendung erneut, so dass *export/repository* aktualisiert wird.
- ☐ Starten Sie Anwendung aus dem *addressbook_user*-Ordner.

Häufige Probleme:

“Cannot complete the install because of a conflicting dependency” beim Produktexport: In dem Zielordner für den Export darf nur das Repository liegen (wird aktualisiert), ein existierender Anwendungsordner löst einen Konflikt aus.

KAPITEL 27

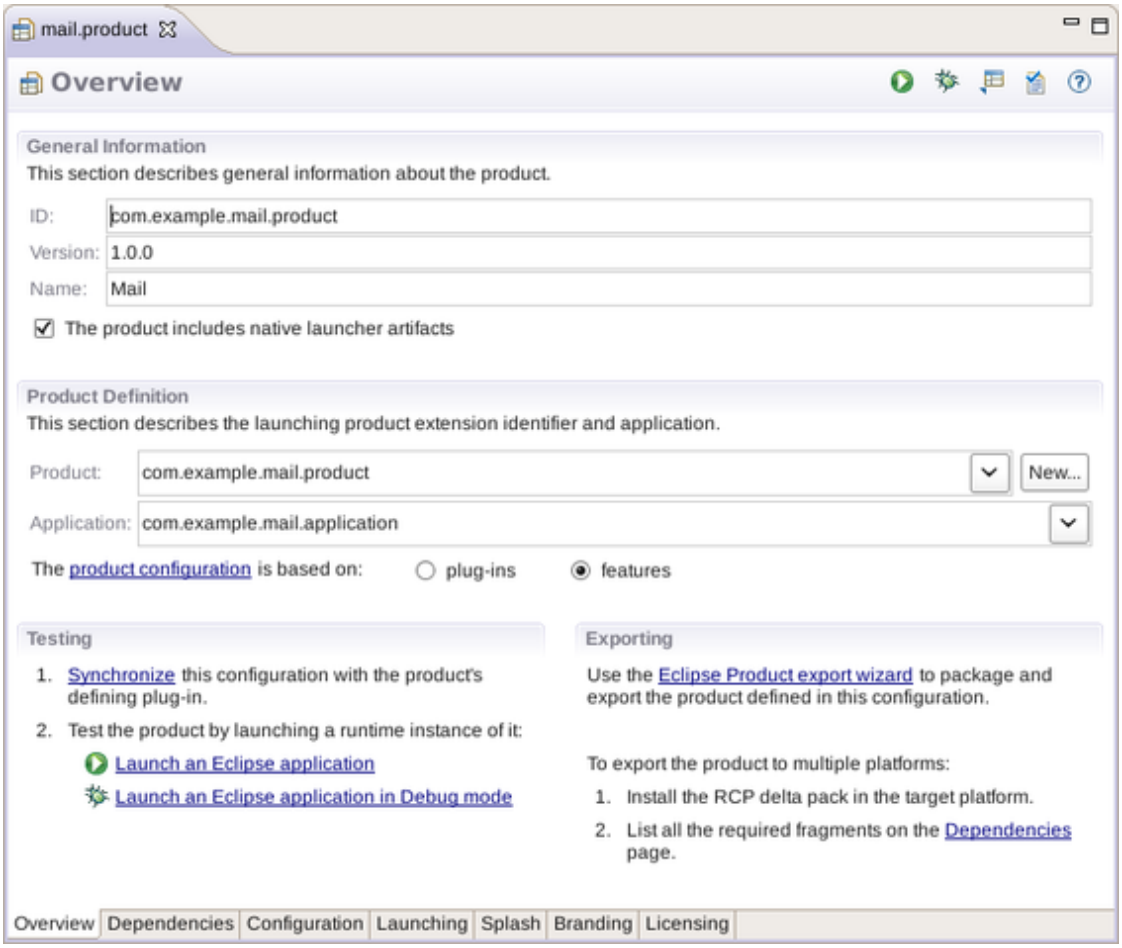
Anhang: Headless Builds

Preparation

> Create a new RCP plug-in *com.example.mail*. Create a rich client application and choose *RCP mail* as template.

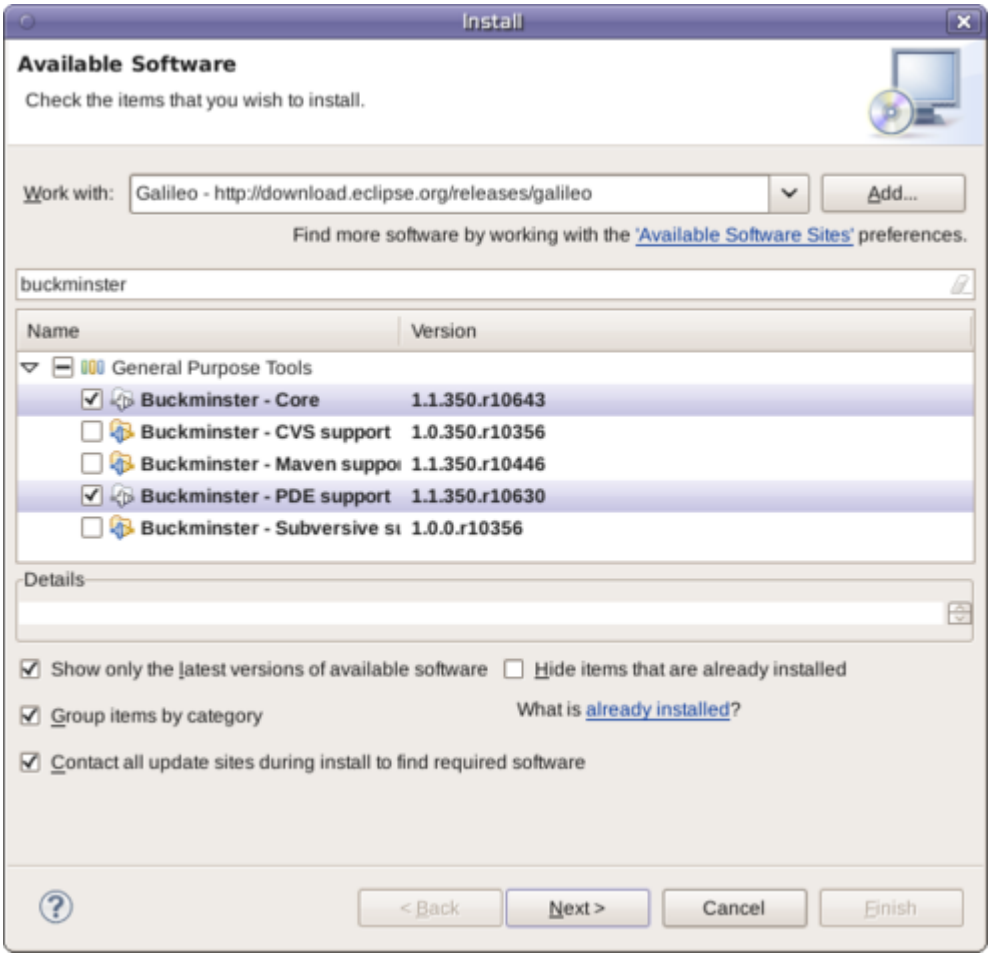
> Create a new feature *com.example.mail.feature*. Include the *com.example.mail* bundle in the feature. Add *org.eclipse.rcp* as *Included Feature*.

> Create a new product *mail.product* in the feature *com.example.mail.feature*. Configure the product: Specify a name for the product and change it to be based on features. Add the *com.example.mail.feature* (so the product contains the feature that hosts the product file). Configure a launcher name like *mail* under *Launching*.



> Run the product.

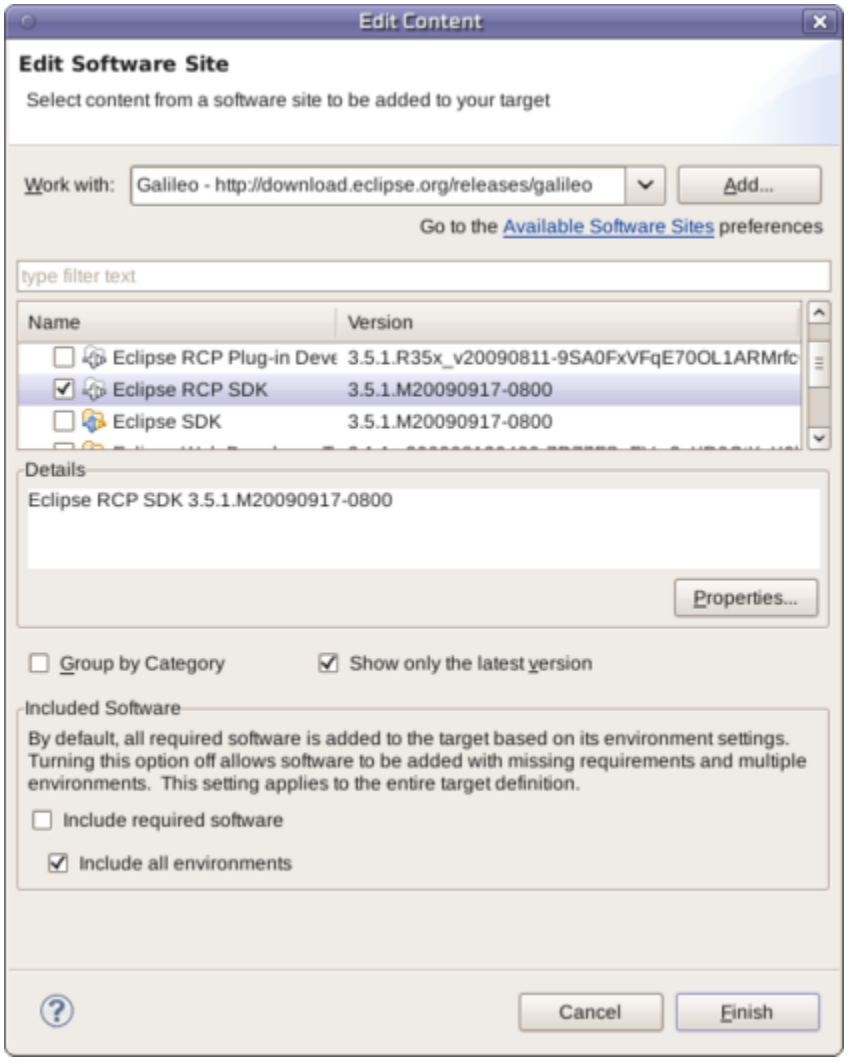
> Install Buckminster from the Eclipse release software site into your Eclipse IDE:



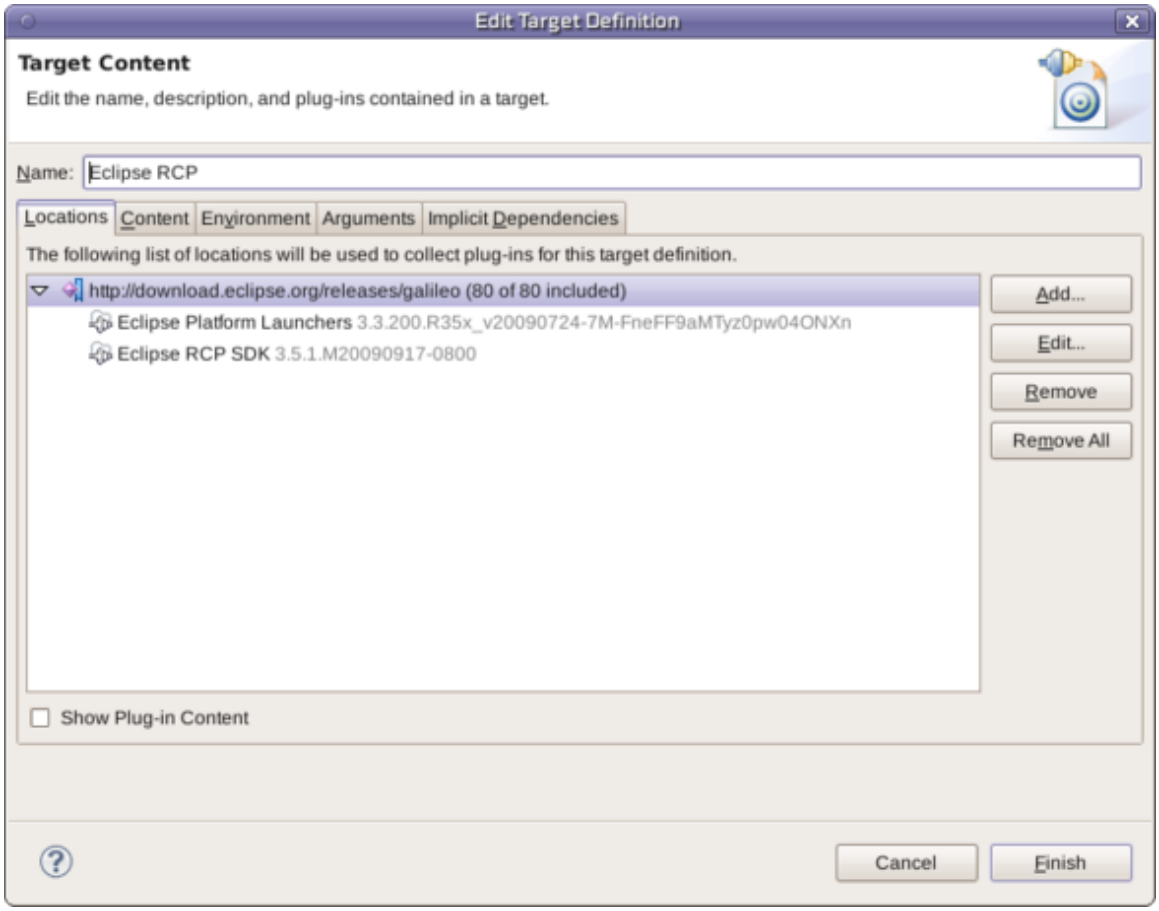
> Create a new feature *com.example.mail.site*. This feature will host all files for the Buckminster build.

Defining and sharing the target platform

Eclipse RCP applications are developed with a *target platform* containing all the platform features and bundles which are required for the application. This is configured in *Window › Preferences › Plug-in Development › Target Platform*. Create a new, empty target definition. Add *Eclipse RCP SDK* and *Eclipse Platform Launchers* from the *Eclipse Software Site* (you have to uncheck *Group by category* to see the feature). Uncheck *Include Required Software* so that you can check *Include all environments* (otherwise the target is only suitable for building on your own platform):



The target definition should look like this:



Click *Move* to move the target definition to a file *rcp.target* in the feature *com.example.mail.site*. This persists the target platform as file. That way, the target definition can be shared with other developers and can be used by the build.

Exporting a p2 site for the application using Buckminster

The first step is to use Buckminster from the IDE to build the contents of the feature *com.example.mail.site* and to create a software site for the results.

> Add the *com.example.mail.feature* as *included feature* to *com.example.mail.site*.

> You can export this feature as p2 site by invoking a Buckminster action. To do this, right-click the *com.example.mail.site* feature project and click *Buckminster > Invoke action*. You get a list of actions which can be invoked on the feature. Choose *site.p2* to create a p2 site. You also need to specify some properties for the Buckminster build. So create a temporary *buckminster.properties* file somewhere containing:

```
# Where all the output should go
buckminster.output.root=${user.home}/tmp/mail
# Where the temp files should go
buckminster.temp.root=${user.home}/tmp/buildtmp
# How .qualifier in versions should be replaced
qualifier.replacement.*=generator:lastRevision

target.os=*
target.ws=*
target.arch=*
```

This tells Buckminster where to output the results and which platforms are to be included in the p2 site.

> Go to the exported site. Under *com.example.mail.site_1.0.0-eclipse.feature/site.p2* you will find the exported p2 site containing your features and bundles for all platforms.

Installing the product from the p2 repository manually

The exported p2 site contains all the features and bundles of the product for all platforms. You could install the mail product from this *software site* now. The task of going to a p2 repository and installing an application locally is performed by the p2 director. If you have not used the *director* before, you should try to do this manually now to learn how it works. Otherwise you can skip to the next section.

> Download the standalone director from the Buckminster download page, look for *director_latest.zip*.

> Execute the director replacing the placeholders with the actual paths on your system:

```
/director
    -consolelog
    -r file://
    -d
    -i com.example.mail.product
```

> This installs the given product (or the given *installable unit*, as it is called by p2) from the repository into the destination folder. You could specify a platform using the command line arguments *-p2.os, -p2.ws, -p2.arch*. If you don't specify them, you install the product for the platform you're running on. This should show you something like:

```
Installing com.example.mail.product 1.0.0.qualifier.
Operation completed in 7096 ms.
```

> Try to start the application from the destination folder.

How to automate installing the product

Usually users don't want to install products by calling the director. Instead, a full installation is zipped together and distributed. You can automate this using Buckminster as well, but it is not supported out of the box. You need to add a custom buckminster task utilizing ant.

> Create a new folder *build* inside *com.example.mail.site* and copy the file *product.ant* in there. You can get this file from the Buckminster examples: product.ant.

> Use *File › New › Other › Buckminster › Component Specification Extension File* to create a new *buckminster.cspex* inside *com.example.mail.site*. This allows to define new Buckminster actions. The contents of this file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<cspecExtension xmlns:com="http://www.eclipse.org/buckminster/Common-1.0"
                    xmlns="http://www.eclipse.org/buckminster/CSpec-1.0">
    <actions>
        <public name="create.product" actor="ant">
            <actorProperties>
                <property key="buildFile" value="build/product.ant" />
                <property key="targets" value="create.product" />
            </actorProperties>
            <properties>
                <property key="profile" value="MailProfile" />
                <property key="iu" value="com.example.mail.product" />
            </properties>
            <prerequisites alias="repository">
                <attribute name="site.p2" />
            </prerequisites>
            <products alias="destination" base="${buckminster.output}">
                <path path="mail.${target.ws}.${target.os}.${target.arch}/" />
            </products>
        </public>
        <public name="create.product.zip" actor="ant">
            <actorProperties>
                <property key="buildFileId" value="buckminster.pdetasks" />
                <property key="targets" value="create.zip" />
            </actorProperties>
            <prerequisites alias="action.requirements">
                <attribute name="create.product" />
            </prerequisites>
            <products alias="action.output" base="${buckminster.output}">
                <path path="mail.${target.ws}.${target.os}.${target.arch}.zip" />
            </products>
        </public>
    </actions>
</cspecExtension>
```

> This adds two new actions *create.product* and *create.product.zip* which basically call the p2 director using ant. Please note that the name of the destination folder/zip and the product id are specified in the *cspex* actions.

> Click *Buckminster > Invoke action* again and choose *create.product*. Please have a look in the *buckminster.properties* file you used to create the p2 site before. Creating the product would not work with * for *os/ws/arch*, because you can create the product only for a specific platform. So create a second file *buckminster_product.properties* to specify a concrete platform, f.e.:

```
# Where all the output should go
buckminster.output.root=${user.home}/tmp/mail
# Where the temp files should go
buckminster.temp.root=${user.home}/tmp/buildtmp
# How .qualifier in versions should be replaced
qualifier.replacement.*=generator:lastRevision

target.os=win32
target.ws=win32
target.arch=x86
```

> Check that the product was correctly exported for the specified os/ws/arch combination by running it on that platform.

Automating the build using Hudson

One very helpful aspect about building with Buckminster is that you can automate builds using the Hudson build system. Before we can do that, we need to define how Buckminster should obtain our projects.

> To tell Buckminster to fetch our feature / bundle projects, we have to use a *component query* (*.cquery*). Create a new file *site.cquery* inside *com.example.mail.site*:

```
<?xml version="1.0" encoding="UTF-8"?>
<cq:componentQuery xmlns:cq="http://www.eclipse.org/buckminster/CQuery-1.0"
                   resourceMap="site.rmap">
    <cq:rootRequest name="com.example.mail.site" componentType="eclipse.feature"/>
</cq:componentQuery>
```

> As Hudson will check out the files and provide them inside a workspace folder for us, we need to tell Buckminster to get the features and bundles from this folder. For this, create a new resource map *site.rmap* inside *com.example.mail.site*:

```
<?xml version="1.0" encoding="UTF-8"?>
<rmap
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.eclipse.org/buckminster/RMap-1.0"
    xmlns:bc="http://www.eclipse.org/buckminster/Common-1.0"
    xmlns:mp="http://www.eclipse.org/buckminster/MavenProvider-1.0"
    xmlns:pp="http://www.eclipse.org/buckminster/PDEMapProvider-1.0">

    <searchPath name="resources">
        <provider readerType="local" componentTypes="osgi.bundle,eclipse.feature"
                  mutable="true" source="true">
            <uri format="file:///{0}/{1}/">
                <bc:propertyRef key="workspace.root" />
                <bc:propertyRef key="buckminster.component" />
            </uri>
        </provider>
    </searchPath>

    <locator searchPathRef="resources"/>
</rmap>
```

> Install the Hudson build server (Installation guide for Hudson server, Hudson on Tomcat, Latest Hudson WAR).

> Go to *Hudson* › *Manage Plug-ins* and install the Hudson Buckminster plug-in.

> So far, we have been running Buckminster from the Eclipse IDE. For running Buckminster from Hudson, we need to get a copy of 'Buckminster Headless'. But you can't download that. Do you remember how we installed the mail application from the p2 repository using the p2 director? That's how you obtain Buckminster Headless:

```
./director
    -r http://download.eclipse.org/tools/buckminster/headless-3.5/
    -d /path/to/buckminster-headless/
    -p Buckminster
    -i org.eclipse.buckminster.cmdline.product
```

It's recommended to use an absolute path as destination path.

> Once you have Buckminster headless, you can install additional features. As we are going to build products and features we need to install the Core and PDE features:

```
cd /path/to/buckminster-headless/
./buckminster install http://download.eclipse.org/tools/buckminster/headless-3.5/
    org.eclipse.buckminster.core.headless.feature
./buckminster install http://download.eclipse.org/tools/buckminster/headless-3.5/
    org.eclipse.buckminster.pde.headless.feature
```

> Configure the path to your buckminster headless installation in *Hudson* > *Manage Hudson* > *Configure System*.

> Create a new free-style job in Hudson.

> Hudson expects to check out the files from some version control system like CVS, SVN or git. To continue, you need to have your files in some version control system. Configure the job to check out the projects from your source code repository.

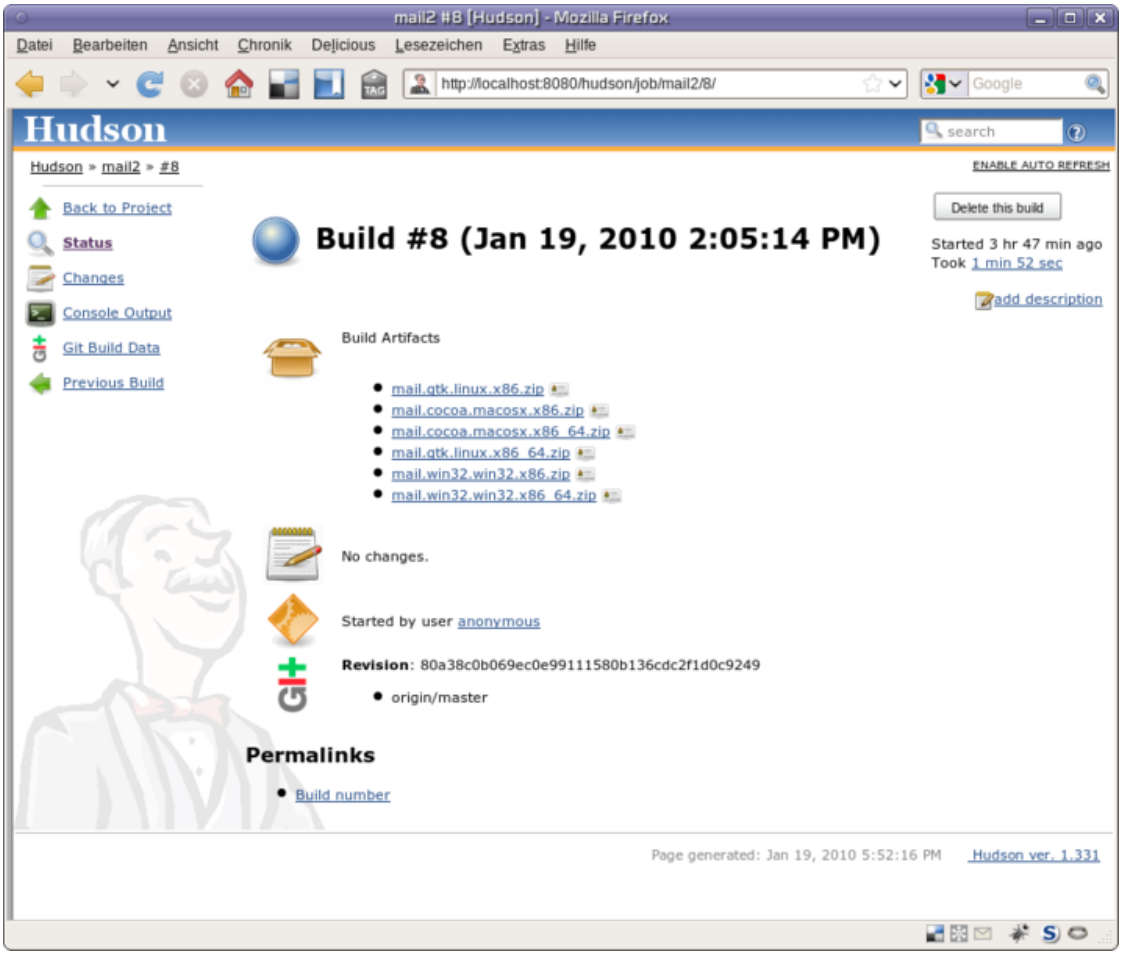
> Create a new build step *Run Buckminster* that 1. imports the target definition and the projects, 2. builds the project and 3. creates product installations for all platforms:

```
importtargetdefinition -A '${WORKSPACE}/com.example.mail.site/rcp.target'
import '${WORKSPACE}/com.example.mail.site/site.cquery'
build
perform -D target.os=* -D target.ws=* -D target.arch=*
        com.example.mail.site#site.p2
perform -D target.os=win32 -D target.ws=win32 -D target.arch=x86
        com.example.mail.site#create.product.zip
perform -D target.os=linux -D target.ws=gtk -D target.arch=x86
        com.example.mail.site#create.product.zip
...
```

> Check *Archive the artifacts* and specify these files (ignore the warning which might be shown):

```
buckminster.output/com.example.mail.site_*-eclipse.feature/mail*.zip
```

> Run the build job:



Example code

> Example project “com.example.mail.buckminster”
http://github.com/ralfebert/com.example.mail.buckminster

> Example project “org.eclipse.buckminster.tutorial.mailapp.releng”
http://dev.eclipse.org/viewsvn/index.cgi/trunk/?root=Tools_BUCKMINSTER

More information

- Buckminster Documentation: Eclipse Buckminster, The Definitive Guide
  http://www.eclipse.org/buckminster/
- Building an RCP application with Hudson (Buckminster)
  http://wiki.eclipse.org/Building_an_RCP_application_with_hudson_%28Buckminster%29
- Hudson Buckminster Plug-in
  http://wiki.hudson-ci.org/display/HUDSON/Buckminster+PlugIn

KAPITEL 28

Änderungshistorie

Version 1.1 vom 19.08.2011

- Update für Eclipse Version 3.7
- Aktualisierte Target Plattform bereitgestellt unter *http://www.ralfebert.de/eclipse_rcp/target/*
- Die Registrierung der Actions im *ActionBarAdvisor* für die Verwendung von Commands ist seit Version 3.7 nicht mehr notwendig.
- Hinweis: Der Produktexport für fremde Zielplattformen funktioniert aktuell aufgrund des Eclipse-Bugs 352361 nicht.
- Tutorial-Schritte zur Einbindung des Hilfesystems
- Beispiel für ColumnLabelProvider in Kapitel 13 korrigiert, ColumnLabelProvider gibt den Text zurück statt ihn der Zelle zu setzen
- Expliziter Schritt “Perspektive hinzufügen” im Tutorial 7.1 “Adressmaske erstellen” ergänzt
- Hinweis zur Einblendung der Perspektiv-Bar in Tutorial 4.1 ergänzt
- Hinweis zum Zugriff auf non-final Variablen in Listenern in Tutorial 6.1 hinzufügt

Version 1.0 vom 17.06.2011

- Initiale Veröffentlichung für Eclipse 3.6.2