

## 16 Das Graphical Editing Framework

In einigen früheren Kapiteln hatten wir bereits auf das *Graphical Editing Framework* (GEF) Bezug genommen. Das GEF ist ein eclipse.org-Projekt, welches einen Programmrahmen für die Entwicklung von Diagrammeditoren und -viewern bereitstellt. GEF wurde ursprünglich für die Entwicklung von Modellierungswerkzeugen geschaffen. So wurden verschiedene UML-Editoren mit Hilfe des GEF implementiert. Diese Vergangenheit hängt dem GEF immer noch etwas an. In Projekten aus anderen Anwendungsbereichen kann man leicht an die Grenzen des GEF stoßen. Es empfiehlt sich also, vor dem Beginn einer Implementierung erst einmal zu prüfen, ob GEF den Anforderungen des Projektes entsprechen kann. Ist das der Fall, kann man sich mit dem Einsatz von GEF allerdings viel Arbeit sparen. Im Rahmen unserer Beispielanwendung werden wir einige der Grenzen von GEF ausloten.

Wir werden hier nur die Grundzüge von GEF streifen können, wobei wir uns auf Eigenschaften konzentrieren, die besonders für die Entwicklung von RCP-Anwendungen (und insbesondere im Datenbankumfeld) wichtig sind. Für weitergehende Informationen über GEF möchte ich auf die Literatur verweisen. Eine gute Einführung in GEF (allerdings bezogen auf Version 2.1) ist das GEF-Redbook von IBM [Moore2004]. Auf dem Web und in der GEF-Hilfe findet man außerdem eine Reihe von Tutorials und von Beispielprojekten.

### 16.1 Stärken und Schwächen von GEF

GEF stellt höherwertige Komponenten zur Verfügung als die meisten grafischen Frameworks. So bietet es z.B. Viewer und Editoren mit:

- grafischer Repräsentation beliebiger Datenmodelle
- Unterstützung einer Werkzeugpalette für das Editieren
- Unterstützung von Zoom-Aktionen

*Stärken*

- Unterstützung von Rulern (Linealen) und Guides (Hilfslinien)
- Unterstützung von Rastern und Einrastpunkten
- mehreren Layern (Ebenen)
- Unterstützung für Verbindungslinien und automatisches Routing
- Cut & Paste, Drag & Drop
- Druckausgabe
- reichhaltigem Arsenal an vordefinierten Aktionen
- Unterstützung für Antialiasing, Gradienten und Blends

*Schwächen* Auf der Negativseite müssen vor allem genannt werden:

- Steile Lernkurve. Es sind jedoch einige Beispielapplikationen vorhanden. Eine große Open-Source-Applikation, die mit dem GEF realisiert wurde, ist der *Eclipse Visual Editor* (VE).
- Viele Konzepte sind nicht generisch genug, um für ein breites Anwendungsspektrum einsetzbar zu sein.

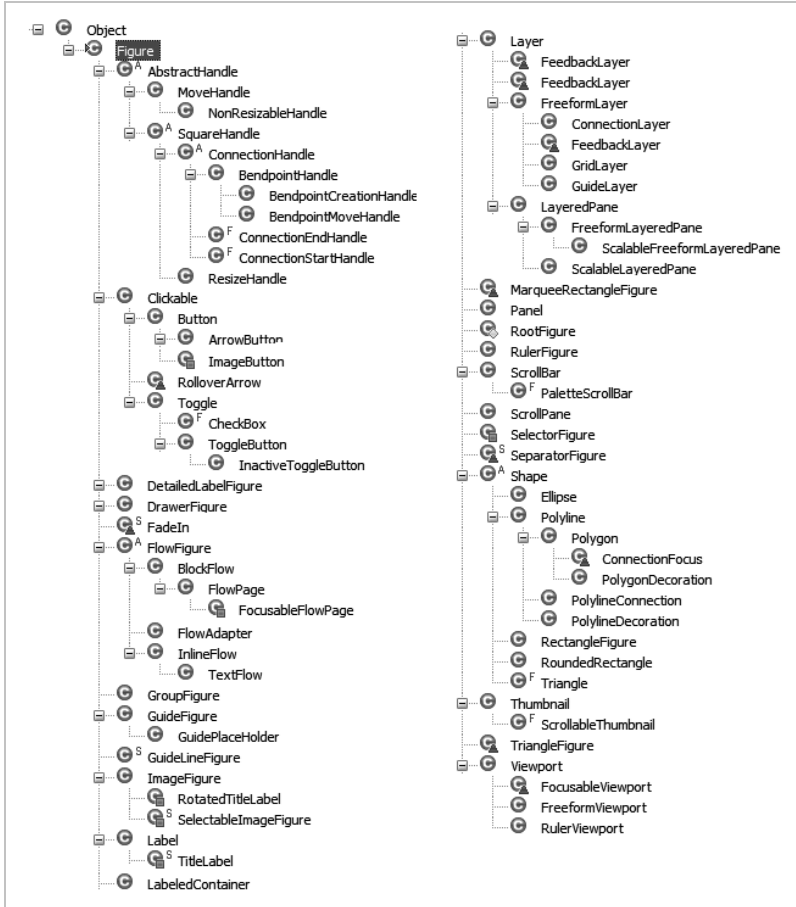
## 16.2 Die GEF-Architektur

Das GEF setzt nicht direkt auf der SWT-Grafik (siehe Abschnitt 9.1.6) auf, sondern auf einer höheren Grafiksicht – der *Draw2D*-Schicht. Dessen Bausteine realisieren einen leichtgewichtigen Überbau über das SWT. Anstelle mit Zeichenoperationen auf einem Grafikkontext und mit *PaintListern* arbeiten zu müssen, kann man hier direkt mit grafischen Objekten (z.B. Panel, Rechteck, Hilfslinie oder Thumbnail) hantieren.

*Draw2D* Dieses grafische Instrumentarium des Draw2D wird vom GEF für die Implementierung des Modell-View-Controller-Entwurfsmusters (MVC) benutzt. Dabei stellt das GEF nur die Unterstützung für die Realisierung der Viewer und Controller bereit – bezüglich des Modells ist es völlig agnostisch. Das Modell muss von der jeweiligen Anwendung bereitgestellt werden.

### 16.2.1 Draw2D

*Figure* Alle grafischen Objekte des Draw2D basieren auf der Klasse *Figure* (beschrieben durch das Interface *IFigure*), welche die gemeinsamen Eigenschaften dieser Objekte definiert. Insbesondere ist hier schon die Unterstützung für die Ereignisverarbeitung – Maus- und Tastaturereignisse sowie Ereignisse bei der Änderung von Eigenschaften – definiert. Alle Grafiken im Draw2D sind hierarchisch organisiert – folglich gibt es auch in der Klasse *Figure* die Methoden *getChildren()* und *getParent()*.



**Abb. 16-1** Die Klasse Figure sowie ihre konkreten und abstrakten Unterklassen. Einige dieser Klassen stehen nur im GEF zur Verfügung.

An der Wurzel einer Grafik liegt in der Regel ein Behälter, z.B. ein Panel. Diesem Wurzelobjekt können mit der Methode `add()` Kindobjekte hinzugefügt werden. Swing-Programmierer werden sich freuen, denn dort funktioniert das genauso. Im SWT dagegen werden Kindelemente mit `new` erzeugt, wobei der Behälter im Konstruktor übergeben werden muss. Der Vorteil bei der `add()`-Methode liegt darin, dass man hier Grafikbäume auch wieder umbauen kann: Mit `remove()` kann man ein Kindelement aus einem Behälter entfernen und anschließend mit `add()` einem anderen Behälter zuordnen.

Figure-Objekte haben eine Reihe weiterer gemeinsamer Features:

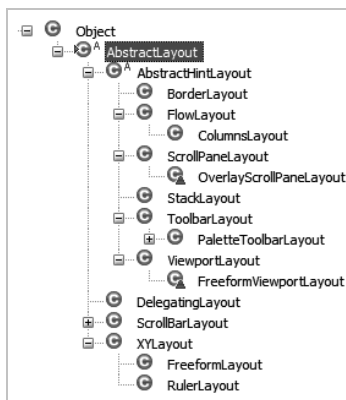
- Eine Reihe von Attributen bestimmt den Zustand eines Figure-Objektes. Mit `setValid()` kann ein Objekt auf gültig oder ungültig

gesetzt werden, mit `setVisible()` auf sichtbar oder unsichtbar und mit `setEnabled()` auf aktiv oder inaktiv. Die Methode `setOpaque()` bestimmt, ob das Objekt transparent oder undurchsichtig gezeichnet wird. Die Methode `setRequestFocusEnabled()` legt fest, ob ein Objekt den Fokus erhalten kann, die Methode `setFocusTraversable()`, ob das Objekt auf Grund eines Traverse-Ereignisses (z.B. TAB-Taste) den Fokus erhalten kann.

- Kindelemente werden nicht über die Grenzen ihrer Elternelemente hinaus gezeichnet, sondern an den Grenzen abgeschnitten.
- Figure-Objekte können mit verschiedenen Rahmen umgeben werden. Ähnlich wie in Swing gibt es eine Vielzahl vordefinierter Rahmen.
- Figure-Objekte können mit Hilfe von `IConnection`-Objekten verbunden werden (die Standardimplementierung ist `PolylineConnection`). In GEF wird das ausgenutzt, um Verbindungslinien und Pfeile zwischen Objekten zu zeichnen.
- Figure-Objekte können mit `setCursor()` definieren, welches Aussehen der Mauszeiger annehmen soll, wenn er sich über dem Objekt befindet.
- Layer sind eine spezielle Art transparenter Container, mit denen grafische Schichten wie Folien übereinander angeordnet werden können.
- Figure-Objekte können mit Hilfe einer `PrintFigureOperation` auf Drucker ausgegeben werden.

#### Layout

Ähnlich wie in Swing und SWT erfolgt unter Draw2D die Positionierung von Grafikobjekten in ihrem Behälter mit Hilfe von Layoutmanagern. Jedes Figure-Objekt kann mit einem Layoutmanager versehen werden, der die Kindelemente nach vorgegebenen Regeln selbsttätig im Elternobjekt anordnet. Abbildung 16–2 zeigt die Hierarchie des reichhaltigen Layoutmanager-Arsenals von Draw2D.



**Abb. 16–2**

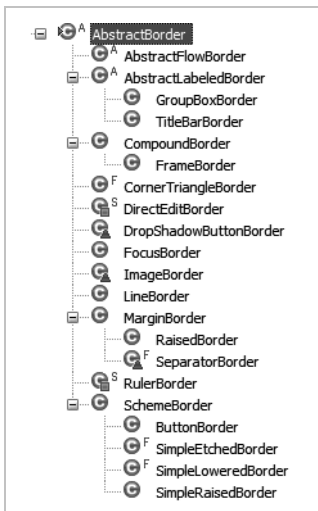
Die Layoutmanager des GEF und des Draw2D

Auch wenn Sie Objekte an einer definierten Position in einem Behälter positionieren wollen, sollten Sie im Rahmen von GEF niemals versuchen, Figure-Objekte manuell zu positionieren, sondern stattdessen lieber das `XYLayout` verwenden. Dabei kann mit `setConstraint()` die Position und die Größe des Kindelementes angegeben werden:

```
Panel panel = new Panel();
panel.setLayoutManager(new XYLayout());
Label label = new Label();
label.setText("Hello Draw2D");
panel.add(label);
panel.setConstraint(label, new Rectangle(x, y, width, height));
```

Ähnlich groß ist die Vielfalt bei den Umrahmungen, die mit `setBorder()` gesetzt werden können (siehe Abb. 16–3). So kann mit `FrameBorder` ein Titelbalken gesetzt werden, `GroupBoxBorder` ähnelt einem SWT-Group-Widget, `LineBorder` umrahmt mit einer einfachen Linie, und bei `MarginBorder` können die Abstände zwischen Objekt und Umrahmung spezifiziert werden.

*Umrahmungen*



**Abb. 16–3**

Die Hierarchie der Draw2D-Umrahmungen

### 16.2.2 GEF aus der Vogelperspektive

Das GEF setzt über diese grafische Schicht noch einmal weitere Abstraktionsschichten. So werden grafische Objekte im GEF mit Hilfe von `EditParts` realisiert. Dabei implementiert jede `EditPart`-Klasse ihre Visualisierung mit Hilfe von `Draw2D-Figures`. Diese `EditParts` werden von `EditPartFactory`-Instanzen erzeugt, die je nach Typ des abzubil-

*EditPart*

denden Objekts aus dem Datenmodell die passende `EditPart`-Instanz liefern.

*Viewer* GEF stellt – ähnlich wie das `JFace` – `Viewer` zur Verfügung (so z.B. den `ScrollingGraphicalViewer`), welche die `Viewer`-Komponente des MVC-Entwurfsmusters realisieren. Jedem `Viewer` wird eine `EditPartFactory` zugeordnet und erledigt ähnliche Aufgaben wie ein `LabelProvider` bei `JFace`-Viewern, nämlich die Herstellung einer konkreten Präsentation für die Elemente des abstrakten Datenmodells.

*Editor* GEF-`Viewer` können bei der Konstruktion von Views und Editoren verwendet werden. GEF stellt bereits einige vorgefertigte Editoren zur Verfügung, so z.B. den `GraphicalEditor`, den `GraphicalEditorWithPalette` und den `GraphicalEditorWithFlyoutPalette`. Diese Editoren sind vollgültige Eclipse-Editoren und können von der Eclipse-Menüleiste aus gesteuert werden. Die beiden letzteren Editoren verfügen außerdem über eine konfigurierbare Palette mit verschiedenartigen Werkzeugen.

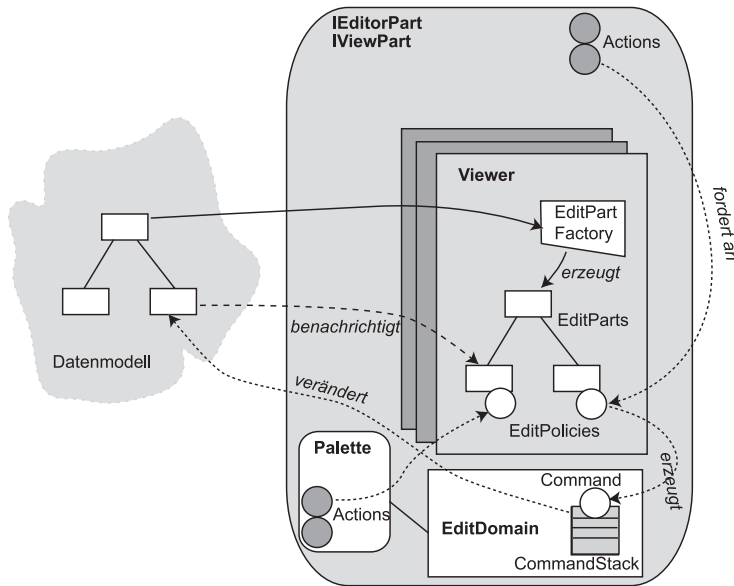
### 16.3 Modell, Viewer, Controller

Hatten wir aber schon bemerkt, dass *Draw2D* vom Konzept stärker an Swing als an SWT orientiert ist, so gilt das auch für die `Viewer` des GEF. Ähnlich wie Swing setzen die `Viewer` nicht direkt auf dem Domänenmodell der Anwendung (wie `JFace`-`Viewer` das tun) auf, sondern verwenden ein eigenes internes Datenmodell, das mit dem Domänenmodell der Anwendung synchronisiert werden muss. Bei GEF enthält dieses interne Datenmodell ausschließlich `EditPart`-Instanzen. Das hat Konsequenzen. Selektionen, die von einem GEF-`Viewer` ausgehen, enthalten `EditPart`-Instanzen, und die Selektion bei GEF-Viewern kann auch nur in Form von `EditParts` gesetzt werden. Beim Austausch von Selektionsereignissen zwischen GEF-Viewern und `JFace`-Viewern ist deshalb ein Übersetzungsvorgang notwendig.

#### 16.3.1 EditParts

GEF-`Viewer` basieren auf der gemeinsamen Superklasse `GraphicalViewers`. Dieser erhält das Domänenmodell über die Methode `setContentts()`. Daraus wird anschließend das interne, aus `EditPart`-Instanzen bestehende Datenmodell mit Hilfe einer `EditPartFactory` erzeugt. Die muss vorher mit `setEditPartFactory()` gesetzt werden. Aus jedem Objekt des Datenmodells fertigt sie eine korrespondierende `GraphicalEditPart`-Instanz an.

Von den `EditPart`-Instanzen zu den Objekten des Domänenmodells zu kommen, ist ziemlich einfach: Man fragt die `EditPart`-Instanz einfach mit Hilfe der Methode `getModel()` nach dem zugeordneten Objekt aus dem Domänenmodell. Umgekehrt ist der Weg über eine Zuordnungstabelle (Map) nötig, die man vom Viewer mit `getEditPartRegistry()` erhält.



**Abb. 16-4** Die Architektur einer typischen GEF-Anwendung

Die Modifikation des Domänenmodells muss über Kommandoobjekte erfolgen. Für jede spezifische Modifikation des Datenmodells durch den Benutzer (z.B. Löschen eines Elements, Einfügen eines neuen Elements) wird ein spezifisches Kommando implementiert. Die Kommandoimplementierung kann natürlich nicht datenmodellagnostisch sein, denn das Kommando muss ja das Datenmodell modifizieren (sofern man diese Aufgabe nicht noch weiter delegiert). GEF stellt eine abstrakte `Command`-Klasse zur Verfügung, auf deren Grundlage Kommandos implementiert werden können. Grundsätzlich gilt, dass jede Kommandoklasse auch die nötige Logik zum Verwerfen des Kommandos (*Undo*) implementieren muss (falls *Undo* unterstützt wird).

*Kommandos*

Umgekehrt ist es für die Visualisierung von Änderungen am Domänenmodell nötig, dass das Datenmodell in der Lage ist, Zustandsänderungen an andere mitzuteilen, d.h., man muss sich beim Datenmodell als Listener, der über Zustandsänderungen informiert wird, registrieren können. Partielle Wiederauffrischung der Grafiken sind über die

refresh()-Methode der EditParts möglich. Ein kompletter Neuaufbau der Grafik kann mit setContents() beim Viewer erreicht werden.

### 16.3.2 Kommandos

Das Kommando-Konzept von GEF ist den in der Eclipse-Plattform implementierten IUndoableOperations ähnlich, aber nicht identisch. Kommando-Instanzen werden auf Anfrage (*Request*) von EditPolicy-Instanzen erzeugt. Solche EditPolicy-Instanzen müssen von den verschiedenen EditParts zur Verfügung gestellt werden. Die erzeugten Kommandos werden ausgeführt und in einem CommandStack gespeichert. Im *Undo*-Falle stehen sie dort zur Verfügung und können die einzelnen *Undo*-Schritte durchführen.

*CommandStack*

Der CommandStack ist in einer sogenannten EditDomain enthalten, welcher die Rolle des Controllers im MVC-Entwurfsmuster zukommt. Außer dem CommandStack verwaltet die EditDomain noch den PaletteViewer und den grafischen Viewer des jeweiligen Editors oder Views. Es ist sogar möglich, mit Hilfe einer EditDomain mehrere grafische Viewer zu verwalten, z.B. in einem MultiPageEditor.

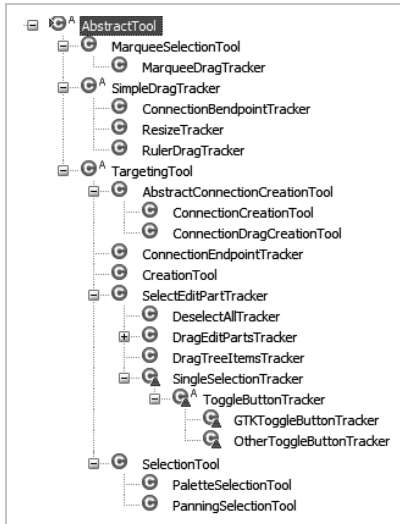
GEF-Kommandos mit Eclipse-Operationen zu integrieren, ist möglich. So kann z.B. ein GEF-Kommando eine IUndoableOperation auslösen, die dann im globalen (und nicht GEF-spezifischen) Kommando-Stack aufbewahrt wird und auch wieder rückgängig gemacht werden kann. Da ein Wiederaufbau der Visualisierung ereignisgesteuert vom Domänenmodell erfolgt, sollte es dabei keine Schwierigkeiten geben. In Abschnitt 16.4.2 zeige ich, wie das geht.

### 16.3.3 Werkzeuge

Der PaletteViewer beherbergt die Palette mit den verschiedenen Werkzeugen. GEF stellt bereits eine ganze Reihe vorgefertigter Grafikwerkzeuge zur Verfügung (siehe Abb. 16–5). Wird ein Werkzeug betätigt, so wird die zugeordnete Aktion ausgeführt. Auch hier stellt GEF im Package org.eclipse.gef.ui.actions ein Arsenal vorgefertigter Aktionen bereit.

*Aktionen und Policies*

Die Aktionen sind dann dafür verantwortlich, Anforderungen (*Requests*) an die jeweiligen EditParts zu stellen (in einigen Fällen werden solche Requests auch direkt von den Tools abgegeben). Die angesprochenen EditParts erzeugen dann auf Grund einer definierten EditPolicy das entsprechende Kommando. Dabei kann ein EditPart verschiedene dieser Policies definieren. Jede Policy wird durch einen String identifiziert, der auch Rolle genannt wird. Die verfügbaren Rollen sind in der Klasse EditPolicy als Konstanten definiert. Die Aus-

**Abb. 16-5**

Die GEF-Werkzeuge sind alle im Package `org.eclipse.gef.tools` enthalten.

wahl der Policy erfolgt dann auf Grund der im *Request* spezifizierten Rolle. Der Vorteil dieses Mechanismus liegt darin, dass die im `EditPart` registrierten Policies dynamisch geändert werden können, also das Verhalten eines `EditParts` von seinem Zustand abhängig gemacht werden kann.

## 16.4 Beispielanwendung: Gantt-Chart mit GEF

Wir wollen nun GEF benutzen, um der Tasks-Perspektive in unserer Beispielanwendung eine Gantt-Grafik hinzuzufügen. Wir werden diese Grafik nicht als `Editor` implementieren, sondern als zusätzlichen `View`.

### 16.4.1 GEF installieren

GEF können Sie sich von `eclipse.org` herunterladen. Da wir eine GEF-Anwendung entwickeln wollen, benötigen wir das SDK und nicht die GEF-*Runtime* aus der Europa-Distribution. Nehmen Sie am besten das All-in-one-Paket. Die GEF-Versionsnummer muss zur Versionsnummer Ihrer Eclipse-Installation passen, also 3.3.0 zu 3.3.0 und 3.3.1 zu 3.3.1.

Entpacken Sie dann die ZIP-Datei in das gleiche Verzeichnis, in das Sie auch Ihre Eclipse-Distribution entpackt hatten. Dann starten Sie Eclipse und gehen zu `Help > Software Updates > Manage Configuration`. Dort expandieren Sie den Eclipse-Knoten und schalten die Option `Show Disabled Features` ein. Anschließend können Sie das `Graphical Editing Framework SDK` auf `Enabled` setzen. Nach einem Neustart können Sie dann mit GEF arbeiten.

### 16.4.2 Der Gantt-View

Zunächst müssen die GEF-Bibliotheken dem in Kapitel 14 entwickelten Plugin `com.bdaum.planner.taskplanner` bekannt gemacht werden. Im Plugin-Manifest fügen Sie das folgende Plugin unter *Dependencies* hinzu:

```
org.eclipse.gef
```

Außerdem ist es erforderlich, die Plugins `org.eclipse.gef` und `org.eclipse.draw2d` zum *Project Planner Platform Feature* hinzuzufügen und im *Project Planner Basic Features* auf der *Dependencies*-Seite des Feature-Manifests die Taste *Compute* zu betätigen.

*Plugin-Manifest*

Dann kann der zusätzliche View im Plugin-Manifest definiert werden. In der Sektion *Extensions* gibt es bereits eine Erweiterung für den Erweiterungspunkt `org.eclipse.ui.views`. Hier deklarieren Sie einen weiteren View:

```
<view category="com.bdaum.planner.viewCategory"
      class="com.bdaum.planner.taskplanner.views.GanttView"
      icon="icons/gantt.GIF"
      id="com.bdaum.planner.taskplanner.views.GanttView"
      name="GANTT"/>
```

Die Klasse `GanttView` wird nach dem Eintrag in das *class*-Attribut mit einem Klick auf *class* erzeugt. In der Klasse definieren wir gleich eine Konstante namens `ID` mit der im Manifest deklarierten View-ID.

#### Die Klasse `PerspectiveFactory`

Vor der weiteren Implementierung der View-Klasse bauen wir noch den neuen View in die Planer-Perspektive ein. Wir platzieren ihn oberhalb des Editorbereichs, also auch oberhalb des Projekt-Views, der unterhalb des (unsichtbaren) Editorbereichs angeordnet war.

*Layout*

```
package com.bdaum.planner.taskplanner;
import org.eclipse.ui.IPageLayout;
import com.bdaum.planner.Perspective;
import com.bdaum.planner.taskplanner.views.GanttView;
import com.bdaum.planner.taskplanner.views.TaskView;

public class PerspectiveFactory extends Perspective {

    @Override
    public void createInitialLayout(IPageLayout layout) {
        super.createInitialLayout(layout);
        // Views hinzu fügen
        layout.addView(GanttView.ID, IPageLayout.TOP, 0.5f,
            layout.getEditorArea());
    }
}
```

```

    layout.addView(TaskView.ID, IPageLayout.BOTTOM, 0.5f,
        layout.getEditorArea());
    // Editorbereich wird nicht benötigt
    layout.setEditorAreaVisible(false);
}
}

```

### Die Klasse GanttView

Der GanttView zeigt – anders als der TaskView – immer nur ein Projekt an. Der GanttView wird nur über zwei Operationen verfügen, nämlich das Löschen von Tasks und den Ausdruck der Grafik. Das anzuzeigende Projekt wird durch die Selektion im TaskView bestimmt. Da der TaskView als SelectionProvider arbeitet und Selektionsereignisse über den globalen SelectionService der Plattform absetzt, muss der GanttView diesen Ereignissen nur zuhören.

Auf der anderen Seite muss auch der TaskView über Selektionsänderungen im GanttView informiert werden, um seine Selektionen mit den Selektionen im GanttView synchronisieren zu können. Wir realisieren das, indem wir den GanttView ähnlich wie den AbstractMasterDetailsView das Interface ISelectionProvider implementieren lassen und beim globalen Selektionsdienst registrieren.

Wie aus Abbildung 16–4 ersichtlich, sollte jeder GEF-Editor oder -View eine EditDomain-Instanz enthalten, welche unter anderem den CommandStack verwaltet. Wir erzeugen hier zunächst eine Instanz einer solchen EditDomain. Später werden wir den noch zu erzeugenden grafischen Viewer mit Hilfe der Methode add() bei dieser EditDomain anmelden.

*EditDomain*

```

package com.bdaum.planner.taskplanner.views;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.persistence.EntityManager;
import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.geometry.Point;
import org.eclipse.gef.EditDomain;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPartFactory;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.GraphicalViewer;
import org.eclipse.gef.commands.CommandStack;

```

```
import org.eclipse.gef.ui.actions.DeleteAction;
import org.eclipse.gef.ui.actions.PrintAction;
import org.eclipse.gef.ui.parts.ScrollingGraphicalViewer;
import org.eclipse.jface.action.IMenuListener;
import org.eclipse.jface.action.IMenuManager;
import org.eclipse.jface.action.IToolBarManager;
import org.eclipse.jface.action.MenuManager;
import org.eclipse.jface.action.Separator;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.ISelectionChangedListener;
import org.eclipse.jface.viewers.ISelectionProvider;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.SelectionChangedEvent;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.swt.events.MouseAdapter;
import org.eclipse.swt.events.MouseEvent;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Menu;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IActionBars;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.ISelectionService;
import org.eclipse.ui.IWorkbenchActionConstants;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.actions.ActionFactory;
import org.eclipse.ui.operations.UndoRedoActionGroup;
import org.eclipse.ui.part.ViewPart;
import com.bdaum.planner.PlannerActivator;
import com.bdaum.planner.core.CoreActivator;
import com.bdaum.planner.core.model.ModelListener;
import com.bdaum.planner.core.model.Project;
import com.bdaum.planner.core.model.Task;

public class GanttView extends ViewPart implements
    ISelectionProvider, ModelListener, ISelectionListener {

    public static final String ID =
        "com.bdaum.planner.taskplanner.views.GanttView";

    // Der grafische Viewer
    private ScrollingGraphicalViewer viewer;
    // Das Datenmodell des anzuzeigenden Projekts
    private Project project;
    // Liste der Instanzen, die über Selektionsänderungen
    // informiert werden wollen
    private List<ISelectionChangedListener>
        selectionChangedListeners =
            new ArrayList<ISelectionChangedListener>(2);
```

```

// Aktion zum Löschen von Tasks
private DeleteAction deleteAction;
// GEF-Druckaktion
private PrintAction printAction;
// Die GEF-Editdomain verwaltet den Zustand des Views
private EditDomain editDomain = new EditDomain();
// Der JPA-Entitymanager
private EntityManager entityManager =
    CoreActivator.getDefault().getEntityManager();
// Undo-Redo-Aktionsgruppe
private UndoRedoActionGroup undoRedoActionGroup;

/**
 * Gibt Ressourcen wieder frei
 */
@Override
public void dispose() {
    if (undoRedoActionGroup != null) {
        undoRedoActionGroup.dispose();
        undoRedoActionGroup = null;
    }
    printAction.dispose();
    deleteAction.dispose();
    super.dispose();
}

```

Da der GanttView als `ISelectionProvider` agiert, muss er die Methoden `getSelection()`, `setSelection()`, `addSelectionChangedListener()` und `removeSelectionChangedListener()` implementieren. Dabei muss, wie oben beschrieben, eine Konversion ausgeführt werden: Während der grafische Viewer mit `EditPart`-Instanzen arbeitet, versenden und empfangen die übrigen Views Objekte des Domänenmodells. Die Umsetzung von `EditParts` in Domänenmodell-Objekte erfolgt in der Methode `getSelection()`. Dort holen wir uns zunächst die aktuelle Selektion vom grafischen Viewer und ersetzen darin jede `EditPart`-Instanz durch das im `EditPart` enthaltene Modellobjekt.

*Ereignisse*

```

// Liefert die aktuelle Selektion
public ISelection getSelection() {
    // Selektion vom Viewer holen
    IStructuredSelection selection =
        (IStructuredSelection) viewer.getSelection();
    // Neue Objektliste aufbauen
    Object[] sel = selection.toArray();
    for (int i = 0; i < sel.length; i++)
        if (sel[i] instanceof EditPart)
            sel[i] = ((EditPart) sel[i]).getModel();
}

```

```

        // Neues Selektionsobjekt erzeugen
        return new StructuredSelection(sel);
    }

```

Die umgekehrte Richtung ist etwas komplizierter, da wir in der Methode `setSelection()` untersuchen müssen, was selektiert wurde. Wurde ein Projekt selektiert, so muss der Viewer eine neue Eingabe erhalten, um das neue Projekt anzuzeigen. Wurden Tasks selektiert, so werden nur die Tasks im Viewer selektiert, die zum ausgewählten Projekt gehören. Die Umsetzung von Task- in EditPart-Instanzen erfolgt anschließend über die EditPart-Registrierung, die wir mit `getEditPartRegistry()` vom Viewer bekommen. Nach dem Setzen einer neuen Viewer-Selektion muss die Aktivierung der Aktionen neu bestimmt werden, was in der Methode `updateActions()` geschieht. Außerdem wird mit der Methode `reveal()` dafür gesorgt, dass das erste ausgewählte Element im sichtbaren Bereich des Viewers liegt.

```

// Setzt die Selektion auf die übergebenen Objekte
public void setSelection(ISelection selection) {
    if (selection instanceof IStructuredSelection) {
        EditPart partToReveal = null;
        Project newProject = null;
        // EditPart-Registrierung
        Map<?, ?> editPartRegistry =
            viewer.getEditPartRegistry();
        // Liste von EditParts aufbauen
        List<Object> selectedParts = new ArrayList<Object>();
        // Alle selektierten Elemente bearbeiten
        Iterator<?> iterator =
            ((IStructuredSelection) selection).iterator();
        while (iterator.hasNext()) {
            Object elem = iterator.next();
            if (elem instanceof Project) {
                // Projekt selektiert, als anzuzeigendes
                // Projekt merken
                if (newProject == null)
                    newProject = (Project) elem;
            } else if (elem instanceof Task) {
                // Task selektiert
                Project taskParent = ((Task) elem).getParent();
                // Gegebenenfalls Projektwechsel durchführen
                if (newProject == null)
                    newProject = taskParent;
            }
        }
    }
}

```

```

        // Nur Tasks des gewählten Projekts
        // selektieren
        if (newProject == taskParent) {
            Object editPart = editPartRegistry.get(elem);
            if (editPart instanceof EditPart) {
                selectedParts.add(editPart);
                if (partToReveal == null)
                    partToReveal = (EditPart) editPart;
            }
        }
    }
}

// Anzuzeigendes Projekt neu setzen
if (newProject != null)
    setProject(newProject);
// Selektion durchführen
IStructuredSelection newSelection =
    new StructuredSelection(selectedParts.toArray());
viewer.setSelection(newSelection);
if (partToReveal != null)
    viewer.reveal(partToReveal);
// Aktionen aktualisieren
updateActions();
}
}
}

```

Es folgen die Methoden zum Management der Selektions-Listener. Hier trägt sich vor allem der globale `SelectionService` als Listener ein, wenn sich der View als `ISelectionProvider` zu erkennen gibt.

Benachrichtigt werden die eingetragenen Listener über die Methode `fireSelectionChanged()`, die wir später bei Selektionsänderung im grafischen Viewer auslösen werden. Auch hier ist es wieder erforderlich, die Aktivierung der Aktionen zu veranlassen.

```

// Registriert von Beobachter für Selektionsänderungen
public void addSelectionChangedListener(
    ISelectionChangedListener listener) {
    if (!selectionChangedListeners.contains(listener))
        selectionChangedListeners.add(listener);
}

// Entfernt den spezifizierten Listener
public void removeSelectionChangedListener(
    ISelectionChangedListener listener) {
    selectionChangedListeners.remove(listener);
}

```

```

// Selektionsereignis an Listener weitergeben
private void fireSelectionChanged() {
    // Selektionsereignis herstellen
    SelectionChangedEvent e = new SelectionChangedEvent(
        this, getSelection());
    // An Listener weitermelden
    for (ISelectionChangedListener listener :
        selectionChangedListeners)
        listener.selectionChanged(e);
    // Aktionen aktualisieren
    updateActions();
}

```

Neben einem `ISelectionProvider` implementiert der View außerdem einen `ISelectionListener`, um sich beim globalen `SelectionService` als Listener registrieren und so auf globale Selektionsereignisse reagieren zu können. Um Zyklen zu vermeiden, prüfen wir vorsorglich ab, dass die auslösende `Workbench`-Komponente nicht der `Gantt-View` selbst ist.

```

public void selectionChanged(IWorkbenchPart part,
    ISelection selection) {
    if (part != GanttView.this)
        setSelection(selection);
}

```

#### *GUI aufbauen*

Im Großen und Ganzen weicht der Aufbau dieses Views nicht von anderen Views ab. Zunächst wird mit der Methode `createGraphicalViewer()` der grafische Anzeigebereich aufgebaut, dann werden Aktionen erzeugt, installiert und schließlich noch aktualisiert. Zu beachten ist, dass GEF-Aktionen wie `DeleteAction` und `PrintAction` wieder – wie oben geschehen – mit `dispose()` freigegeben werden müssen. Im Anschluss an den Aufbau des Views erfolgt die Registrierung als `ISelectionProvider` und als `ISelectionListener` beim `SelectionService` des aktuellen `Workbench`-Fensters. In der Methode `setFocus()` wird der Fokus auf den Canvas des Viewers gesetzt, wenn der View aktiviert wird.

```

// Inhalt des Views erzeugen
@Override
public void createPartControl(Composite parent) {
    // Viewer erzeugen
    createGraphicalViewer(parent);
    // Aktionen erzeugen und installieren
    makeActions();
    registerGlobalActions();
    hookContextMenu();
    contributeToActionBars();
}

```

```

// Aktionen aktualisieren
updateActions();
// Den Viewer als SelectionProvider mit der
// View-Site registrieren
getSite().setSelectionProvider(this);
// Beim zentralen Selektionservice anmelden
ISelectionService selectionService =
    getSite().getWorkbenchWindow().getSelectionService();
selectionService.addSelectionListener(this);
}

// Setzt Fokus auf den Viewer
@Override
public void setFocus() {
    viewer.getControl().setFocus();
}

```

Der für den Anzeigebereich verantwortliche `ScrollingGraphicalViewer` wird in zwei Schritten erzeugt: zunächst die `ScrollingGraphicalViewer`-Instanz selbst, dann wird dessen GUI-Element (ein `Draw2D-FigureCanvas`) mit Hilfe der Methode `createControl()` erzeugt. Der Viewer wird dann in der `EditDomain` registriert und anschließend konfiguriert.

*Viewer*

In diesem Beispiel implementieren wir keine Palette. Stattdessen unterstützen wir die Mausklicks des Benutzers direkt. Ein Mausklick auf ein grafisches Objekt soll dieses Objekt selektieren. Wird dabei die *Shift*-Taste gedrückt, soll das Objekt der bestehenden Selektion hinzugefügt oder davon entfernt werden. Im *GANTT*-View definierte Aktionen können dann Operationen auf die selektierten Objekte veranlassen. Wir fügen deshalb dem Canvas des grafischen Viewers einen `MouseListener` hinzu. Dieser sucht im Viewer das grafische Objekt an der Mausposition und setzt dann dieses Objekt im Viewer auf selektiert. Diese Selektionsereignisse werden wiederum von einem anonymen `ISelectionChangedListener` empfangen und über `fireSelectionChanged()` weitergemeldet.

```

/**
 * Erzeugt und konfiguriert den grafischen Viewer
 * @param parent – der übergeordnete Behälter
 */
protected void createGraphicalViewer(Composite parent) {
    // Viewer erzeugen
    viewer = new ScrollingGraphicalViewer();
    viewer.createControl(parent);
    // Viewer muss für die Kommandounterstützung
    // einer EditDomain hinzugefügt werden
    editDomain.addViewer(viewer);
}

```

```

// Viewer konfigurieren
configureGraphicalViewer();
// Den Viewer mit einem MouseListener versehen
viewer.getControl().addMouseListener(new MouseAdapter() {
    @Override
    public void mouseUp(MouseEvent e) {
        // EditPart anhand der Mausposition suchen
        EditPart part = viewer.findObjectAt(
            new Point(e.x, e.y));
        // Selektion im Viewer setzen
        if ((e.stateMask & SWT.SHIFT) == SWT.SHIFT) {
            IStructuredSelection sel =
                (IStructuredSelection)
                    viewer.getSelection();
            @SuppressWarnings("unchecked")
            List<Object> list = sel.toList();
            if (list.remove(part))
                list.add(part);
            viewer.setSelection(new StructuredSelection(
                list.toArray()));
        } else
            viewer.setSelection(new StructuredSelection(
                part));
    }
});
viewer.addSelectionChangedListener(
    new ISelectionChangedListener() {
        public void selectionChanged(
            SelectionChangedEvent event) {
            fireSelectionChanged();
        }
    });
}

```

*EditParts* Bei der Konfiguration des grafischen Viewers wird die Hintergrundfarbe des Viewers gesetzt und eine EditPartFactory eingerichtet. Diese erzeugt für jede Project-Instanz eine GanttProjectEditPart-Instanz und für jede Task-Instanz eine GanttTaskEditPart-Instanz.

```

protected void configureGraphicalViewer() {
    // Hintergrundfarbe einstellen
    viewer.getControl().setBackground(
        ColorConstants.listBackground);
    // Mit der hier definierten EditPartFactory werden
    // grafische Objekte dynamisch erzeugt.
    viewer.setEditPartFactory(new EditPartFactory() {

```

```

public EditPart createEditPart(
    EditPart context, Object model) {
    // Grafische Repräsentation des Projektes
    return (model instanceof Project)
        ? new GanttProjectEditPart((Project) model)
        // Grafische Repräsentation eines Tasks
        : (model instanceof Task)
        ? new GanttTaskEditPart(
            GanttView.this, (Task) model)
        : null;
    }
});
}

```

Mit der Methode `setProject()` wird das jeweils anzuzeigende Projekt an den `GanttView` übergeben. Dieser übernimmt die `Project`-Instanz und meldet sich beim Projekt als `ModelListener` an. Natürlich müssen wir uns bei dem vorher angezeigten Projekt auch als `ModelListener` abmelden. Auch hier muss die Aktivierung der Aktionen aktualisiert werden.

*Modelländerungen*

Modelländerungen werden über das `ModelListener`-Interface an der Methode `valueChanged()` gemeldet. Hier wird immer die Grafik komplett neu aufgebaut, was durch die `setContents()`-Methode geschieht. Anschließend wird die Selektion über die Methode `setSelection()` neu auf das übergebene Objekt gesetzt. Die komplette Verarbeitung muss in einem `Display.syncExec()`-Block geschehen, denn die Modelländerung könnte in einem anderen Thread durchgeführt worden sein.

```

/**
 * Projekt setzen
 * @param newProject
 */
private void setProject(Project newProject) {
    if (newProject != project) {
        if (project != null)
            project.removeModelListener(GanttView.this);
        // Neues Projekt übernehmen
        project = newProject;
        if (project != null) {
            // und dem Viewer als Inhalt übergeben
            viewer.setContents(project);
            // Änderungen am Datenmodell beobachten
            project.addModelListener(GanttView.this);
            updateActions();
        }
    }
}
}

```

```

/**
 * Baut Grafik neu auf und setzt die Selektion
 * @param - zu selektierendes Element
 */
public void valueChanged(int operation,
                          List<PlannerItem<?>> changed,
                          final PlannerItem<?> select) {
    getSite().getShell().getDisplay().asyncExec(
        new Runnable() {
            public void run() {
                viewer.setContents(project);
                setSelection((select != null)
                    ? new StructuredSelection(sel)
                    : StructuredSelection.EMPTY);
            }
        });
}

```

#### Aktionen

Das Erzeugen und Installieren der Aktionen erfolgt auf die übliche Art wie schon bei den anderen Views dieser Beispielapplikation. Wir erzeugen hier nur eine einzige Aktion zum Löschen von selektierten Objekten. Die Klasse `DeleteAction` ist bereits im GEF definiert. Wird sie ausgeführt, fordert sie bei den selektierten `EditParts` ein Kommandoobjekt an, das anschließend ausgeführt wird. Dabei stellt die Klasse, die eine Unterklasse von `SelectionAction` ist, selbsttätig fest, welche `EditParts` selektiert sind. Normalerweise versucht sie das, indem sie sich beim globalen `SelectionService` registriert. Da wir aber auf dieser Ebene mit Domänenmodellobjekten arbeiten, würde die `Delete`-Aktion nie aktiviert werden. Wir registrieren deshalb den grafischen Viewer, der mit `EditParts` arbeitet, bei der `DeleteAction` als `SelectionProvider`.

Auch die Klasse `PrintAction` ist im GEF definiert. Bei ihrer Ausführung wird der Inhalt des Views auf Drucker ausgegeben. Die `PrintAction`-Instanz wird mit der globalen `PRINT`-Aktion registriert, da sie vom `File`-Menü der Workbench und nicht von der lokalen Werkzeugleiste des Views angesteuert wird. Das Gleiche tun wir mit den Aktionen der `UndoRedoActionGroup`.

```

// Aktionen erzeugen
protected void makeActions() {
    deleteAction = new DeleteAction(this);
    deleteAction.setSelectionProvider(viewer);
    printAction = new PrintAction(this);
}

```

```

// Globale Aktionen registrieren
private void registerGlobalActions() {
    getViewSite().getActionBars().setGlobalActionHandler(
        ActionFactory.PRINT.getId(), printAction);
    undoRedoActionGroup = new UndoRedoActionGroup(
        getSite(),
        PlannerActivator.ALWAYS_UNDO_CONTEXT, false);
    undoRedoActionGroup.
        fillActionBars(getViewSite().getActionBars());
}

// Kontextmenü registrieren
private void hookContextMenu() {
    // Menümanager anfertigen
    MenuManager menuMgr = new MenuManager("#PopupMenu");
    // Menüpunkte nicht bis zum nächsten Aufruf
    // aufbewahren
    menuMgr.setRemoveAllWhenShown(true);
    // Beim Aufruf Menü dynamisch erzeugen
    menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager manager) {
            GanttView.this.fillContextMenu(manager);
        }
    });
    // Menü erzeugen – Viewer wird dabei dem Menü bekannt gemacht
    Menu menu = menuMgr.createContextMenu(viewer.getControl());
    // Das Menü der Table oder dem Tree bekannt machen
    viewer.getControl().setMenu(menu);
    // Mit der ViewSite registrieren
    getSite().registerContextMenu(menuMgr, viewer);
}

// Menüs und Werkzeugleisten füllen
private void contributeToActionBars() {
    IActionBars bars = getViewSite().getActionBars();
    fillLocalPullDown(bars.getMenuManager());
    fillLocalToolBar(bars.getToolBarManager());
}

/**
 * Das Kontextmenü füllen
 * @param manager – der Menümanager
 */
protected void fillContextMenu(IMenuManager manager) {
    manager.add(deleteAction);
    // Andere Plugins können hier andocken
    manager.add(new Separator(
        IWorkbenchActionConstants.MB_ADDITIONS));
}

```

```

/**
 * Menü des Views füllen
 * @param manager – der Menümanager
 */
protected void fillLocalPullDown(IMenuManager manager) {
    manager.add(deleteAction);
}

```

*Adapter* Schließlich implementieren wir noch die Methode `getAdapter()`, mit welcher der `GanttView` wesentliche Funktionalität an der `IAdaptable`-Schnittstelle (siehe Abschnitt 3.3.4) bereitstellt, insbesondere den grafischen Viewer und den `CommandStack`. Ohne diese Definitionen funktioniert z.B. die Kommandoausführung nicht. Außerdem stellen wir noch die aktuelle `Shell` und den `JPA-EntityManager` für auszuführende Operationen zur Verfügung.

```

/**
 * Liefert den passenden Adapter für den angegebenen
 * Schlüssel.
 */
@SuppressWarnings("unchecked")
@Override
public Object getAdapter(Class type) {
    if (type == EntityManager.class)
        return entityManager;
    if (type == Shell.class)
        return getSite().getShell();
    // Der grafische Viewer
    if (type == GraphicalViewer.class)
        return viewer;
    // Der CommandStack für die Kommandoausführung
    if (type == CommandStack.class)
        return editDomain.getCommandStack();
    // Das grafische Wurzelobjekt
    // In einigen Fällen kann der Aufruf erfolgen, bevor ein
    // Viewer erzeugt wurde
    if (type == EditPart.class && viewer != null)
        return viewer.getRootEditPart();
    // Die Grafik des grafischen Wurzelobjekts
    if (type == IFigure.class && viewer != null)
        return ((GraphicalEditPart) viewer.getRootEditPart()).
            getFigure();
    return super.getAdapter(type);
}
}

```

## Die EditParts

Die Klasse `GanttProjectEditPart` realisiert die grafische Präsentation des gesamten Projektes in Form einer Gantt-Grafik. In einer solchen Grafik sind die einzelnen Tasks nach Startzeit untereinander angeordnet. Links befindet sich ein Bereich mit den Namen der Tasks, rechts davon wird das Task-Intervall auf einer Zeitachse je nach Startzeit und Dauer in unterschiedlicher Position und Länge angezeigt.

*Projekt*

Wird eine neue `GanttProjectEditPart`-Instanz erzeugt, so wird dem Konstruktor als Parameter das Datenmodell des Projekts mitgegeben. Dieses wird in der Instanz gespeichert, so dass jederzeit auf das Datenmodell zugegriffen werden kann. Ebenfalls implementiert wird die Methode `getModelChildren()`, welche die Kindobjekte der `Project`-Instanz, also die Tasks, abliefert. Diese Methode wird von GEF aufgerufen, um herauszufinden, welche grafischen Kindelemente noch für die aktuelle `EditPart`-Instanz zu erzeugen sind.

Übrigens erzeugt GEF nicht jedes Mal alle `EditPart`-Kindelemente neu, sondern hält verwendete `EditPart`-Elemente in einem Pool vor und verwendet sie wieder, wenn das entsprechende Datenobjekt angezeigt werden muss. Das kann in manchen Fällen (wie weiter unten in der Methode `refreshVisuals()`) zu unerwünschten Effekten führen, so dass es erforderlich werden kann, die wiederverwendeten `EditPart`-Objekte mit Hilfe von `refresh()` zu aktualisieren.

*Caching*

```
package com.bdaum.planner.taskplanner.views;
import java.util.ArrayList;
import java.util.List;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.FrameBorder;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.XYLayout;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;
import com.bdaum.planner.core.model.Project;
import com.bdaum.planner.core.model.Task;

/**
 * Diese Klasse implementiert die grafische Repräsentation eines
 * Projekts im Gantt-View
 */
public class GanttProjectEditPart extends
    AbstractGraphicalEditPart {
```

```

// Konstruktor
public GanttProjectEditPart(Project project) {
    super();
    setModel(project);
}

// Liefert die Tasks des Projektes in Form einer Liste
@Override
protected List<Task> getModelChildren() {
    return new ArrayList<Task>(
        ((Project) getModel()).getTasks());
}

```

*Draw2D-Repräsentation*

Mit der Methode `createFigure()` muss die grafische Repräsentation des Objektes konstruiert werden. Das geschieht mit Draw2D-Hilfsmitteln. Hier wird nur ein generisches Figure-Objekt als eine Art Canvas erzeugt, auf undurchsichtig gesetzt und mit einem XY-Layout (d.h. einem Layout, in dem absolut positioniert werden kann) versehen. Außerdem wird mit Hilfe einer `FrameBorder` eine Titelzeile erzeugt.

```

/**
 * Erzeugt die Grafik dieses Projekts. Dies ist anfänglich
 * ein leerer Bereich, der mit einem XY-Layout formatiert wird.
 */
@Override
protected IFigure createFigure() {
    Figure f = new Figure();
    f.setOpaque(true);
    f.setLayoutManager(new XYLayout());
    // Titelzeile setzen
    f.setBorder(
        new FrameBorder(((Project) getModel()).getName()));
    return f;
}

```

*Komponente aktivieren*

Die Methode `activate()` wird aufgerufen, wenn ein `EditPart`-Objekt sichtbar gemacht wird. In diesem Fall muss die Methode `refreshVisuals()` aufgerufen werden, um die Darstellung zu aktualisieren. Wird das Projekt nicht mehr im Viewer angezeigt, wird die Methode `deactivate()` aufgerufen.

```

/**
 * Aktiviert das Objekt.
 * Die grafische Darstellung wird aktualisiert.
 */
@Override
public void activate() {

```

```

        if (!isActive()) {
            super.activate();
            // Grafik aktualisieren
            refreshVisuals();
        }
    }

    // Deaktiviert das Objekt.
    @Override
    public void deactivate() {
        if (isActive())
            super.deactivate();
    }
}

```

Normalerweise wäre es ausreichend, in der Methode `refreshVisuals()` die Super-Methode und die Methode `refreshChildren()` aufzurufen und es den einzelnen Kindobjekten zu überlassen, ihrerseits die ihnen zugeordneten Modellelemente zu beobachten und bei Änderungen dann `refreshVisuals()` aufzurufen. Hier, bei der Gantt-Darstellung, liegen die Dinge jedoch etwas anders. Änderungen an einem einzigen Task können die Aktualisierung aller Task-Darstellungen nötig machen, nämlich dann, wenn sich die Reihenfolge der Tasks ändert. In diesem Fall müssen alle Tasks (wenigstens die, deren Position sich geändert hat) aktualisiert werden. Das geschieht hier in der `while`-Schleife.

*Kindelemente  
aktualisieren*

```

    // Aktualisiert die grafische Darstellung
    @Override
    protected void refreshVisuals() {
        super.refreshVisuals();
        refreshChildren();
        for (Object child : getChildren())
            ((GraphicalEditPart) child).refresh();
    }

    // Für Projekte gibt es keine EditPolicies
    @Override
    protected void createEditPolicies() {}
}

```

Die Abbildung von Tasks erfolgt mit der Klasse `GanttTaskEditPart`. Auch hier wird im Konstruktor wieder das zugehörige Modell-Objekt (Task) übergeben, zusätzlich aber noch das `GanttView-IAdaptable`, über das sowohl Zugriff auf die aktuelle Shell als auch auf den JPA-Entity-Manager besteht, denn an Tasks sollen Änderungen möglich sein (wir implementieren eine Löschkaktion). Eine Methode `getModelChildren()`

*Tasks*

muss hier nicht implementiert werden, denn unterhalb von Tasks wollen wir keine Details abbilden.

```

package com.bdaum.planner.taskplanner.views;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import org.eclipse.core.runtime.IAdaptable;
import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.Panel;
import org.eclipse.draw2d.PositionConstants;
import org.eclipse.draw2d.RectangleFigure;
import org.eclipse.draw2d.XYLayout;
import org.eclipse.draw2d.geometry.Point;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.EditPolicy;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;
import com.bdaum.planner.core.model.Project;
import com.bdaum.planner.core.model.Task;

public class GanttTaskEditPart extends AbstractGraphicalEditPart {
    // Höhe der Task-Darstellung
    private final static int TASKHEIGHT = 25;
    // Maximale Breite für Namen
    private final static int NAMEWIDTH = 80;
    // Konstante zur Umrechnung von Zeitangaben
    private static final long MILLISEC_PER_MINUTE = 60000L;
    // Shell und EntityManager-Lieferant
    private IAdaptable info;
    // Konstruktor
    public GanttTaskEditPart(IAdaptable info, Task task) {
        super();
        setModel(task);
        this.info = info;
    }
}

```

#### *Draw2D-Repräsentation*

In der Methode `createFigure()` wird dann wieder die grafische Repräsentation mit Draw2D-Mitteln konstruiert. In diesem Falle ist sie etwas komplexer. Zunächst wird als Behälter ein `Panel` erzeugt, das auch mit einem `XY-Layoutmanager` versehen wird. Auf diesem `Panel` wird links ein `Label` mit dem Task-Namen angebracht. Mit der Methode `setConstraint()` wird die Position und die Größe des Labels relativ zum `Panel` spezifiziert.

```

@Override
protected IFigure createFigure() {
    // Panel erzeugen, auf undurchsichtig setzen und mit
    // einem XYLayout formatieren
    final Panel panel = new Panel();
    panel.setOpaque(true);
    panel.setLayoutManager(new XYLayout());
    // Der Task-Name wird als Label angezeigt
    Label label = new Label();
    Task task = ((Task) getModel());
    label.setText(" " + task.getName());
    // Text links im Label anzeigen
    label.setLabelAlignment(PositionConstants.LEFT);
    // Label hellgrau und undurchsichtig
    label.setOpaque(true);
    label.setBackgroundColor(ColorConstants.lightGray);
    panel.add(label);
    // Label in Position (0,0) und Größe konfigurieren
    panel.setConstraint(label, new Rectangle(0, 0,
        NAMEWIDTH, TASKHEIGHT));
}

```

Als Nächstes erfolgt die Abbildung des Task-Intervalls in Form eines farbigen Rechtecks, das ebenfalls auf dem Panel angeordnet wird. Die Position entspricht dem Startzeitpunkt des Tasks relativ zum Projektstart. Die Breite entspricht der Task-Dauer. Bei unkritischen Tasks wird mit einem zweiten grauen Rechteck noch das Ausmaß des Spiels angezeigt.

*Darstellung*

Vom Label bis zu diesem Rechteck ziehen wir noch eine Trennlinie. Der Einfachheit halber verwenden wir dazu ein Rechteck der Höhe 1. Schließlich geben wir mit `setSize()` und `setLocation()` die Größe und Position des gesamten Panels an. Dabei geben wir aber die Größe und Position des farbigen Rechtecks relativ zum Panel an. Diese Angaben haben keinen Einfluss auf das Layout, sondern definieren einen Hot-spot dieses `Edi tParts`, der angezeigt werden soll, wenn beim Viewer die Methode `reveal()` aufgerufen wird.

```

// Nun das Intervall des Tasks als Rechteck anzeigen
RectangleFigure rect = new RectangleFigure();
// Dunkelgrün mit dunkelgrauem Rand
rect.setOpaque(true);
rect.setBackgroundColor(ColorConstants.darkGreen);
rect.setForegroundColor(ColorConstants.darkGray);
panel.add(rect);

```

```

// Wir positionieren relativ zum Startdatum des Projektes
Project project = task.getParent();
int x = (int) ((task.getEarlyStart().getTime() -
    project.getStartDate().getTime()) / MILLISEC_PER_MINUTE);
// Die Breite entspricht der Task-Dauer
int w = task.getDuration();
panel.setConstraint(rect, new Rectangle(NAMEWIDTH + x,
    0, w, TASKHEIGHT - 1));
// Ein zweites Rechteck für den Slack bei nicht
// kritischen Tasks
int slack = (int) ((task.getLateStart().getTime()
    - task.getEarlyStart().getTime())
    / MILLISEC_PER_MINUTE);
if (slack > 0) {
    RectangleFigure rect2 = new RectangleFigure();
    rect2.setOpaque(true);
    rect2.setBackgroundColor(ColorConstants.lightGray);
    rect2.setForegroundColor(ColorConstants.lightGray);
    panel.add(rect2);
    panel.setConstraint(rect2, new Rectangle(NAMEWIDTH
        + x + w, 0, slack, TASKHEIGHT - 1));
}
// Nun die Trennlinie erzeugen
final RectangleFigure line = new RectangleFigure();
line.setForegroundColor(ColorConstants.gray);
line.setOpaque(true);
panel.add(line);
// Als Länge wird die Position des farbigen Blocks verwendet
panel.setConstraint(line, new Rectangle(0,
    TASKHEIGHT - 1, NAMEWIDTH + x, 1));
/*
 * Mit setSize() und setLocation() täuschen wir für das
 * Panel die Größe des farbigen Rechtecks vor. Diese
 * Angaben spielen nur beim Sichtbarmachen (reveal) eine
 * Rolle, nicht jedoch beim Layout.
 */
panel.setSize(w, TASKHEIGHT);
panel.setLocation(new Point(NAMEWIDTH + x, 0));
return panel;
}

```

*Aktualisieren*

In der Methode `refreshVisuals()` wird dann die endgültige Position des gesamten Task-Panels festgelegt. Vorher werden aber noch die Farben des Panels und des Task-Intervalls abhängig vom Selektionszustand gesetzt. Die Tasks werden zunächst entsprechend ihrer Anfangszeit sortiert.

Anschließend bestimmen wir dann die Position des Tasks auf der Y-Achse und positionieren den Task entsprechend mit Hilfe der Methode `setLayoutConstraint()`. Auch die Position des Hotspots wird entsprechend angepasst.

```
// Grafik aktualisieren
@Override
protected void refreshVisuals() {
    Task task = ((Task) getModel());
    // Rechteck, welches das Intervall repräsentiert, holen
    Panel panel = (Panel) getFigure();
    RectangleFigure rect =
        (RectangleFigure) panel.getChildren().get(
            1);
    // Farben abhängig von der Selektion setzen
    boolean isSelected = getSelected() != SELECTED_NONE;
    panel.setBackgroundColor((isSelected)
        ? ColorConstants.tooltipBackground
        : panel.getParent().getBackgroundColor());
    rect.setBackgroundColor((isSelected)
        ? ColorConstants.darkBlue
        : (task.getEarlyStart().equals(task.getLateStart()))
        ? ColorConstants.red
        : ColorConstants.darkGreen);
    rect.setForegroundColor((isSelected)
        ? ((task.getEarlyStart().equals(task.getLateStart()))
        ? ColorConstants.red
        : ColorConstants.green)
        : ColorConstants.darkGray);
    // Alle Tasks im Projekt holen, um die Reihenposition
    // des aktuellen Tasks bestimmen zu können
    List<Task> tasks = task.getParent().getTasks();
    Collections.sort(tasks, new Comparator<Task>() {
        public int compare(Task o1, Task o2) {
            return o1.getEarlyStart().compareTo(
                o2.getEarlyStart());
        }
    });
    @Override
    public boolean equals(Object obj) {
        return false;
    }
});
// Die Position des Tasks bestimmen
int y = tasks.indexOf(task) * TASKHEIGHT;
Rectangle r = new Rectangle(0, y, -1, TASKHEIGHT);
y += TASKHEIGHT;
```

```

        // Position im Parent-EditPart setzen
        ((GraphicalEditPart) getParent()).setLayoutConstraint(
            this, panel, r);
        // Position des Hotspots setzen
        panel.setLocation(new Point(panel.getLocation().x, y));
    }

```

Wird ein Objekt selektiert, so muss es neu gezeichnet werden, denn wir wollen das Objekt in anderen Farben darstellen.

*EditPolicy*

Schließlich wird für diesen EditPart-Typ noch eine EditPolicy installiert. Wird eine DeleteAction (siehe oben) aufgerufen, so erfolgt mit Hilfe der EditPolicy die Erzeugung eines Kommandoobjektes. Hier erzeugen wir zunächst eine Instanz der Klasse und installieren sie unter dem Schlüssel COMPONENT\_ROLE mit Hilfe von installEditPolicy().

```

        // Markiert das Objekt als selektiert
        @Override
        public void setSelected(int value) {
            super.setSelected(value);
            // Neu zeichnen, da sich Farbe ändert
            refreshVisuals();
        }

        // EditPolicy für das Löschen von Tasks
        @Override
        protected void createEditPolicies() {
            installEditPolicy(EditPolicy.COMPONENT_ROLE,
                new TaskComponentEditPolicy(info));
        }
    }

```

### Die Klasse TaskComponentEditPolicy

*Rollen*

Da es sich bei der TaskComponentEditPolicy um eine EditPolicy in einer Komponentenrolle handelt, wird diese Klasse als Unterklasse von ComponentEditPolicy erzeugt. Deren Methode createDeleteCommand() wird überschrieben, um im Falle eines Task-EditParts ein Löschkommando zu erzeugen. Dabei wird dem Kommando das schon oben erwähnte IAdaptable mitgegeben, welches der TaskComponentEditPolicy-Instanz im Konstruktor übermittelt wurde.

```

package com.bdaum.planner.taskplanner.views;
import java.util.List;
import org.eclipse.core.runtime.IAdaptable;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.ComponentEditPolicy;
import org.eclipse.gef.requests.GroupRequest;

import com.bdaum.planner.core.model.Task;
import com.bdaum.planner.taskplanner.operations.TaskDeleteCommand;

public class TaskComponentEditPolicy extends ComponentEditPolicy {
    // Das IAdaptable des GANTT-View
    private IAdaptable info;

    /**
     * Konstruktor
     * @param info – IAdaptable, das mindestens eine Shell und
     *             den JPA-EntityManager liefert
     */
    public TaskComponentEditPolicy(IAdaptable info) {
        super();
        this.info = info;
    }

    // Erzeugt ein Löschkommando
    @Override
    protected Command createDeleteCommand(
        GroupRequest deleteRequest) {
        // Beteiligte Komponenten holen
        @SuppressWarnings("unchecked")
        List<EditPart> editParts = deleteRequest.getEditParts();
        Object child = getHost().getModel();
        // Prüfen, ob Typ stimmt
        return (child instanceof Task) ? new TaskDeleteCommand(
            editParts, info) : null;
    }
}

```

### Die Klasse TaskDeleteCommand

Die Klasse TaskDeleteCommand baut auf der Klasse Command auf, die zur GEF-Kommandoinfrastruktur gehört. Diese Infrastruktur verfügt über einen CommandStack in der EditDomain, der auch das *Undo* von Kommandos ermöglicht.

*GEF-Kommando*

Wir möchten allerdings die *Undo*-Logik lieber in die globale Undo-Infrastruktur der Plattform integrieren, da wir auch von anderen Views

aus Operationen durchführen. Wir nutzen deshalb hier die *Undo*-Fähigkeiten des GEF nicht aus, sondern delegieren die auszuführende Operation an `RemoveTaskOperation`-Instanzen (siehe Abschnitt 2.5.2), die wieder in der bewährten Weise in eine `JpaTransaction` (siehe Abschnitt 14.1.6) gekapselt und asynchron über den `OperationJob` (siehe Abschnitt 3.3.3) ausgeführt werden. Der *Undo* wird dann nur noch über die `JpaTransaction` abgewickelt, die Klasse `TaskDeleteCommand` bleibt davon unbeeinflusst. Die Funktionalität dafür ist in der Superklasse `Command` vorhanden, wir nutzen sie jedoch nicht.<sup>1</sup>

```
package com.bdaum.planner.taskplanner.operations;
import java.util.List;
import javax.persistence.EntityManager;
import org.eclipse.core.runtime.IAdaptable;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.commands.Command;
import com.bdaum.planner.OperationJob;
import com.bdaum.planner.core.model.Task;
import com.bdaum.planner.core.operations.JpaTransaction;
import com.bdaum.planner.core.operations.RemoveTaskOperation;

public class TaskDeleteCommand extends Command {
    // Die zu löschenden Editparts
    List<EditPart> editParts;
    private IAdaptable info;

    /**
     * Konstruktor
     * @param editParts – die Liste der zu löschenden EditParts
     * @param info – IAdaptable, das mindestens eine Shell und
     *             einen EntityManager liefert
     */
    public TaskDeleteCommand(List<EditPart> editParts,
                             IAdaptable info) {
        super();
        this.editParts = editParts;
        this.info = info;
    }
}
```

---

1. Der globale Undo-Stack wurde in der Eclipse-Plattform erst relativ spät eingeführt, als GEF seinen eigenen Undo-Stack schon besaß.

```

/**
 * Kommando ausführen
 */
@Override
public void execute() {
    EntityManager entityManager =
        (EntityManager) info.getAdapter(EntityManager.class);
    JpaTransaction transaction =
        new JpaTransaction("Remove Tasks", entityManager);
    for (EditPart part : editParts) {
        Task task = (Task) part.getModel();
        RemoveTaskOperation operation =
            new RemoveTaskOperation(task);
        transaction.add(operation);
    }
    // Transaktion starten
    OperationJob.executeOperation(transaction, info, false);
}
}

```

## Lineale

Damit ist eigentlich die Definition des Gantt-Views abgeschlossen. Wir haben einen einfachen grafischen Viewer für Projektpläne implementiert, der sogar rudimentäre Bearbeitungsmöglichkeiten (Löschen von Tasks) anbietet. Was wir allerdings vermissen, ist die Anzeige einer Zeitachse mit Zeiteinteilung. Unschön ist außerdem, dass die Task-Namen auf der linken Seite verschwinden, wenn das Projekt im Viewport verschoben wird. Ideal wäre es, wenn wir ein horizontales Lineal (Ruler) einblenden könnten, auf dem Datum und Uhrzeit angegeben sind. Ein vertikales Lineal könnte die Task-Namen aufnehmen, und zwar vertikal mit Scrollen, bei horizontalen Scrolls jedoch an Ort und Stelle bleiben. Soweit unsere Wunschvorstellung.

Schauen wir uns an, was GEF in dieser Hinsicht zu bieten hat. In der Tat gibt es Unterstützung für Lineale. Ruler werden intern in einem eigenen Viewer implementiert. Sie sind aktive Grafikobjekte, welche ein eigenes Datenmodell besitzen und auch Kommandos auslösen können. Auch passen sich die Ruler beim Scrollen und Zoomen der Grafik an und unterstützen die Arbeit mit Hilfslinien (Guides).

*Ruler*

Allerdings lässt die Enttäuschung nicht lange auf sich warten. Zwar kann man bei Maßeinheiten zwischen Zentimetern, Zoll und Pixeln wählen und kann auch das Layout der Ruler in Grenzen konfigurieren, das war's aber dann schon. Weder ist es möglich, für ein Lineal einen Ursprungspunkt zu definieren, noch können selbstge-

*Probleme*

schriebene Beschriftungsgeneratoren für Skalen eingesetzt werden. So lässt sich denn auch unser Wunsch nach einem mit Datums- und Zeitangaben beschrifteten Lineal nicht mit Hilfe von GEF-Rulern realisieren. Das Gleiche gilt natürlich auch für ein mit Task-Namen beschriftetes vertikales Lineal.

Trotzdem wollen wir zur Demonstration einen horizontalen Ruler in unseren Gantt-View einbauen. Als Maßeinheit wählen wir Pixel, wobei ein Pixel einer Minute entspricht.

Dazu ändern wir die Methode `createGraphicalViewer()` in der Klasse `GanttView` ab (Änderungen sind fett gedruckt). Wie schon oben erwähnt, werden Ruler innerhalb eines eigenen Viewers implementiert. Dieser muss dann zusammen mit dem Viewer für den Grafikbereich in ein `Composite` gruppiert werden. Deshalb wird zunächst ein `RulerComposite` erzeugt und das GUI-Element des grafischen Viewers innerhalb dieses `Composites` angelegt. Außerdem muss der Viewer mit dem `RulerComposite` mit Hilfe der Methode `setGraphicalViewer()` registriert werden.

Anschließend wird dem Viewer mit Hilfe der Methode `setProperty()` ein `RulerProvider` zugeordnet. Der `RulerProvider` arbeitet als Fabrik für Ruler und liefert Information wie das Ruler-Datenmodell oder die Maßeinheit. Wie schon erwähnt, besitzt ja jeder Ruler sein eigenes Datenmodell. Im einfachsten Fall handelt es sich dabei um ein leeres Objekt. Hier verzichten wir sogar darauf, extra mit `new Object()` ein solches Objekt anzufertigen, sondern geben in der Methode `getRuler()` die aktuelle `RulerProvider`-Instanz selbst als Datenmodell des Rulers zurück. Schließlich wird mit einem zweiten `setProperty()`-Aufruf der Ruler sichtbar geschaltet. Mit einem solchen Aufruf kann ein Ruler dynamisch ein- und ausgeblendet werden.

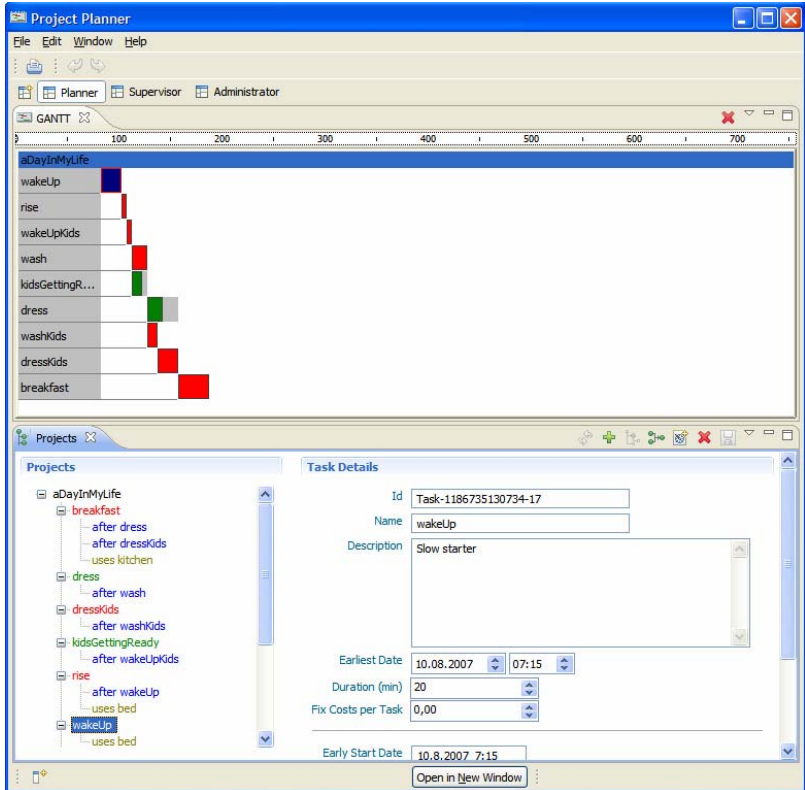
```
protected void createGraphicalViewer(Composite parent) {
    // Wenn Ruler verwendet werden, muss der Viewer in einem
    // RulerComposite angelegt werden
    final RulerComposite rc = new RulerComposite(parent, SWT.NONE);
    // Viewer erzeugen
    viewer = new ScrollingGraphicalViewer();
    // Viewer muss nun im RulerComposite erzeugt werden
    viewer.createControl(rc);
    // Viewer muss für die Kommando- und Ruler-Unterstützung
    // einer EditDomain hinzugefügt werden
    editDomain.addViewer(viewer);
    // Der Viewer muss dem RulerComposite bekannt gemacht werden
    rc.setGraphicalViewer(viewer);
    // Der RulerProvider versorgt den Viewer mit Rulern und Guides
    RulerProvider rp = new RulerProvider() {
```

```
@Override
public Object getRuler() {
    // Minimalimplementierung. Gibt sich selbst als
    // Ruler-Modell zurück
    return this;
}

@Override
public int getUnit() {
    // Wir verwenden Pixel als Maßeinheit
    return RulerProvider.UNIT_PIXELS;
}
};
// RulerProvider wird mit dem Ruler registriert
viewer.setProperty(
    RulerProvider.PROPERTY_HORIZONTAL_RULER, rp);
// Ruler werden sichtbar geschaltet
viewer.setProperty(
    RulerProvider.PROPERTY_RULER_VISIBILITY,
    Boolean.TRUE);
// Viewer konfigurieren
configureGraphicalViewer();
// Den Viewer als SelectionProvider mit der View-Site
// registrieren
getSite().setSelectionProvider(viewer);
// Den Viewer mit einem MouseListener versehen
...
...
}
```

### Test

Nun kann die Applikation erneut getestet werden. In Abschnitt 16.4.2 hatten wir bereits Draw2D und GEF dem Infrastruktur-Feature hinzugefügt. Wir können deshalb die Anwendung aus dem Produkteditor heraus sofort mit einem Klick auf *Synchronize* und *Launch the product* starten.



**Abb. 16–6** Projekt- und Gantt-View in der Planer-Perspektive. Schön wäre es, wenn der Ursprung der Skala auf den rechten Rand des grauen Tasknamen-Bereichs gelegt werden könnte, aber das scheint in GEF noch nicht möglich zu sein.

### 16.4.3 Aufgabe

Versehen Sie die Tasks im *Gantt*-View mit einer stärkeren Umrahmung.

## 16.5 Andere grafische Möglichkeiten

Welche Alternativen für GEF in Betracht kommen, hängt natürlich vom Anwendungsfall ab. Eine direkte Alternative, die sich ähnlich gut in die Eclipse-Plattform integriert wie das GEF, gibt es eigentlich nicht.

Für allgemeine grafische zweidimensionale Zwecke kann man auf die Grafik des SWT zurückgreifen, insbesondere wenn die Plattform die mit Eclipse 3.1 fortgeschrittenen Grafikfunktionen (siehe Abschnitt 9.1.6) unterstützt. Eine andere Möglichkeit ist die Verwendung von Java2D. Dazu muss natürlich Swing in SWT eingebettet werden; wie das geht und welche Voraussetzungen gegeben sein müssen, haben wir bereits in Abschnitt 9.1.9 diskutiert.

Eine andere Möglichkeit zur Realisierung hochwertiger interaktiver Grafiken ist die Verwendung von SVG (*Scalable Vector Graphics*). SVG ist ein XML-Dialekt für die Definition von Vektorgrafiken. SVG ist sehr leistungsfähig, und SVG-Grafiken können recht einfach erstellt werden. Für das Ansehen und Einbetten der Grafiken in eine Eclipse-Plattform gibt es zwei Möglichkeiten:

SVG

- Die Verwendung von Apache Batik. Dessen JSVGCanvas kann in beliebige Swing-Umgebungen eingebettet werden. Für Eclipse muss also wieder mit der Swing-Integration (Abschnitt 9.1.9) gearbeitet werden. Es gibt bereits ein fertiges Eclipse-Plugin für Batik, welches auch einen View für die SVG-Vorschau und einen Editor für SVG-Dateien bereitstellt. Das Plugin ist auf <http://sourceforge.net/projects/svgplugin> erhältlich. Die reine Batik-Laufzeitumgebung ist allerdings auch in der Europa-Distribution enthalten.
- Eine weitere Möglichkeit ist die Verwendung des SWT-Browser-Widgets (siehe Abschnitt 9.1.3). Da dieses Widget den nativen Browser der Ablaufplattform benutzt, kann auf dessen SVG-Funktionalität zurückgegriffen werden. Diese wird in der Regel durch ein Browser-Plugin von Adobe oder Corel hergestellt. Auf diese Weise können SVG-Zeichnungen mit guter Performanz im Browser-Widget dargestellt werden. Auch Interaktion ist möglich, da ein `LocationListener` am Browser-Widget über aufgerufene URLs informiert.