

## 4 RCP-Entwicklung

Nach der Einführung in die Plugin-Entwicklung können wir uns nun der Erstellung von Rich-Client-Anwendungen widmen, denn auch unter der Eclipse-Rich-Client-Plattform erfolgt die Implementierung von Anwendungsfunktionalität in Form von Plugins. Dabei kann eine Anwendung aus mehreren Plugins bestehen bzw. später durch zusätzliche Plugins erweitert werden. Grundsätzlich besitzt aber jede RCP-Anwendung ein bestimmtes Plugin, welches die RCP-Anwendung konfiguriert und die Anwendung gegenüber der Eclipse-Plattform identifiziert. Dieses Plugin bezeichnen wir in Zukunft als Anwendungs-Plugin.

### 4.1 Plugins und die RCP

Neben der in Abschnitt 2.2 erstellten Anwendung kennen Sie bereits eine weitere Rich-Client-Anwendung, und zwar die Eclipse-IDE. Auch die Eclipse-IDE ist nur eine spezifische Anwendung, die auf die Eclipse-RCP aufsetzt. Das Plugin `org.eclipse.ui.ide.application` besorgt das Starten dieser Applikation. Schauen wir uns einmal das Plugin-Manifest `plugin.xml` dieses Plugins etwas genauer an. (Sie können dieses mit einem Doppelklick auf den Eintrag `org.eclipse.ui.ide.application` im *Plug-ins* View öffnen.) Hier können wir lernen, wie das Anwendungs-Plugin einer RCP-Anwendung deklariert sein muss. So wird am Erweiterungspunkt `org.eclipse.core.runtime.applications` die Wurzelklasse für die RCP-Applikation definiert:

```
<extension id="org.eclipse.ui.ide.workbench"
  point="org.eclipse.core.runtime.applications">
  <application>
    <run class=
      "org.eclipse.ui.internal.ide.application.IDEApplication">
    </run>
  </application>
</extension>
```

Anschließend erfolgt die Definition einer Perspektive, denn eine Standardperspektive (Layout des Workbench-Fensters) sollte jede RCP-Applikation haben. Beim Eclipse-IDE wird die *Resource*-Perspektive als Standardperspektive verwendet:

```
<extension point="org.eclipse.ui.perspectives">
  <perspective name="%Perspective.resourcePerspective"
    icon="$nl$/icons/full/eview16/resource_persp.gif"
    class=
      "org.eclipse.ui.internal.ide.application.ResourcePerspective"
    id="org.eclipse.ui.resourcePerspective">
    <description>
      %Perspective.resourceDescription
    </description>
  </perspective>
</extension>
```

Die Einzelheiten des Layouts des Workbench-Fensters werden in der Klasse *ResourcePerspective* oder in nachfolgenden Erweiterungen des Punktes *org.eclipse.ui.perspectiveExtensions* festgelegt.

## 4.2 Applikationen

Jede RCP-Anwendung muss mindestens eine Applikationsklasse am Erweiterungspunkt *org.eclipse.core.runtime.applications* deklarieren, die das Interface *IApplication* implementieren muss. Im obigen Beispiel war das die Klasse *IDEApplication*, in Abschnitt 2.2 die Klasse *com.bdaum.planner.Application*. Das Interface dient dazu, seine Implementatoren gegenüber Eclipse als Applikations-Eintrittspunkte zu identifizieren. Wir sprechen hier in der Mehrzahl, denn es ist durchaus möglich, mehrere Eintrittspunkte zu spezifizieren. Welcher der Eintrittspunkte beim Start von Eclipse aktiv werden soll, lässt sich über die Konfigurationsdatei *config.ini* von außerhalb des Plugins nachträglich steuern. Doch dazu später mehr.

### 4.2.1 Das Interface *IPlatformRunnable*

Klassen, die das Interface *IApplication* implementieren, müssen eine *start()*- und eine *stop()*-Methode bereitstellen. Die typische Implementierung für diese *start()*-Methode sieht so aus:

```

public Object start(IApplicationContext context)
    throws Exception {
    Display display = PlatformUI.createDisplay();
    try {
        int returnCode = PlatformUI.createAndRunWorkbench(
            display, new ApplicationWorkbenchAdvisor());
        if (returnCode == PlatformUI.RETURN_RESTART)
            return IApplication.EXIT_RESTART;
        else
            return IApplication.EXIT_OK;
    } finally {
        display.dispose();
    }
}

```

Zunächst wird von der Klasse PlatformUI eine neue Display-Instanz erzeugt, welche die Bildschirmanzeige repräsentiert. Anschließend erfolgt der Start der Workbench mit Hilfe der PlatformUI-Methode createAndRunWorkbench(). Dabei wird zunächst ein neuer WorkbenchAdvisor erzeugt, der für die Konfiguration der Benutzeroberfläche zuständig ist (siehe Abschnitt 4.3.1). Wird die Workbench beendet, wird der Rückgabecode der Workbench noch in das OSGi-konforme IPlatformRunnable-Protokoll übersetzt.

Selbstverständlich ist man keineswegs gezwungen, die Methoden der Klasse PlatformUI auszuführen. Stattdessen könnte man eine mit SWT und JFace komplett selbst gebaute Benutzeroberfläche zum Ablauf bringen, müsste dann allerdings auf die höheren Schichten der Workbench wie Editoren und Views verzichten. Eine mögliche Anwendung einer solchen selbst gebauten Benutzeroberfläche wäre ein Login-Dialog, der erscheinen soll, bevor die Workbench aktiv wird.

So könnte man beispielsweise in der Klasse Application im Plugin com.bdaum.planner am Anfang des try-Blockes den folgenden Codeblock einfügen:

*Beispiel*

```

LoginDialog dialog = new LoginDialog(null);
dialog.create();
dialog.open();
User user = dialog.getUser();
if (user == null)
    // Shutdown
    return IApplication.EXIT_OK;
PlannerActivator.getDefault().setUser(user);

```

Hier wird ein Anmeldedialog aufgebaut. Erhält dieser Dialog gültige Benutzerdaten, werden sie der `PlannerActivator`-Instanz mitgeteilt. Ist das nicht der Fall, wird die Anwendung beendet. Da wir später in Abschnitt 5.1.2 eine elegantere Möglichkeit finden, um einen Login-Dialog zu zeigen, verzichten wir hier auf eine Änderung der Klasse `Application`.

Die `stop()`-Methode der Anwendungsklasse sieht im Regelfall folgendermaßen aus:

```
public void stop() {
    final IWorkbench workbench = PlatformUI.getWorkbench();
    if (workbench == null)
        return;
    final Display display = workbench.getDisplay();
    display.syncExec(new Runnable() {
        public void run() {
            if (!display.isDisposed())
                workbench.close();
        }
    });
}
```

Ist die `Workbench` bereits geschlossen, beendet sich die Methode. Andernfalls wird die `Workbench` mit `close()` geschlossen. Dies erfolgt mittelbar über die `Display`-Methode `syncExec()` (siehe Abschnitt 9.1.3), so dass die `stop()`-Methode auch aus einem anderen Thread aufgerufen werden darf. Die `stop()`-Methode wird von der Eclipse-Plattform dazu verwendet, die `Workbench` zwangsweise zu schließen. Sie sollte nie aus einer Anwendung heraus aufgerufen werden. Der korrekte Weg, die `Workbench` herunterzufahren ist

```
PlatformUI.getWorkbench().close();
```

### 4.3 Die Workbench der Beispielanwendung

Im Projekt `com.bdaum.planner` können Sie sehen, dass außer der Klasse `Application` noch einige weitere Klassen generiert werden. Dazu zählen insbesondere die Klassen:

```
ApplicationActionBarAdvisor
ApplicationWorkbenchAdvisor
ApplicationWorkbenchWindowAdvisor
Perspective
```

Diese Klassen dienen dazu, die Workbench der Anwendung zu konfigurieren, und können entsprechend abgeändert werden.

### 4.3.1 Die Workbench konfigurieren

In Abschnitt 4.2.1 hatten wir bereits gesehen, wie in der PlatformUI-Methode `createAndRunWorkbench()` eine `ApplicationWorkbenchAdvisor`-Instanz erzeugt und an die Workbench übergeben wurde. Diese Klasse wird dazu verwendet, an verschiedenen Punkten des Lebenszyklus einer Applikation die Konfiguration der Eclipse-Workbench in geeigneter Weise einzustellen. An diesen Punkten werden bestimmte, von der Superklasse `WorkbenchAdvisor` implementierte Methoden aufgerufen, die geeignet überschrieben werden können.

Sehen wir uns diese Methoden etwas genauer an.

*WorkbenchAdvisor*

<b>getInitialWindow-PerspectiveId</b>	Diese Methode muss die Identifikation der anfänglich zu öffnenden Perspektive liefern.
<b>getDefaultPageInput</b>	Diese Methode liefert Eingabewerte für neu geöffnete Workbench-Seiten, sofern keine Eingabewerte beim Öffnen angegeben wurden. Die Standardimplementierung liefert den Wert <code>null</code> .
<b>getMainPreference-PageId</b>	Liefert die Identifikation der wichtigsten Präferenzseiten. Die Standardimplementierung liefert den Wert <code>null</code> , was heißt, dass die Seiten alphabetisch sortiert werden.
<b>initialize</b>	Diese Methode wird von Eclipse aufgerufen, bevor eine neue Workbench-Instanz angelegt wird. Standardmäßig geschieht in dieser Methode nichts. Durch Überschreiben der Methode kann man hier aber z.B. die Kommandozeile, die beim Starten der Plattform mitgegeben wurde, untersuchen. Außerdem können andere Initialisierungen wie das Registrieren von Adaptern oder das Laden von Bildern vorgenommen werden.
<b>preStartup</b>	Diese Methode wird nach der Initialisierung der Workbench (jedoch vor dem Öffnen des ersten Fensters) aufgerufen. Auch hier geschieht standardmäßig nichts. Durch Überschreiben der Methode kann man bestimmen, welche Editoren und Views anfänglich geöffnet werden sollen.
<b>postStartup</b>	Diese Methode wird aufgerufen, nachdem alle Fenster geöffnet bzw. wiederhergestellt wurden, aber bevor die zentrale Ereignisschleife gestartet wird. Hier können Prozesse aufgerufen werden, die automatisch ablaufen sollen, oder es können Tipps angezeigt oder weitere Fenster geöffnet werden. Standardmäßig geschieht hier nichts. →

<b>preShutdown</b>	Diese Methode wird nach Beendigung der Ereignisschleife aufgerufen, aber noch bevor Fenster geschlossen werden. Durch Rückgabe des Wertes <code>false</code> kann ein nicht erzwungener Shutdown verhindert werden (Veto). Standardmäßig gibt die Methode den Wert <code>true</code> zurück.
<b>postShutdown</b>	Diese Methode wird während des Herunterfahrens der Workbench aufgerufen, nachdem alle Fenster geschlossen wurden. Standardmäßig geschieht hier nichts. Durch Überschreiben der Methode kann hier der Zustand der Workbench dauerhaft abgespeichert werden, oder Ressourcen können wieder freigegeben werden.
<b>eventLoopException</b>	Diese Methode wird beim Auftreten einer nicht abgefangenen Ausnahmebedingung ( <i>Exception</i> ) aufgerufen. Die Standardimplementierung schreibt die Ausnahmebedingung in die Eclipse-Logdatei.
<b>eventLoopIdle</b>	Diese Methode wird aufgerufen, wenn der Ereignisschleife keine (weiteren) Ereignisse zur Bearbeitung vorliegen. Hier könnten Arbeiten ausgeführt werden, die im Hintergrund ablaufen sollen.

#### *IWorkbenchConfigurer*

Der mit der Methode `getWorkbenchConfigurer()` erhaltene *IWorkbenchConfigurer* stellt die notwendigen Methoden für die Konfiguration der Workbench zur Verfügung. So kann mit Methode `setSaveAndRestore()` bestimmt werden, ob der Zustand der Workbench beim Schließen dauerhaft gespeichert werden soll, damit beim erneuten Start der aktuelle Zustand wiederhergestellt werden kann. In der Standardeinstellung der Workbench geschieht das nicht!

Mit den Methoden `setData()` und `getData()` können einer Workbench innerhalb einer Sitzung beliebige Datenobjekte unter einem Schlüssel zugeordnet und später wieder abgefragt werden. Mit der Methode `declareImage()` werden unter einem symbolischen Namen Bilder deklariert, die während des Lebenszyklus der Workbench zur Verfügung stehen sollen. Solche Bilder können später von der Workbench-Instanz mit der Methode `getSharedImages()` geholt werden. Die Workbench kümmert sich komplett um die Verwaltung dieser Bilder – das lästige, unter SWT übliche Laden und Wiederfreigeben der Bildressourcen entfällt.

Außerdem stellt der *IWorkbenchConfigurer* noch die Methode `emergencyClose()` zur Verfügung, mit der die Applikation in Notfällen (z.B. bei *Out of Memory*) zwangsweise beendet werden kann. Mit der Methode `emergencyClosing()` lässt sich feststellen, ob gerade ein solcher Abbruch vorliegt. Alle *IWorkbenchConfigurer*-Methoden sollten, bevor Sie eine Benutzerinteraktion durchführen wollen, diese Methode aufrufen und im Falle eines Notfall-Abbruchs diese Interaktion tunlichst unterlassen.

Schließlich besteht noch über die Methoden `getWorkbenchWindowManager()` und `getWindowConfigurer()` Zugriff auf die Workbench-Fenster, um deren Konfiguration wir uns im nächsten Abschnitt kümmern.

In unserem Projektplaner soll die anfänglich gezeigte Perspektive vom jeweiligen Benutzer abhängen. Wir nehmen hier zunächst an, dass die Daten des aktuellen Benutzers im Aktivator gespeichert sind, der ja den Lebenszyklus der Anwendung verwaltet. Wir fragen deshalb einfach den Aktivator nach der anfänglichen Perspektive ab. Gibt es den Wert `null` zurück, verwenden wir die vorgenerierte Perspektive *Login*. Wir modifizieren deshalb in der Klasse `ApplicationWorkbenchAdvisor` die Methode `getInitialWindowPerspectiveId()`:

*Beispiel*

```
package com.bdaum.planner;
import org.eclipse.core.runtime.IRegistryChangeEvent;
import org.eclipse.core.runtime.IRegistryChangeListener;
import org.eclipse.core.runtime.Platform;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPage;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.application.IWorkbenchConfigurer;
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchAdvisor;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;

public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
    private static final String PERSPECTIVE_ID =
        "com.bdaum.planner.perspective";

    @Override
    public WorkbenchWindowAdvisor createWorkbenchWindowAdvisor(
        IWorkbenchWindowConfigurer configurer) {
        return new ApplicationWorkbenchWindowAdvisor(configurer);
    }

    @Override
    public String getInitialWindowPerspectiveId() {
        String initialPerspectiveId = PlannerActivator.getDefault().
            getInitialPerspectiveId();
        if (initialPerspectiveId != null)
            return initialPerspectiveId;
        return PERSPECTIVE_ID;
    }
}
```

Da die Methode `getInitialPerspectiveId()` in der Klasse `PlannerActivator` noch nicht existiert, erzeugen wir sie kurzerhand mit einem Klick auf das *QuickFix*-Symbol.

*Initialisieren*

In unserer Beispielanwendung möchten wir erreichen, dass der Benutzer die Workbench bei der nächsten Sitzung so vorfindet, wie er sie bei der vorigen Sitzung verlassen hat. Wir überschreiben dazu die Methode `initialize()`.

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

*Nach Startup*

Allerdings benötigen wir eine Ausnahme von der Regel. Beim Beginn einer neuen Session soll der Benutzer nicht mit der zuletzt verwendeten Perspektive starten, sondern mit der in der Rollendefinition festgelegten Perspektive. Wir können das erreichen, indem wir nach dem Startup der Workbench zwangweise in die Anfangsperspektive umschalten.

Der Benutzer muss ja nicht in der anfänglichen Perspektive bleiben, sondern kann je nach seiner Rolle unter den verfügbaren Perspektiven auswählen. Welche Perspektiven verfügbar sind, hängt freilich von den installierten Plugins ab. Doch was geschieht, wenn während des laufenden Betriebs neue Plugins hinzugefügt oder Plugins deaktiviert werden? In diesem Falle müsste das GUI entsprechend aktualisiert werden. Auch das ist möglich. Es genügt, sich bei der ExtensionRegistry als Listener zu registrieren (siehe Abschnitt 3.3.5) und bei einer Änderung der Registratur alle Perspektiven zurückzusetzen. Die Perspektiven werden dann neu aufgebaut, folglich auch die Registratur implizit neu ausgewertet. Am besten meldet man sich direkt nach dem Start der Workbench als `IRegistryChangeListener` an:

```
@Override
public void postStartup() {
    super.postStartup();
    // Anfangsperspektive erzwingen
    String perspectiveId = getInitialWindowPerspectiveId();
    IWorkbench workbench =
        getWorkbenchConfigurer().getWorkbench();
    IWorkbenchWindow firstWindow =
        workbench.getWorkbenchWindows()[0];
    try {
        workbench.showPerspective(perspectiveId, firstWindow);
    } catch (WorkbenchException e) {
        // Ignorieren
    }
}
```

```

// Registraturänderungen berücksichtigen
IRegistryChangeListener registryChangeListener =
    new IRegistryChangeListener() {
        public void registryChanged(
            IRegistryChangeEvent event) {
            if (event.getExtensionDeltas(
                PlannerActivator.PLUGIN_ID, "roles").length > 0) {
                IWorkbench workbench = PlatformUI.getWorkbench();
                for (IWorkbenchWindow window : workbench
                    .getWorkbenchWindows())
                    for (IWorkbenchPage page : window.getPages())
                        page.resetPerspective();
            }
        }
    };
Platform.getExtensionRegistry()
    .addRegistryChangeListener(registryChangeListener);
}

```

### 4.3.2 Workbench-Fenster konfigurieren

Zusätzlich zu den Methoden, die der `WorkbenchAdvisor` für die Konfiguration der gesamten RCP-Applikation bereitstellt, kann er die Workbench auch mit einem `WorkbenchWindowAdvisor` versorgen, welcher die Konfiguration der einzelnen Workbench-Fenster übernimmt. In der konkreten `WorkbenchWindowAdvisor`-Implementierung können dazu die folgenden Lebenszyklus-Methoden überschrieben werden.

<b>createEmptyWindowContents</b>	Diese Methode kann Fensterinhalte für den Fall erzeugen, dass das Fenster keine aktiven Seiten (Perspektiven) besitzt.
<b>openIntro</b>	Der Willkommensschirm wird normalerweise im ersten geöffneten Workbench-Fenster geöffnet, aber nur, wenn das in den Einstellungen so gewünscht ist und wenn der Willkommensschirm vor dem letzten Herunterfahren nicht geschlossen wurde. Durch geeignetes Überschreiben dieser Methode kann dieses Verhalten geändert werden. So könnte der Willkommensschirm zwangsweise immer beim Öffnen des ersten Fensters angezeigt werden.
<b>createWindowContents</b>	Diese Methode erzeugt den Fensterinhalt. Die Standardimplementierung erzeugt die Menüleiste, die Werkzeugleiste, die Statuszeile, die Perspektivenleiste und die Leiste für die <i>Fast Views</i> .
<b>postWindowCreate</b>	Diese Methode wird nach der Erzeugung eines Workbench-Fensters aufgerufen. Standardmäßig geschieht hier nichts. Durch Überschreiben der Methode kann hier das Fenster konfiguriert werden, z.B. kann festgelegt werden, ob das Fenster ein Menü besitzt oder nicht. Allerdings sind zu diesem Zeitpunkt die Fensterinhalte noch nicht vorhanden. →

<b>postWindowRestore</b>	Diese Methode wird aufgerufen, nachdem ein Fenster wiederhergestellt wurde, dessen Zustand dauerhaft gespeichert wurde. Standardmäßig geschieht hier nichts.
<b>preWindowOpen</b>	Wird aufgerufen, bevor der Fensterinhalt aufgebaut wird.
<b>postWindowOpen</b>	Diese Methode wird aufgerufen, nachdem ein Fenster geöffnet wurde. Hier können Fensterinhalte verändert werden. Typischerweise wird hier der Fenstertitel oder die Fenstergröße gesetzt. Standardmäßig geschieht hier nichts.
<b>preWindowShellClose</b>	Diese Methode wird beim Schließen eines Fensters aufgerufen, jedoch bevor die Shell des Fensters geschlossen wird. Es ist möglich, durch Rückgabe des Wertes <code>false</code> das Schließen des Fensters zu verhindern. Eine typische Anwendung dafür wäre die Abfrage, ob offene Dateien geschlossen werden sollen, wenn das Workbench-Fenster geschlossen wird. Betätigt der Benutzer die Abbruch-Taste, wird <code>false</code> zurückgegeben, um das Schließen des Workbench-Fensters zu verhindern. Standardmäßig gibt diese Methode den Wert <code>true</code> zurück.
<b>postWindowClose</b>	Diese Methode wird nach dem Schließen eines Fensters aufgerufen. Üblicherweise wird man hier Aufräumarbeiten erledigen. Standardmäßig geschieht hier nichts.

#### *IWorkbenchWindow- Configurer*

Mit der Methode `getWindowConfigurer()` erhält man eine Instanz des Typs `IWorkbenchWindowConfigurer`. Durch Aufrufe von entsprechenden `IWorkbenchWindowConfigurer`-Methoden kann das jeweilige Workbench-Fenster konfiguriert werden (das Fenster erhält man mittels Methode `getWindow()` von der `IWorkbenchWindowConfigurer`-Instanz). Beispielsweise kann mit `setTitle()` der Fenstertitel eingestellt werden. Daneben gibt es noch Methoden zum Ein- bzw. Ausschalten verschiedener Fensterkomponenten:

#### *Aktionsleisten ausschalten*

<b>setShowTitleBar()</b>	Mit dieser Methode kann die Titelzeile an- oder abgeschaltet werden.
<b>setShowMenuBar()</b>	Mit dieser Methode kann die Menüleiste an- oder abgeschaltet werden.
<b>setShowCoolBar()</b>	Mit dieser Methode kann die Werkzeugleiste an- oder abgeschaltet werden.
<b>setShowFastViewsBars()</b>	Mit dieser Methode kann die Leiste für die <i>Fast Views</i> an- oder abgeschaltet werden.
<b>setShowPerspectiveBar()</b>	Mit dieser Methode kann die Perspektivenleiste an- oder abgeschaltet werden.
<b>setShowProgressIndicator()</b>	Mit dieser Methode kann der integrierte Fortschrittsbalken an- oder abgeschaltet werden.
<b>setShowStatusLine()</b>	Mit dieser Methode kann die Statuszeile an- oder abgeschaltet werden.

Selbstverständlich gibt es für alle `set...()`-Methoden auch die entsprechende `get...()`-Methode.

Kommen im Rahmen einer Rich-Client-Anwendung Editoren zum Einsatz, wird man in vielen Fällen den Editorbereich als *Drag&Drop*-Bereich ausrüsten wollen. So wird es z.B. möglich, Objekte aus einem View oder vom Desktop der Ablaufplattform in den Editorbereich hineinzuziehen. Mit Hilfe der `IWorkbenchWindowConfigurer`-Methoden `addEditorAreaTransfer()` ist das möglich. Registriert man beispielsweise den Transfertyp `EditorInputTransfer` mit Hilfe der Methode `addEditorAreaTransfer()`, so können Objekte vom Typ `IEditorInput` mittels *Drag&Drop* verschoben werden. Mit `configureEditorAreaDropListener()` wird ein neuer `DropTargetListener` für den Editorbereich registriert. Dieser Listener kann auf *Drag&Drop*-Ereignisse reagieren. Ausführliche Informationen über *Drag&Drop* geben wir in Kapitel 9.

*Drag&Drop*

Mit der Methode `setShellStyle()` können Aussehen und Verhalten der Fenster-Shell beeinflusst und mit `setInitialSize()` die anfängliche Größe des Fensters gesetzt werden. Natürlich müssen diese Methoden in Methode `preWindowOpen()` aufgerufen werden. Als Stilkonstanten kommen in `setShellStyle()` in Frage:

*Shell-Konfiguration*

<code>SWT.NONE</code>	Standardfenster (betriebssystemabhängig)
<code>SWT.BORDER</code>	Fenster hat einen Rand (plattformabhängig)
<code>SWT.CLOSE</code>	Fenster hat eine Titelzeile mit einem Button zum Schließen des Fensters
<code>SWT.MIN</code>	Fenster hat eine Titelzeile mit einem Button zum Minimieren des Fensters
<code>SWT.MAX</code>	Fenster hat eine Titelzeile mit einem Button zum Maximieren des Fensters
<code>SWT.NO_TRIM</code>	Fenster hat weder Titelzeile noch Rand
<code>SWT.RESIZE</code>	Fenstergröße kann durch Ziehen mit der Maus verändert werden
<code>SWT.TITLE</code>	Fenster hat eine Titelzeile
<code>SWT.SHELL_TRIM</code>	Kombination der Stilelemente <code>SWT.CLOSE</code> , <code>SWT.TITLE</code> , <code>SWT.MIN</code> , <code>SWT.MAX</code> , <code>SWT.RESIZE</code>
<code>SWT.DIALOG_TRIM</code>	Kombination der Stilelemente <code>SWT.CLOSE</code> , <code>SWT.TITLE</code> , <code>SWT.BORDER</code>

Daneben bestünde noch die Möglichkeit, das Workbench-Fenster mit `SWT.SYSTEM_MODAL` systemweit modal zu machen. Ein modales Fenster, das sich im Vordergrund befindet, lässt keine anderen Fenster in den Vordergrund kommen. Ein solches Fenster sollte so instrumentiert sein, dass der Anwender das Fenster auch schließen kann! Die Standardeinstellung für das Workbench-Fenster ist `SWT.SHELL_TRIM`.

Beispiel

Bei unserem Projektplaner möchten wir das Workbench-Fenster etwas anders konfigurieren als vorgeneriert. Es soll größer sein, außerdem möchten wir eine Werkzeugleiste, eine Statusleiste mit Fortschrittsbalken, eine Perspektivauswahlliste und eine *FastView*-Leiste haben, um Views auch als *Fast Views* anlegen und öffnen zu können. Wir modifizieren deshalb die Klasse `ApplicationWorkbenchWindowAdvisor`:

```
package com.bdaum.planner;
import org.eclipse.swt.graphics.Point;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;
import org.eclipse.ui.application.IWorkbenchWindowConfigurer;
import org.eclipse.ui.application.WorkbenchWindowAdvisor;

public class ApplicationWorkbenchWindowAdvisor extends
        WorkbenchWindowAdvisor {

    public ApplicationWorkbenchWindowAdvisor(
        IWorkbenchWindowConfigurer configurer) {
        super(configurer);
    }

    public ActionBarAdvisor createActionBarAdvisor(
        IActionBarConfigurer configurer) {
        return new ApplicationActionBarAdvisor(configurer);
    }

    public void preWindowOpen() {
        IWorkbenchWindowConfigurer configurer =
            getWindowConfigurer();
        configurer.setInitialSize(new Point(800, 600));
        configurer.setShowPerspectiveBar(true);
        configurer.setShowCoolBar(true);
        configurer.setShowStatusLine(true);
        configurer.setShowFastViewBars(true);
        configurer.setShowProgressIndicator(true);
        configurer.setTitle("Project Planner");
    }
}
```

### 4.3.3 Aktionsleisten erzeugen

Mit Hilfe der Methode `createActionBarAdvisor()` des `WorkbenchWindowAdvisor` werden `ActionBarAdvisor`-Implementierungen in die Workbench eingeführt. Dessen Methode `fillActionBars()` mit den Untermethoden `fillMenuBar()`, `fillCoolBar()` und `fillStatusLine()` ist für die Konstruktion der verschiedenen Aktionsleisten (Menü,

Werkzeuggeste, Statuszeile) zuständig. Dabei besteht auch die Möglichkeit, mit Hilfe von Separatoren Andockpunkte für spätere Erweiterungen zu schaffen.<sup>1</sup>

Die Methode `registerAction()` erlaubt die Zuordnung eines Aktionsobjektes zu einer Identifikation. Mit `getAction()` kann dann später die Registratur abgefragt werden. Das Registrieren hat (neben dieser Aufbewahrungsfunktion) auch den Zweck, die Aktionen auch dem Mechanismus für die Vergabe von Tastaturkürzeln (*Key Binding*) bekannt zu machen. Zusätzlich ruft diese Methode intern beim `IActionBarConfigurer` die Methode `registerGlobalAction()` auf, um die Aktion global zu machen. Damit können später einzelne Workbench-Komponenten wie Views und Editoren ihre jeweiligen konkreten lokalen Aktionen mit der globalen Aktion verknüpfen, indem sie diese unter der ID der globalen Aktion bei ihrer jeweiligen `IActionBars`-Instanz registrieren (siehe Abschnitt 3.5.9). Durch diese gemeinsame Nutzung einer globalen Aktion wird vermieden, dass die gleichen Aktionen mehrfach im Menü oder auf der Werkzeuggeste erscheinen.

*Aktionen registrieren*

Der `IActionBarConfigurer`, den man mit der Methode `getActionBarConfigurer()` erhält, erlaubt außerdem den Zugriff auf die verschiedenen Manager für Menü, Werkzeuggeste und Statuszeile. Über diese Manager können dann das Menü, die Werkzeuggeste und die Statuszeile aufgebaut werden, wie unten im Beispiel gezeigt.

*Aktionsleisten*

Schließlich gibt die `ActionBarAdvisor`-Methode `isApplicationMenu()` an, wie sich die Anwendung beim *OLE-In-Place-Editing* (nur in Windows-Plattformen) verhält. Wird hier der Wert `true` zurückgegeben, bleibt ihr Menü im Fenster erhalten, beim Wert `false` (Standard) wird das Menü aus dem eingebetteten Fenster entfernt.

## Beispiel

Bereits generiert wurde für uns die Klasse `ApplicationActionBarAdvisor`. Als Beispiel zeigen wir hier den Aufbau einer Menüleiste, die aus folgenden Menüpunkten besteht: *File>Print*, *File>Exit*, *Window>Open Perspective*, *Edit>Undo*, *Edit>Redo*, *Window>Preferences*, *Help>Welcome*, *Help>Help Contents* und *Help>About*. (Wie der Inhalt des *About*-Dialogs konfiguriert werden kann, erläutern wir im nächsten Kapitel.) Alle diese Aktionen sind Standardaktionen der Eclipse-Plattform. Sie stehen in der Klasse `ActionFactory` abrufbar bereit – mit Ausnahme der Funktion *Open Perspective*, die keine

---

1. Grundsätzlich gilt in Eclipse: Neue Menüs werden mit Java-Programmierung erzeugt, wobei diese Menüs Andockpunkte definieren können. Später können solche Menüs dann per Manifest-Eintrag erweitert werden.

Aktion, sondern ein `IContributionItem` ist, den wir von der Klasse `ContributionItemFactory` erhalten. Wir holen uns zunächst in der Methode `makeActions()` Instanzen dieser Aktionen und registrieren sie mit dem `ActionBarAdvisor`. Im Falle der Aktion `INTRO` ist eine Sonderaktion notwendig. Diese Aktion kann nur erfolgreich erzeugt werden, wenn Willkommenseiten definiert sind. Das ist derzeit noch nicht der Fall, aber für später vorgesehen. Wir führen deshalb zuvor eine Abfrage beim `IntroManager` durch, den wir bei der `Workbench` erhalten.

In der Methode `fillMenuBar()` werden dann Menütitel angelegt und mit den abgespeicherten Aktionen aufgefüllt. An verschiedenen Punkten der Menüs legen wir benannte Separatoren an. Diese dienen als spätere Andockpunkte, falls zusätzliche Plugins das Menü erweitern wollen. Dabei benutzen wir die Standardpfadbezeichnungen für Menüs, so wie sie im Interface `IWorkbenchActionConstants` definiert sind. Der `IContributionItem PERSPECTIVES_SHORTLIST` wird nicht direkt zum Fenstermenü hinzugefügt. Dies wäre zwar auch möglich, es würden jedoch alle in der *Shortlist* enthaltenen Perspektiven direkt in das Menü eingefügt. Stattdessen erzeugen wir mit einem neuen `MenuManager` ein kaskadierendes Untermenü, dem wir dann die Perspektiven-Shortlist hinzufügen.

Ganz verläuft auch der Aufbau der Werkzeugleiste. Hier werden allerdings statt `MenuManagern` `ToolBarManager` verwendet, und die Definition von Andockpunkten erfolgt mit `GroupMarker`-Instanzen. Wie Sie sehen, können auch auf der Statuszeile Aktionen platziert werden.

```
package com.bdaum.planner;
import org.eclipse.jface.action.GroupMarker;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.action.IContributionItem;
import org.eclipse.jface.action.ICoolBarManager;
import org.eclipse.jface.action.IMenuManager;
import org.eclipse.jface.action.IStatusLineManager;
import org.eclipse.jface.action.MenuManager;
import org.eclipse.jface.action.Separator;
import org.eclipse.jface.action.ToolBarManager;
import org.eclipse.ui.IWorkbenchActionConstants;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.actions.ActionFactory;
import org.eclipse.ui.actions.ContributionItemFactory;
import org.eclipse.ui.application.ActionBarAdvisor;
import org.eclipse.ui.application.IActionBarConfigurer;

public class ApplicationActionBarAdvisor extends
    ActionBarAdvisor {

    private IContributionItem perspShortList;
```

```
public ApplicationActionBarAdvisor(
    IActionBarConfigurer configurer) {
    super(configurer);
}

@Override
protected void makeActions(IWorkbenchWindow window) {
    register(ActionFactory.PRINT.create(window));
    register(ActionFactory.QUIT.create(window));
    register(ActionFactory.UNDO.create(window));
    register(ActionFactory.REDO.create(window));
    register(ActionFactory.ABOUT.create(window));
    register(ActionFactory.RESET_PERSPECTIVE.create(window));
    register(ActionFactory.HELP_CONTENTS.create(window));
    register(ActionFactory.PREFERENCES.create(window));
    if (window.getWorkbench().getIntroManager().hasIntro())
        register(ActionFactory.INTRO.create(window));
    register(ActionFactory.OPEN_NEW_WINDOW.create(window));
    perspShortList =
        ContributionItemFactory.PERSPECTIVES_SHORTLIST
            .create(window);
}

@Override
protected void fillMenuBar(IMenuManager menuBar) {
    // File
    MenuManager fileMenu = new MenuManager("&File",
        IWorkbenchActionConstants.M_FILE);
    menuBar.add(fileMenu);
    fileMenu.add(new Separator(
        IWorkbenchActionConstants.FILE_START));
    fileMenu.add(getAction(ActionFactory.PRINT.getId()));
    fileMenu.add(getAction(ActionFactory.QUIT.getId()));
    fileMenu.add(new Separator(
        IWorkbenchActionConstants.FILE_END));
    // Edit
    MenuManager editMenu = new MenuManager("&Edit",
        IWorkbenchActionConstants.M_EDIT);
    menuBar.add(editMenu);
    editMenu.add(new Separator(
        IWorkbenchActionConstants.EDIT_START));
    editMenu.add(getAction(ActionFactory.UNDO.getId()));
    editMenu.add(getAction(ActionFactory.REDO.getId()));
    editMenu.add(new Separator(
        IWorkbenchActionConstants.EDIT_END));
}
```

```

// Window
MenuManager windowMenu = new MenuManager("&Window",
    IWorkbenchActionConstants.M_WINDOW);
menuBar.add(windowMenu);
MenuManager perspMenu = new MenuManager(
    "&Open Perspective");
windowMenu.add(perspMenu);
perspMenu.add(perspShortList);
windowMenu.add(getAction(ActionFactory.RESET_PERSPECTIVE
    .getId()));
windowMenu.add(new Separator(
    IWorkbenchActionConstants.WINDOW_EXT));
windowMenu.add(getAction(ActionFactory.PREFERENCES
    .getId()));
// Help
MenuManager helpMenu = new MenuManager("&Help",
    IWorkbenchActionConstants.M_HELP);
menuBar.add(helpMenu);
helpMenu.add(new Separator(
    IWorkbenchActionConstants.HELP_START));
IAction introAction = getAction(ActionFactory.INTRO
    .getId());
if (introAction != null)
    helpMenu.add(introAction);
helpMenu.add(getAction(ActionFactory.HELP_CONTENTS
    .getId()));
helpMenu.add(new Separator("update")); // für
// Update-Aktionen
helpMenu.add(new Separator(
    IWorkbenchActionConstants.HELP_END));
helpMenu.add(getAction(ActionFactory.ABOUT.getId()));
}

@Override
protected void fillCoolBar(ICoolBarManager coolBar) {
    ToolBarManager fileTools = new ToolBarManager();
    coolBar.add(fileTools);
    fileTools.add(getAction(ActionFactory.PRINT.getId()));
    ToolBarManager editTools = new ToolBarManager();
    coolBar.add(editTools);
    editTools.add(getAction(ActionFactory.UNDO.getId()));
    editTools.add(getAction(ActionFactory.REDO.getId()));
    coolBar.add(new GroupMarker(
        IWorkbenchActionConstants.MB_ADDITIONS));
}

```

```
@Override
protected void fillStatusLine(IStatusLineManager statusLine) {
    statusLine.addAction(ActionFactory.OPEN_NEW_WINDOW
        .getId());
}
}
```

#### 4.3.4 Das Anwendungs-Plugin der Beispielanwendung

Nachdem wir den RCP-Programmrahmen für den Projektplaner aufgebaut und konfiguriert haben, können wir uns um die Plugin-Klasse `PlannerActivator` kümmern. Da diese Klasse für den Lebenszyklus des Anwendungs-Plugins zuständig ist, ist sie zugleich für den Lebenszyklus der gesamten Anwendung zuständig, also der geeignete Platz, um Zustandsdaten über die gesamte Sitzung zu speichern. Ein solches Datum ist der Name des angemeldeten Benutzers. Später kommen zu den Aufgaben dieser Klasse noch das Hoch- und Runterfahren der Datenbank und die Verwaltung von Datenbanksitzungen hinzu.

Doch zunächst zur Benutzerverwaltung. Jeder Benutzer wird durch eine `User`-Instanz repräsentiert, welche Namen, Passwort und die Rollen des Benutzers speichert.

##### Die Klasse `User`

Die Klasse `User` implementiert das Datenmodell der Projektplanerbenutzer. Jeder Benutzer hat einen Namen, ein Passwort und die ihm zugeordneten Rollen. In sicherheitskritischen Fällen sollte man das Passwort verschlüsselt abspeichern.

```
package com.bdaum.planner;
import java.util.List;

public class User {
    String name;
    String password;
    List<String> roles;
}
```

Zu diesen Feldern generieren Sie noch mit `Source>Generate Getters and Setters...` und `Source>Generate Constructor using Fields...` die `get...()`- und `set...()`-Zugriffsmethoden und einen Konstruktor.

In der Klasse `PlannerActivator` definieren wir dann ein entsprechendes `user`-Feld mit seinen Zugriffsmethoden. Für Testzwecke weisen wir zunächst jedem Benutzer eine Administratorrolle zu und verzichten auf eine Autorisierung:

```

// Aktueller Benutzer
private User user;

public User getUser() {
    return user;
}

public User login(String username, String password) {
    List<String> roles = new ArrayList<String>(1);
    roles.add("Administrator");
    user = new User(username, password, roles);
    return user;
}

```

Nun können wir darangehen, die schon weiter oben per *QuickFix* angelegte Methode `getInitialPerspectiveId()` zu vervollständigen. Dazu werten wir in der Methode die in der aktuellen User-Instanz gespeicherten Rollen aus. Wir verwenden die erste definierte Rolle und suchen dann die passende Perspektive dazu.

```

/**
 * Liefert die ID der rollenspezifischen Anfangsperspektive
 * @return – die ID der Anfangsperspektive
 */
public String getInitialPerspectiveId() {
    if (user != null) {
        String role = (user.roles != null) ? user.roles.get(0)
            : null;
        if (role != null)
            return getInitialPerspectiveId(role);
    }
    return null;
}

```

### Die Klasse Perspective

In der Regel benötigt jede Rich-Client-Anwendung mindestens eine Perspektive. Deshalb wurde auch beim Anlegen des Projekts `com.bdaum.planner` die Klasse `Perspective` generiert und am Erweiterungspunkt `org.eclipse.ui.perspectives` registriert. Dort können wir auch einen anderen Perspektivennamen vergeben, z.B. »Log-in«, denn der generierte Name *RCP-Perspective* ist etwas zu generisch.

*Beispiel*

Grundsätzlich soll aber in unserer Beispielanwendung ein Benutzer in verschiedenen Rollen arbeiten können. Zusätzlich besteht die Möglichkeit, einer Rolle mehrere Perspektiven zuzuordnen. Damit benötigt der Benutzer die Möglichkeit der Perspektivauswahl. Die besteht automatisch über die Perspektivenleiste, ist aber umständlich. Betätigt der

Benutzer die Taste *Open Perspective*, so sieht er nur einen einzigen Menüpunkt *Other*. Erst beim Aufruf dieser Menüfunktion erhält er einen Auswahldialog für die ihm zur Verfügung stehenden Perspektiven. Schöner wäre es, wenn diese Perspektiven in Form von *Shortcuts* gleich angeboten würden.

Dabei scheidet allerdings die statische Definition solcher *Shortcuts* in den jeweiligen Plugin-Manifesten mit Hilfe des Erweiterungspunktes `org.eclipse.ui.perspectiveExtensions` aus. Diese Art der Definition würde voraussetzen, dass das jeweilige Plugin vorab weiß, welche Perspektiven zur Verfügung stehen. Dies ist bei wechselnden Benutzern jedoch nicht gegeben.

Alternativ können jedoch in den jeweiligen `PerspectiveFactory`-Instanzen mit Hilfe der Methode `addPerspectiveShortcut()` solche Perspektiven-Shortcuts gesetzt werden. Damit ergibt sich die Möglichkeit, die Erweiterungs-Registrierung auszuwerten und je nach registrierten Perspektiven die nötigen Shortcuts zu setzen.

*Shortcuts anlegen*

```
package com.bdaum.planner;
import org.eclipse.core.runtime.IConfigurationElement;
import org.eclipse.core.runtime.IExtensionRegistry;
import org.eclipse.core.runtime.InvalidRegistryObjectException;
import org.eclipse.core.runtime.Platform;
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

public class Perspective implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
        fillPerspectiveMenu(layout);
    }

    public void fillPerspectiveMenu(IPageLayout layout) {
        IExtensionRegistry registry =
            Platform.getExtensionRegistry();
        try {
            // Alle Konfigurationselemente
            for (IConfigurationElement conf : registry
                .getConfigurationElementsFor(
                    "org.eclipse.ui.perspectives"))
                // Perspektiven genauer untersuchen
                if (conf.getName().equals("perspective")) {
                    String perspId = conf.getAttribute("id");
                    if (perspAllowedForUser(perspId,
                        PlannerActivator.getDefault().getUser()))
                        layout.addPerspectiveShortcut(perspId);
                }
        }
    }
}
```

```

        } catch (InvalidRegistryObjectException e) {
            // Ignorieren
        }
    }

    /**
     * Prüft, ob Perspektive für Benutzer zugelassen ist.
     * @param perspId – Perspektive-ID
     * @return – true, wenn Perspektive für aktuellen Benutzer
     *         zugelassen ist.
     */
    protected static boolean perspAllowedForUser(
        String perspId, User user) {

        if (user == null)
            return false;
        IExtensionRegistry registry =
            Platform.getExtensionRegistry();
        try {
            // Alle Konfigurationslemente
            for (IConfigurationElement role : registry
                .getConfigurationElementsFor(
                    PlannerActivator.PLUGIN_ID, "roles"))
                for (IConfigurationElement persp : role
                    .getChildren("perspective"))
                    // Assoziation für gesuchte Perspektive
                    if (perspId.equals(persp.getAttribute("id"))
                        &&
                        // Prüfen, ob aktueller Benutzer
                        // diese Rolle besitzt
                        user.roles.contains(
                            role.getAttribute("name")))
                        return true;
        } catch (InvalidRegistryObjectException e) {
            // Ignorieren
        }
        return false;
    }
}

```

Auf diese Art kann man also die Perspektivenmenüs dynamisch gestalten. In Abschnitt 4.3.1 hatten wir ja bereits veranlasst, dass alle Perspektiven zurückgesetzt werden, wenn eine Veränderung am Erweiterungspunkt `roles` eintritt. Dieses Zurücksetzen bewirkt eine erneute Ausführung der Methode `createInitialLayout()` und somit eine erneute Untersuchung der `ExtensionRegistry` auf zum Benutzer passende Perspektiven.

Da diese Funktionalität auch in allen zusätzlichen Perspektiven benötigt wird, müssen die zusätzlichen `PerspectiveFactory`-Implementierungen als Unterklassen der Klasse `com.bdaum.planner.Perspective` angelegt werden, welche die hier definierte Methode `createInitialLayout()` erweitern. Dazu erlauben wir den anderen Plugins den Zugriff auf das Package `com.bdaum.planner`, indem wir dieses Package auf der *Runtime*-Seite des Manifesteditors zur Liste der exportierten Packages hinzufügen.

Nun kann die bereits im Plugin `com.bdaum.planner.administrator` implementierte Klasse `PerspectiveFactory` wie folgt abgeändert werden:

```
public class PerspectiveFactory extends Perspective {  
  
    @Override  
    public void createInitialLayout(IPageLayout layout) {  
        super.createInitialLayout(layout);  
        layout.addView("com.bdaum.planner.admin.views.UserView",  
            IPageLayout.TOP, 0.5f, IPageLayout.ID_EDITOR_AREA);  
        layout.addView("org.eclipse.pde.runtime.LogView",  
            IPageLayout.BOTTOM, 1.0f,  
            IPageLayout.ID_EDITOR_AREA);  
        layout.setEditorAreaVisible(false);  
    }  
}
```

Beim Testen werden Sie allerdings feststellen, dass Änderungen an einer `PerspectiveFactory` beim erneuten Aufruf des Programms nicht greifen. Der Grund liegt darin, dass die Workbench, wenn entsprechend konfiguriert (siehe Abschnitt 4.3.1), den aktuellen Zustand der Perspektiven abspeichert und beim nächsten Start wiederherstellt. Man muss also nach einer Änderung die Funktion *Reset Perspective* aufrufen, um eine erneute Ausführung der Methode `createInitialLayout()` zu erzwingen. In Abschnitt 4.3.3 hatten wir deshalb diese Funktion zum Menü hinzugefügt. Sie ist natürlich auch für den Anwender wichtig, um eine Perspektive wieder in den Grundzustand zu bringen.

Testen

### Perspektivenverwaltung

Ein weiteres Problem taucht beim Benutzerwechsel auf, denn dann zeigt die Perspektivenleiste des Workbench-Fensters eventuell offene Perspektiven, die für diesen Benutzer gar nicht erlaubt sind. Wir müssen also beim Start der Anwendung solche Perspektiven schließen.

Eclipse kennt zwei Orte zur Perspektivenverwaltung: die `PerspectiveRegistry`, die man von der Workbench-Instanz erhält, und die `WorkbenchPage`, bei der Perspektiven geschlossen und auch gesetzt wer-

den können. Hier genügt uns die letztere: Wir prüfen für alle offenen Perspektiven, ob sie für den betreffenden Benutzer erlaubt ist. Wenn nicht, wird sie geschlossen. Der geeignete Zeitpunkt ist nach dem Öffnen des Workbench-Fensters. Dieses Ereignis kann im `ApplicationWorkbenchWindowAdvisor` durch Überschreiben der Methode `postWindowOpen()` abgefangen werden.

```
@Override
public void postWindowOpen() {
    User user = PlannerActivator.getDefault().getUser();
    IWorkbenchPage activePage =
        getWindowConfigurer().getWindow().getActivePage();
    if (activePage != null)
        // Nicht benötigte Perspektiven schließen
        for (IPerspectiveDescriptor perspectiveDescriptor :
            activePage.getOpenPerspectives())
            if (!Perspective.perspAllowedForUser(
                perspectiveDescriptor.getId(), user))
                activePage.closePerspective(
                    perspectiveDescriptor, false, false);
}
```

### 4.3.5 Montieren einer RCP-Anwendung

Im Moment besteht unsere Anwendung nur aus einem einzigen Plugin, dem Anwendungs-Plugin `com.bdaum.planner`. Um die übrigen Plugins in die Anwendung zu integrieren, sind keinerlei Programmierarbeiten notwendig. Es reicht, diese Plugins der Launch-Konfiguration hinzuzufügen.

Dazu rufen Sie die Funktion *Run>Open Run Dialog* auf, wählen die Launch-Konfiguration `com.bdaum.planner` aus und gehen dann zur Seite *Plug-ins*. Dort markieren Sie auch die Plugins `com.bdaum.planner.help`, `com.bdaum.planner.admin`, `com.bdaum.planner.admin.help` und `com.bdaum.planner.core`.

Das Administrations-Plugin benötigt als zusätzliche Infrastruktur das Plugin `org.eclipse.pde.runtime`, welches für den *Error-Log-View* verantwortlich ist. Markieren Sie deshalb auch dieses Plugin.

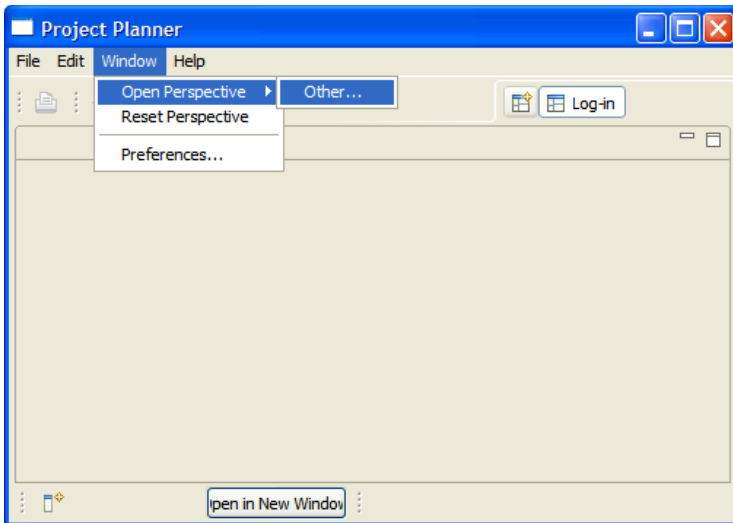
Hilfesystem

Für eine erfolgreiche Ausführung der Hilfsfunktionen fehlt allerdings noch das Hilfesystem, das ebenfalls nicht standardmäßig zur Rich-Client-Plattform gehört. Markieren Sie deshalb noch die Plugins `org.apache.lucene`, `org.eclipse.help.appserver`, `org.eclipse.help.base`, `org.eclipse.help.ui`, `org.eclipse.help.webapp` und `org.eclipse.ui.forms`, die damit zur Ablaufplattform hinzugefügt werden. Das noch in Version 3.2 verwendete Plugin `org.eclipse.tomcat` wird nicht mehr benötigt, da in

Version 3.3 nun der Server *Jetty* diese Aufgabe übernimmt. Dieser ist im Plugin `org.mortbay.jetty` enthalten und gehört bereits zur Equinox-Grundausrüstung.

Anschließend betätigen Sie noch die Taste *Add Required Plug-ins*. Damit werden die in den OSGi-Manifesten der markierten Plugins eingetragenen Plugin-Abhängigkeiten ausgewertet, die zusätzlich benötigten Plugins ermittelt und zur Ablaufplattform hinzugefügt. Nun können Sie mit der Taste *Run* den neuen Test starten.

*Abhängigkeiten*



**Abb. 4-1** Der nun mit einem Menü versehene Projektplaner. Einige Menüaktionen können schon ausprobiert werden.

Im nächsten Kapitel wollen wir diskutieren, welche Möglichkeiten zur Produktgestaltung in der Eclipse-RCP bestehen.

### 4.3.6 Aufgabe

In Abschnitt 4.3.1 hatten wir gelernt, dass die Workbench angewiesen werden kann, ihren Zustand für die nächste Sitzung zu sichern. Finden Sie heraus, wo diese Zustandsdaten gesichert werden, und untersuchen Sie, welche Informationen gesichert werden.