

一. RCP 简介

1. RCP 简要介绍

富客户端（RCP）这个词早在上个世纪 90 年代初，人们还在利用 VB，delphi 开发桌面应用程序的时候就已经出现了。随着桌面应用程序数量不断增长，各种各样的应用程序孕育而生，小到 Windows 自带的扫雷游戏，大到企业级桌面 ERP 系统，桌面应用充斥着我们的生活，也就是这样，我们更加渴望能得到更加丰富（Rich）的用户体验。

富客户端提供给用户高质量的用户体验，能让界面元素更加丰富，用户更容易操作，使应用程序的设计贴近用户，并且，由于富客户端采用了本地接口的调用，而不同于基于 web 浏览器的网上作业，也让富客户端的处理速度比起 B/S 结构的应用速度快。好的富客户端，提供了例如拖拽操作、系统剪切板、导航、用户自定义等 UI 元素，让程序的用户界面（UIs）更为人性化。

随着桌面应用的增加，RCP 技术也随之完善。UI 设计者可以利用各种界面设计工具，轻松地设计出完美的用户界面，并且，这些成熟的设计开发工具，也让开发者提高模块的复用性，缩短了开发时间。

早期的富客户端，开发者把复杂的业务逻辑和操作，紧紧地和系统以及用户界面本身粘在了一起，这使得他们的应用维护、扩展成本大大增加，也使得这些开发者不得不减少应用的许多功能，而把更多的精力放在了创建用户界面和连接数据库等工作上。当中间件的出现，提供给了开发者许多框架以及基础设施，这才使得开发者从那些繁琐的事务中解放出来，专心研究业务逻辑。

用户对于那些又好看又好用的桌面应用程序可谓是情有独钟，而 IT 管理者却发现，想要为用户打造这么一个即好看又实用的桌面应用，虽然不是什么难事，但是却存在着许多隐形的消耗支出和潜在的危险，就拿自动部署和升级来举例，往往开发人员会让用户变相地自动去安装某些应用，或者删除移动一些文件，更可怕的是在安装了某些新的应用后，那些本来共享的数据源却被覆盖了。

随之而来的 Web 应用（或者称为瘦客户端）称可以解决早期 RCP 带来的许多问题，就拿上面安装部署升级更新为例，Web 应用由于是集中在服务器进行了更新，而只要客户端拥有了浏览器就可以去访问服务器、执行也取流程，这就免去了安装部署的麻烦，并且，若我们需要更新我们的业务逻辑，我们只要去更新服务器就可以了。虽然这减少了维护成本，但是用户却无法得到良好高速的交互以及人性化的 UI。

虽然减少成本是好事，但是由于 Web 应用本身的特点是基于网络的，那这种请求/响应的交互方式需要基于良好的网络环境才能保证和用户之间的交互达到最好，并且由于用户的需求日趋复杂，用户对交互的要求也越来越高，Web 应用慢慢呈现出捉襟见肘的态势——很多高效率的用户交互性无法实现，用户新的要求也开始无法满足了。

如今，用户的需求以及以上的这些问题，让我们的应用开发又返回到了桌面应用上来。各种行业领域的业务越来越复杂，数据量也越来越大，这也开始需要在各个应用系统之间进行整合，Web 应用开始力不从心了。富客户端的良好特性恰恰能很好地解决这些问题。

但那些导致早期采用瘦客户端的部署升级问题呢？是的，但是如今提出的分布组件式的应用系统很好地解决了部署和升级这一问题，正如 Eclipse 那样，是这种分布组件应用的杰出代表。我们接下来将会讲述什么是 Eclipse。

2. Eclipse 和 RCP

2.1. Eclipse 介绍

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建开发环境。更多信息可以去 Eclipse 的主站点获得：www.eclipse.org。

当我们提到 Eclipse 的时候，很多 Java 开发者的第一个反映就是那个性能良好，界面漂亮，Debug 功能奇佳的 Java IDE 工具。

我们不可否认 Eclipse 常常是作为一款优秀的 IDE 出现在开发者面前的，不仅仅是 Java 的 IDE，Eclipse 还可以是 C 语言的 IDE、Python 的 IDE，或许以后还可以是其他什么

语言的 IDE。

但这些 IDE 严格来说，都是 Eclipse RCP 应用。真正的 Eclipse，是一个提供了完善插件机制的 RCP 平台，它以 SWT/JFace 座位界面元素组件，提供给用户一个名为 Workbench 的 UI 平台，加上它本身优秀的插件机制，能够构造出扩展能力强、性能优秀、并能提供给用户良好 UI 体验的 Rich Client Platform。

Eclipse 分为几大部分：

➤ **Workbench 工作台**

工作台为 Eclipse 提供用户界面。它是使用标准窗口工具包 (SWT) 和一个更高级的 API (JFace) 来构建的；SWT 是 Java 的 Swing/AWT GUI API 的非标准替代者，JFace 则建立在 SWT 基础上，提供用户界面组件。

SWT 已被证明是 Eclipse 最具争议的部分。SWT 比 Swing 或 SWT 更紧密地映射到底层操作系统的本机图形功能，这不仅使得 SWT 更快速，而且使得 Java 程序具有更像本机应用程序的外观和感觉。使用这个新的 GUI API 可能会限制 Eclipse 工作台的可移植性，不过针对大多数流行操作系统的 SWT 移植版本已经可用。

Eclipse 对 SWT 的使用只会影响 Eclipse 自身的可移植性——使用 Eclipse 构建的任何 Java 应用程序都不会受到影响，除非它们使用 SWT 而不是使用 Swing/AWT。

如果想要获得更多 SWT 的信息，请登陆 www.eclipse.org/swt 获得更多信息。

➤ **Workspace 工作区**

工作区是负责管理用户资源的插件。这包括用户创建的项目、那些项目中的文件，以及文件变更和其他资源。工作区还负责通知其他插件关于资源变更的信息，比如文件创建、删除或更改。

➤ **Help 帮助系统**

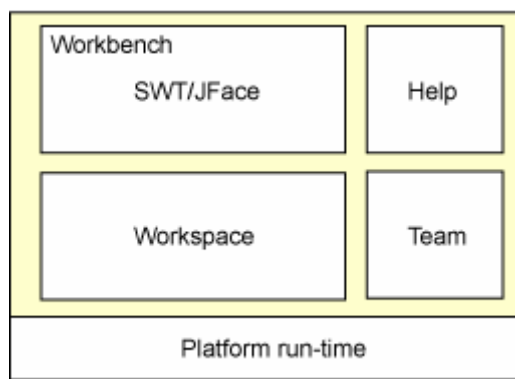
帮助组件具有与 Eclipse 平台本身相当的可扩展能力。与插件向 Eclipse 添加功能相同，帮助提供一个附加的导航结构，允许工具以 HTML 文件的形式添加文档。

➤ Team 团队支持系统

团队支持组件负责提供版本控制和配置管理支持。它根据需要添加视图，以允许用户与所使用的任何版本控制系统（如果有的话）交互。大多数插件都不需要与团队支持组件交互，除非它们提供版本控制服务。

➤ Platform run-time 运行平台

平台运行库是内核，它在启动时检查已安装了哪些插件，并创建关于它们的注册表信息。为降低启动时间和资源使用，它在实际需要任何插件时才加载该插件。除了内核外，其他每样东西都是作为插件来实现的。



在上面的介绍中，我们频频用到了“插件”（Plugins）这一词。我们会在下一节讲解什么是 Eclipse 插件。

2.2. Eclipse 插件和 RCP

Eclipse 的价值是它为创建可扩展的集成开发环境提供了一个开放源码平台。这个平台允许任何人构建与环境和其它工具无缝集成的工具。

工具与 Eclipse 无缝集成的关键是**插件**。除了小型的运行时内核之外，Eclipse 中的所有东西都是插件。从这个角度来讲，所有功能部件都是以同等的方式创建的。从这个角度来讲，所有功能部件都是以同等的方式创建的。

但是，某些插件比其它插件更重要些。正如上节图中所示，Workbench 和 Workspace 是 Eclipse 平台的两个必备的插件——它们提供了大多数插件使用的扩展点，如图 1 所示。插件需要扩展点才可以插入，这样它才能运行。

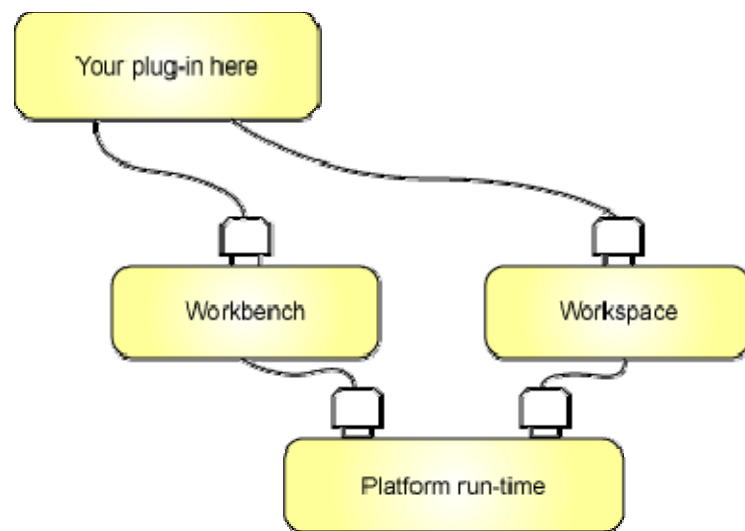


图 2.2.1 插件结构图

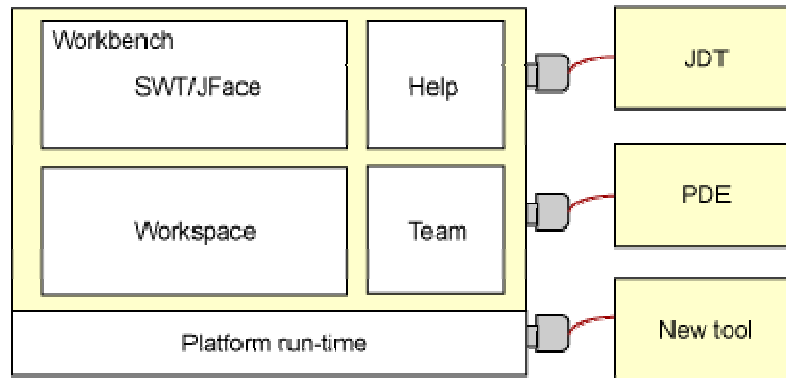
每一个 plug-in 都用一个目录包起来，而且起目录名也是有讲究的，比如 plug-in 的名字叫

`org.eclipse.swt.example`，版本是 `1.0.0`，那这个目录名就是 `org.eclipse.swt.example_1.0.0`。

而一个 plugin 工程目录下，可以发现总有一个文件叫一个 `plugin.xml`，`plugin.xml` 文件是一个描述插件的配置文件，它将插件的许多扩展信息、版本、classpath 等插件运行必须的东西都包括进来了，而在新版本的 Eclipse 中，`plugin.xml` 更多地是描述插件的扩展信息，而其他比较重要的信息都收录到了 `MF` 文件当中。

这里我们就不详细讲述如何去构建一个插件工程并生成插件了，我们会在后续的章节里介绍。

我们从上面的图了解到了 Eclipse 插件的简单机制，现在我们可以看看下图：



这个图是第一节图的一个扩充。我们可以看得出来，Eclipse 所带得 JDT（Java 开发工具）以及 PDE（插件开发工具），实际上都是以插件形式存在的，也就是说，如果我们把整个 Eclipse 的 JDT 和 PDE 等和核心组件无关的插件清楚后，那这时候的 Eclipse 就是一个完整意义上的 RCP 平台了，但是没有任何应用而已。

正如上面所说“Eclipse 的价值是它为创建可扩展的集成开发环境提供了一个开放源码平台。这个平台允许任何人构建与环境和其它工具无缝集成的工具。”，Eclipse 其实本身并不能做什么，它只是提供了这么一个平台给开发者，但是它所提供的“无缝集成”也就是它具有的插件机制，正是我们开发 RCP 应用时候最需要的东西，我们可以利用它的插件机制，对我们的 RCP 应用进行扩充以及升级，而这些操作对原有的系统是不会有有什么损害的。

正因为 Eclipse 的插件机制有如此强大的功能，所以 Eclipse 才逐渐成为 Java 开发者开发 RCP 应用的首选平台。

2.3. Eclipse 的 RCP 历程

Eclipse 并不打算先从一开始就建立一个 RCP 平台的，相反，它的目标是建立一个整个开发工具的平台。

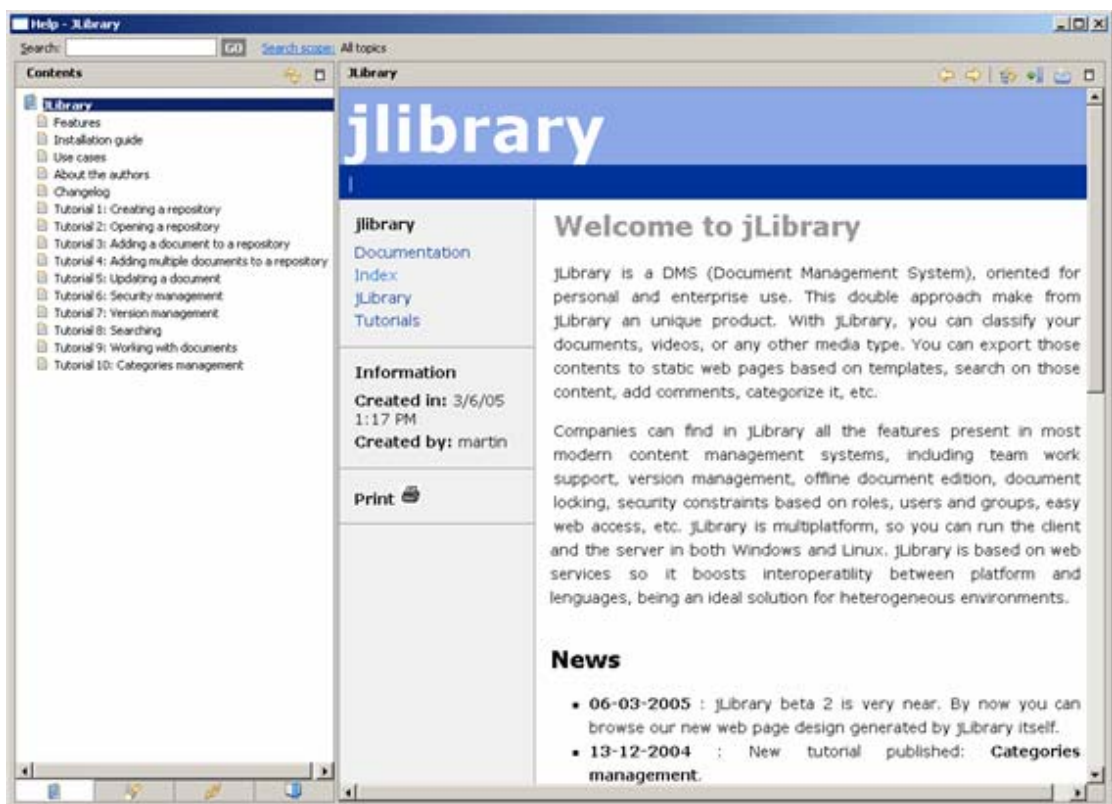
Eclipse 做为 RCP 出现是在 Eclipse 2.1 版本发布的时候而提出的，这一举动说明了 Eclipse IDE 本身是专业、漂亮、性能优良的，一些前卫的开发者指出，这一架构容易扩展，能够吸引更多的人在上面为其开发。事实上他们所说的话都灵验了。

Eclipse3.0 是使得 Eclipse 成为 RCP 一个重要的里程碑。这一版本中, Eclipse 将许多 UI 部分的组件都进行了修改, 让他们能够用户自定义化, 并且 3.0 在更新、安装方面引入了 OSGi 机制, 也就是这两大变化, 使得 Eclipse 真正成为了 RCP。

2.4. Eclipse RCP 应用介绍

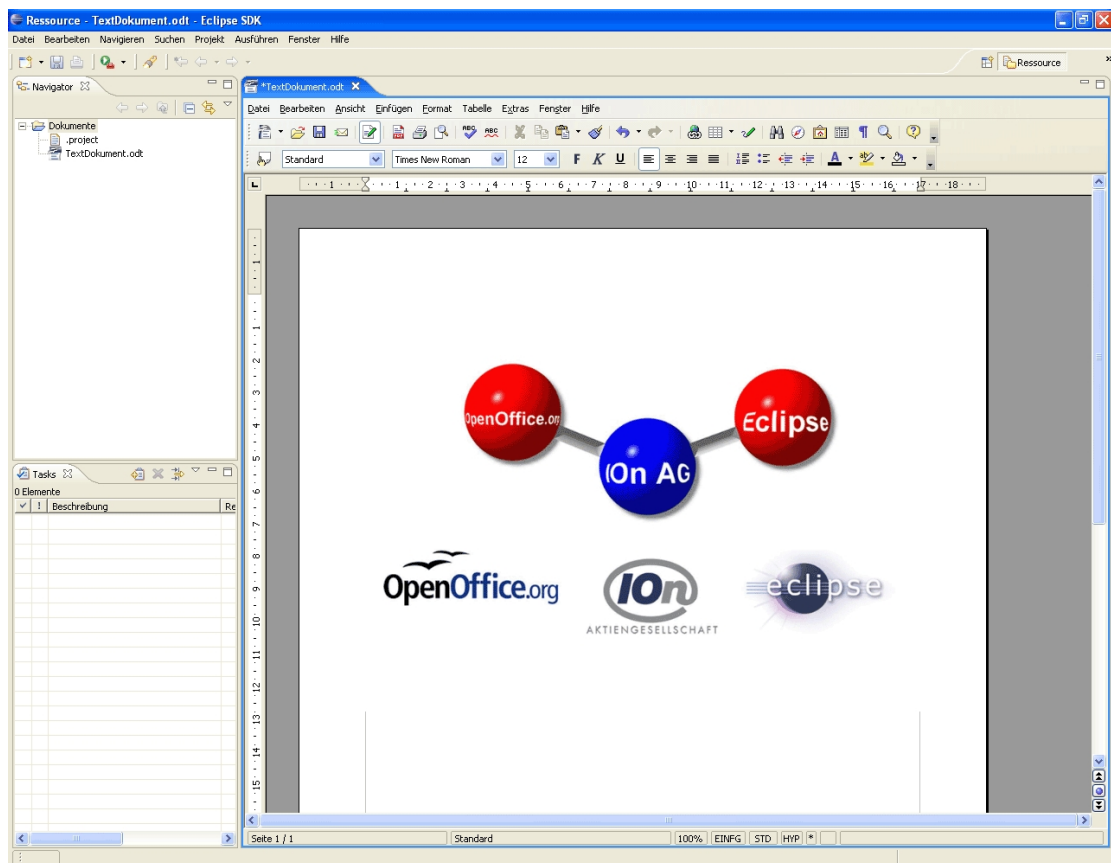
➤ jLibrary

jLibrary 是一款开源的 CMS 系统, 它将自己的 CMS 编辑客户端放在 Eclipse RCP 上来做。



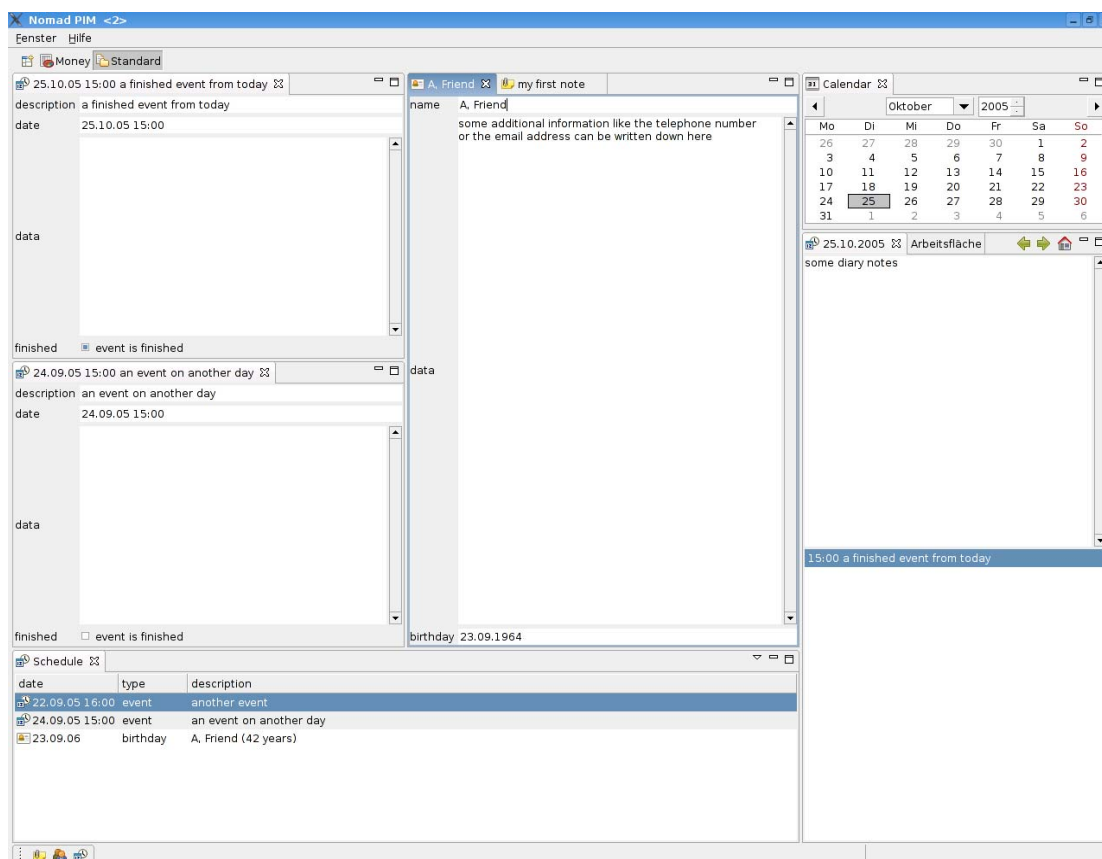
➤ NOA Office Integration Editor

这是集成了 NOA Office 的 RCP 应用, 能够让 Open Office 操作集中到 RCP 上来。



➤ **Nomad PIM**

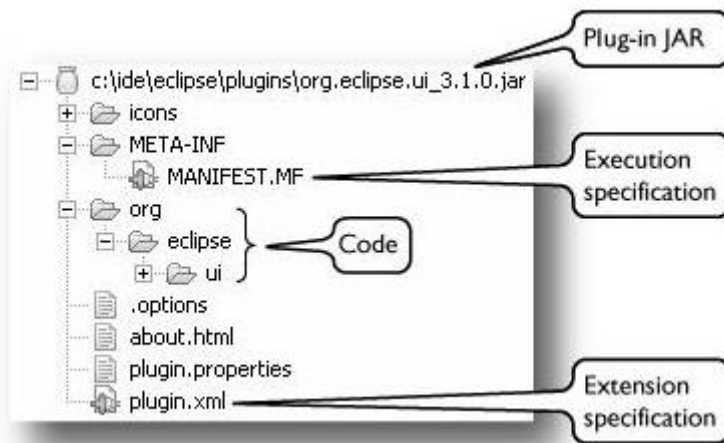
Nomad PIM 是一款简易的个人信息管理系统，可以对个人信息例如日程、支出、日记等进行管理。



2.5. 进一步了解 Eclipse RCP

2.5.1. Eclipse 插件

我们近距离看看 Eclipse 插件。Eclipse 插件其实和一般的 Java 工程类似，但是不同于普通的 Java 工程的是，Eclipse 插件工程的组织安排都有一些特殊的配置文件进行维护。一个插件工程是有一些列的 Java 文件和一个进行描述和连接其他依赖插件的 manifest 文件组成，其中还包括一个名为 plugin.xml 的配置文件，该文件是描述该插件项目具体扩展了哪些 Eclipse 的扩展点。下图是一个 Eclipse 工程的截图：

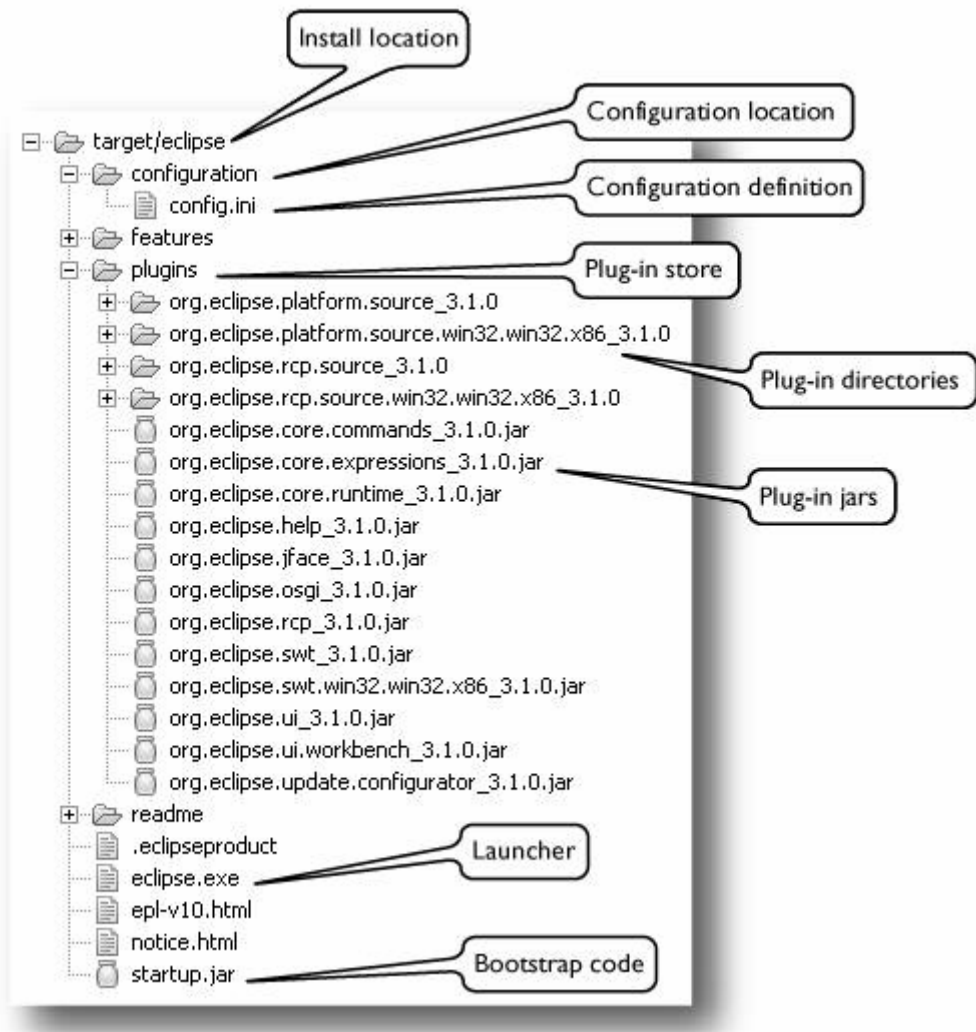


插件工程包括有代码或者一系列的资源文件，例如图片、Html 文件等、Web 页面、文档等，如上图所示，该插件工程的代码包含在 `org/eclipse/ui..` 文件架下，另外还包括了 `about.html` 文件以及 `plugin.properties` 资源文件。

请注意该工程下的 `plugin.xml` 文件，从 Eclipse3.1 开始，大部分信息比如 `classpath`、依赖插件项等信息，都储存在了 `manifest.mf` 文件中，`plugin.xml` 的功能延续了 Eclipse 以前的扩展点描述以及扩展点的定义。

2.5.2. Eclipse RCP 系统概要

Eclipse RCP 整个系统又是什么样子的呢？最顶层的文件架是 RCP 安装目录，它的子目录中包含了一个 `plugins` 目录，这里是所有 RCP 所需要的 `plugin`，以及开发者所部署的 `plugin`。此外还有一些 RCP 的启动代码和可执行文件 `eclipse.exe`。



二. Eclipse RCP 开发 (1)

1. 简单 CRM 系统

我们将通过利用 Eclipse 构建一个简单的 CRM 系统 RCP 应用，来深入讲解 Eclipse RCP。

CRM（客户关系管理系统）是目前流行的一种应用系统，而且有相当多的公司都开发了自己的 CRM 系统，其中包括有一些是利用 web 构建的系统，当然也不乏桌面应用程序。

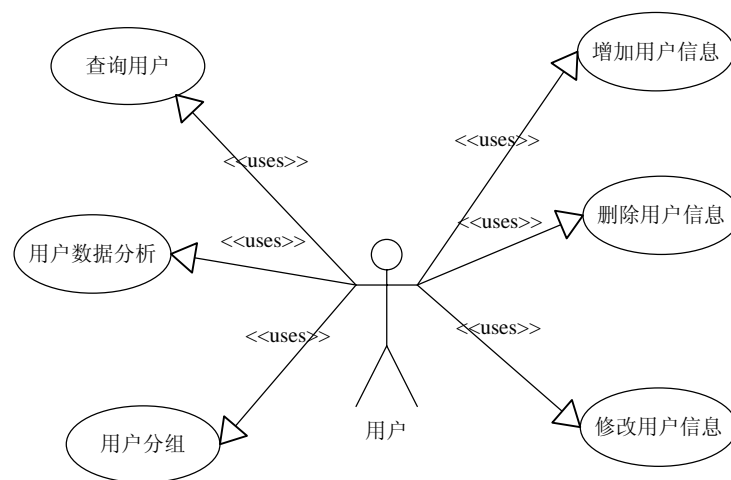
在这里，我们不会去深入研究如何构建一个合格的 CRM，我们就一些简单的需求来创建我们的后台模型，再利用 RCP 搭建一个扩展性强的 CRM 应用。

在构建这个简单的 CRM 的过程中，我们的需求在不断地变化，这一方面模拟了在真正构建一个系统的开发过程，另一方面我们也通过需求的改变来扩充我们的应用，你会发现 RCP 强大的插件机制是如何进行扩展功能的。

1.1 CRM 的需求分析

我们前面已经讲过了，这个 CRM 系统是一个简单的示例程序，主要是为了演示如何构建一个 RCP，所以我们的需求也会很简单。

先让我们看看这个 CRM 能做些什么。



上面的用例非常简单，仅仅是一些常见的客户信息管理（增、删、改、查）：

- 客户可以增加一个用户信息
- 客户可以删除一个或多个用户信息
- 客户可以对用户信息进行修改
- 客户可以对他的用户进行分类管理
- 对于客户的数据，我们可以进行简单的分析得出分析结果。
- 客户可以通过自己定制的条件对用户进行查询搜索。

对于以上的这些简单用例，我们可以得到以下的一些信息：

- 系统中主要以存储客户信息为主

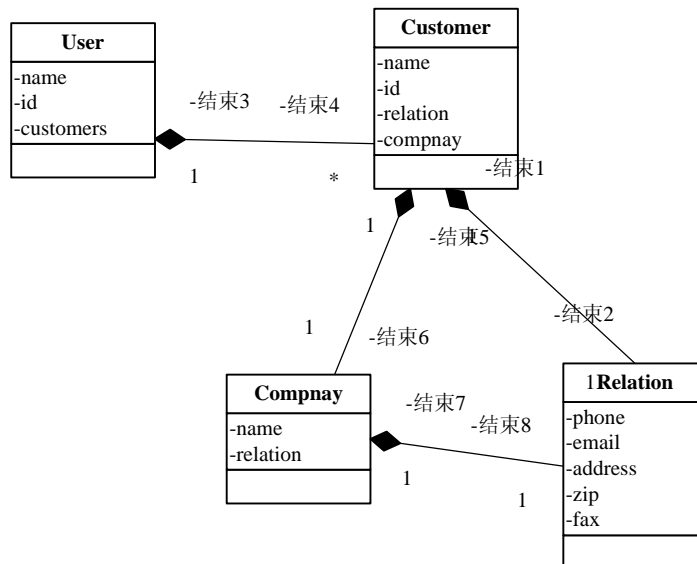
➤ 系统可能是多个用户使用

➤ 客户信息除了一些个人信息以外还需要一些其他的信息，比如所在公司、联系方式等

我们现在就以目前的这些需求作为出发点，开始设计我们的 CRM RCP 应用。

1.2 模型设计

这里我们所说的模型其实就是一些 java 类，这些 java 类反应了以上所搜集的信息的一个反应，RCP 程序中，很多显示部件都采用了 MVC 模式，模型和视图之间为松耦合，所以我们这里设计出的模型也是为了迎合 RCP 程序中的一些需要。



➤ User

该类包含了姓名、ID 等属性，并且包含了多个 Customer 类实例。

➤ Customer

除了简单的一些基本信息以外，Customer 还具有所在公司（Compnay）以及联系方式（Relation）。

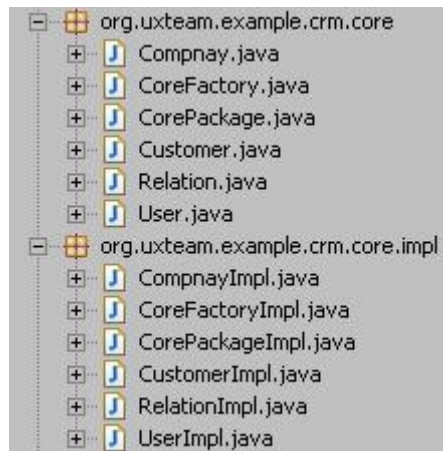
➤ Company

简单的一些公司信息

➤ Relation

包括目前常用的一些联系方式：电话、电子邮箱、地址等

我们通过上面简单的类图结构，设计出以下的代码：



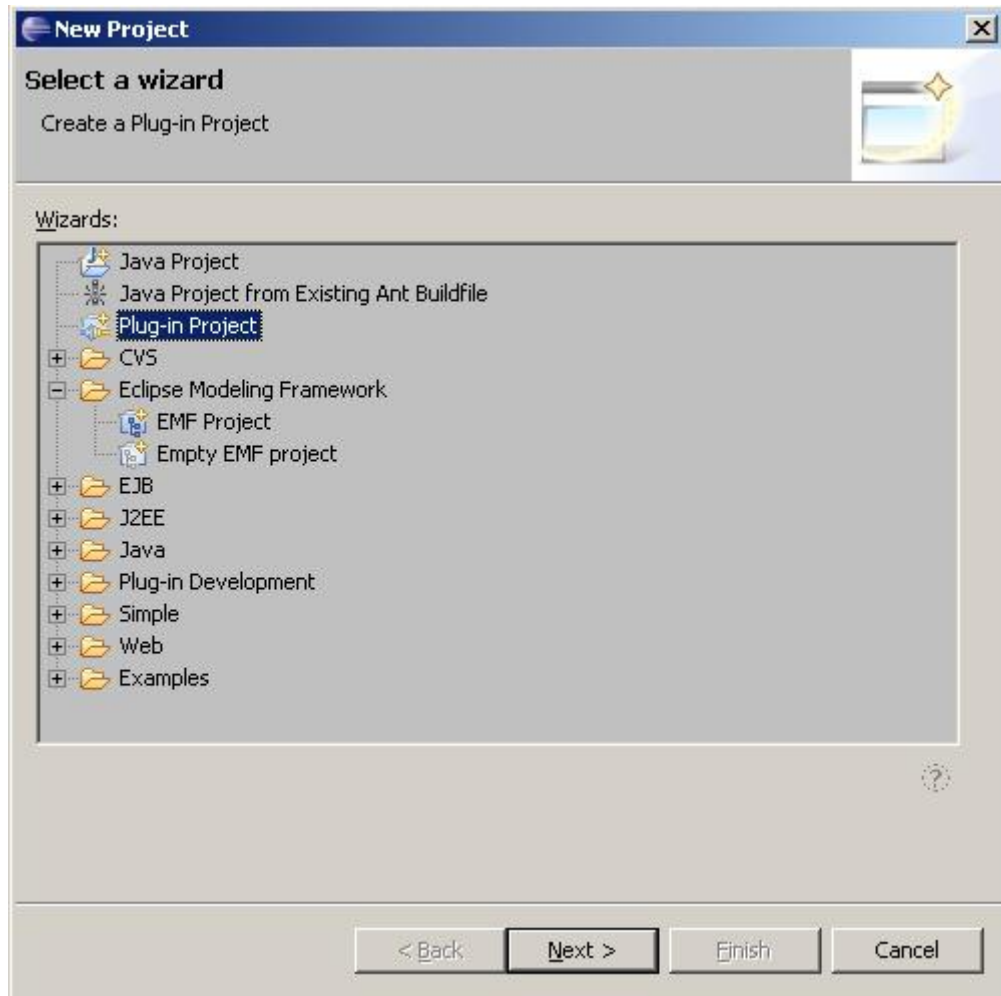
这些代码其实是利用 EMF 建模生成的，其中的 CoreFactory 以及 CorePackage 接口以及其实现是描述模型以及创建模型的单态类，虽然我们没有介绍 EMF，但这并不影响我们的下一步开发。

2. 利用向导生成我们第一个 RCP 应用

我们选用了 Eclipse 3.1 版本，整个 CRM RCP 应用将基于它来开发。

我们首先看看如何利用 Eclipse 提供的向导生成一个简单的 RCP 应用。

打开 Eclipse 之后我们会在菜单项中找到 File 这项，然后选择 New->Project，我们将会进入一个创建向导，请选择 Plugin-Project 后点击 Next 按钮：



根据向导提示，我们输入了项目名后进入到下一个向导页面，在这里我们需要注意，创建一个独立的 RCP 程序，需要将单选按钮项选择到 Yes：

New Plug-in Project

Plug-in Content
Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Classpath:

Plug-in Class

☒ Generate the Java class that controls the plug-in's life cycle
Class Name:

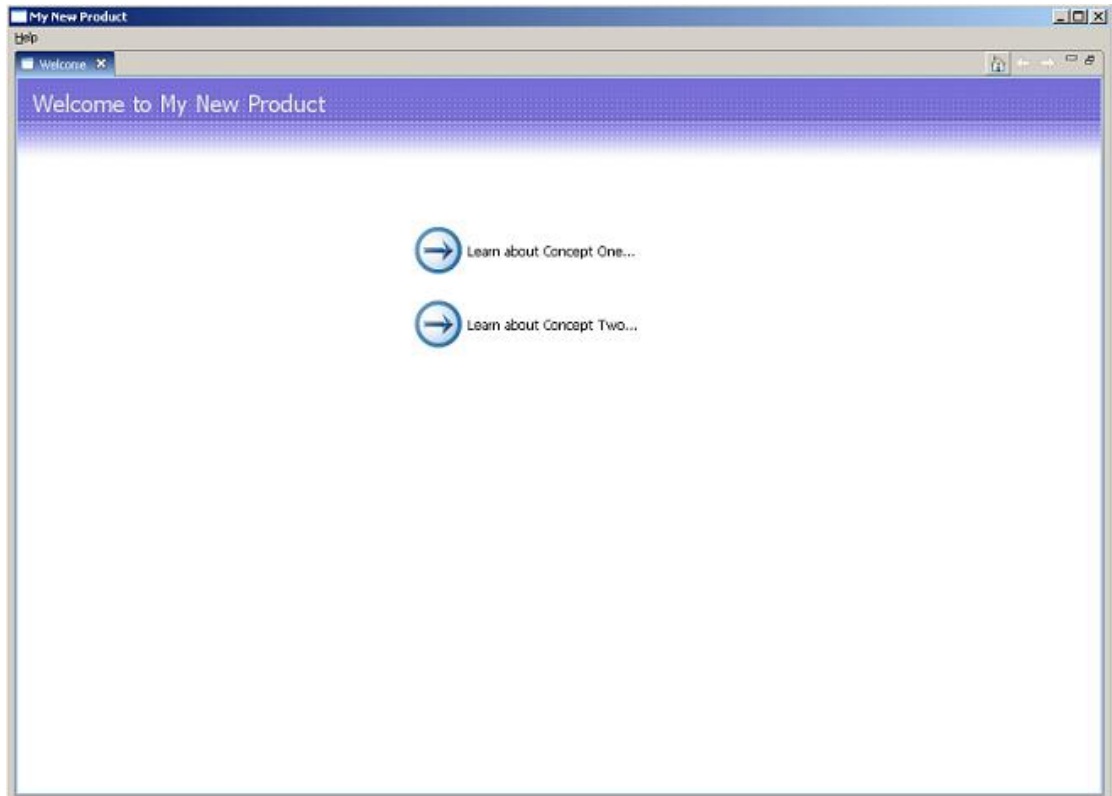
☒ This plug-in will make contributions to the UI

Rich Client Application
Would you like to create a rich client application? ☒ Yes ☐ No

< Back Next > Finish Cancel

这样我们生成的插件工程其实就是一个独立的 RCP 工程，向导会自动为我们生成 RCP 所需要的一些配置文件以及扩展点等项。

在点击 Next 后，我们选择好一个示例模板：RCP application with a intro 后点击 Finish，这样我们的第一个简单的 RCP 建立完成。运行后效果如下：



以后的开发，我们将基于上述这个例子进行扩展。

3. RCP 的 Application 以及 Product

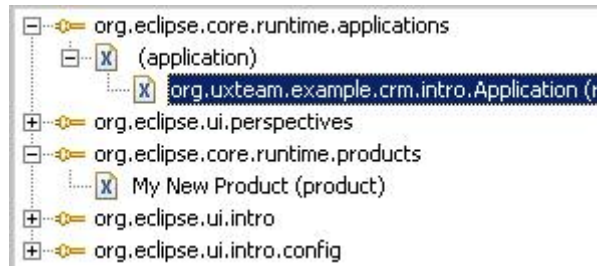
一个 RCP 应用所需要的不仅仅是一个完整的 RCP 依赖插件集以及开发的插件，还需要我们去实现一些扩展点。

其中有两个扩展点：一个是 Application，一个是 Product。

Application 扩展点定义了一个 application 类，这个类可以说是一个 RCP 应用的入口，它实现了 IPlatformRunnable 接口类，在 RCP 启动的时候将会执行这个接口的 run 方法。

而 Product 扩展点则是一个 RCP 应用的引导入口，它指定了该 RCP 工程将会去找那个 Application 进行运行，我们要在 RCP 安装的外部配置文件中指明该 RCP 的 Product ID，不过 Eclipse 3.1 让这些步骤都变得简单了许多。

看看下面的截图，我们会更清楚认识这两个扩展点：

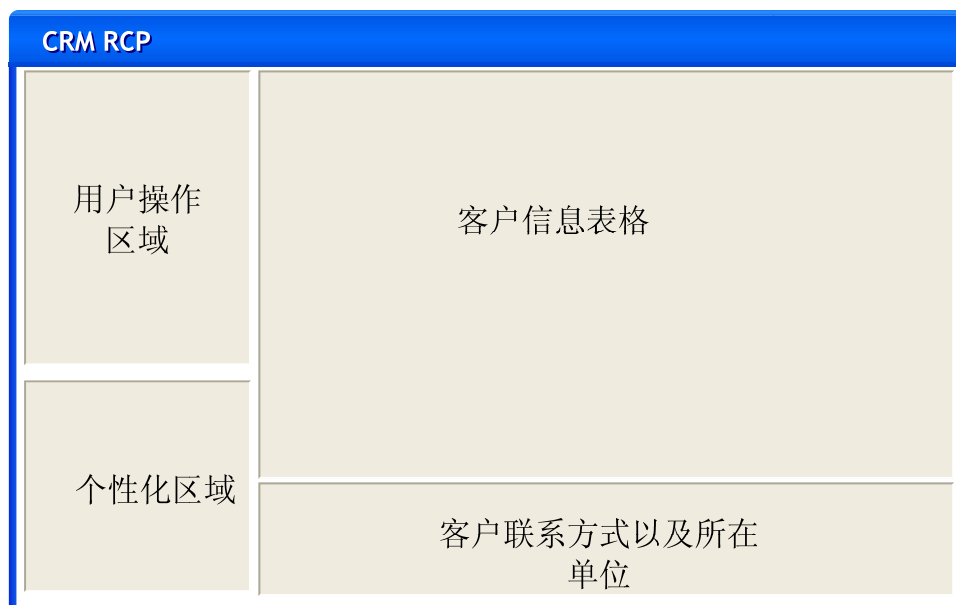


对于这两个扩展点，开发人员在开发的时候其实可以忽略掉，因为 Eclipse 所提供的工具能够很好地维护以及帮我们动态生成它们，而开发人员更多地则需要关注一下其他扩展点的开发以及我们的业务逻辑。

4. CRM 的整体 UI 设计以及 Perspective 扩展点。

我们设想一下所要做的 CRM 应用的界面布局应该是个什么样子。

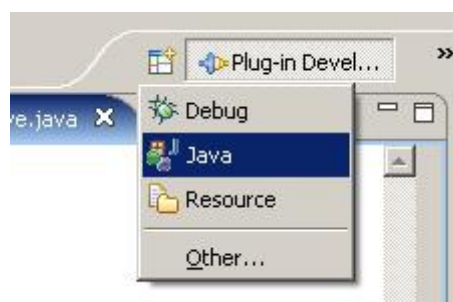
首先，我们需要一个展现所有客户信息的表格，这个表格应该放置在显著位置即应用程序的中心位置。在这个表格的左边，我们会放置一些个性化的元素，比如日历、天气信息，以及一个操作栏，包括增加、删除、查询客户。在客户信息表的下方我们将会显示出客户的联系方式以及他所在单位的一些情况。诸如此类的一些界面安排。



上面所显示的界面安排其实在很多情况下是不可靠的，比如现在我们需要对所有客户所

在的单位进行一些操作以及浏览，那么我们又如何去安排这个 UI 界面呢？

Eclipse RCP 在这方面替开发者考虑得很周全，它提出了 *Perspective* 的概念。*Perspective* 其实并不是一个实在的东西，它是对一个界面布局的说明，比如上面的界面安排，我们可以定义一个 *Perspective* 来进行描述，当我们需要对客户单位进行编辑浏览时，我们就可以定义另一个类似的 *Perspective*，这样，当我们需要进行什么样的操作时就切换到相应的 *Perspective* 下，如此一来就使得界面布局更加的容易并且给用户的体验也比起以往的弹出对话框要强很多。



如何去定义一个 *Perspective* 呢？一个 *Perspective* 是利用 `org.eclipse.ui.perspective` 扩展点扩展得到的，让我们看看这个扩展点：



从图中可以看出，这个扩展点需要我们去实现一个类，这个类需要实现 `IPerspectiveFactory` 接口类，在这个接口中，有一个 `createInitialLayout` 方法，每当我们切换或者主动调用 *perspective* 时，Eclipse 就会去调用该方法。

举一个简单的例子，代码如下：

```

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

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

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

```

首先看第一行代码，这里获得的是一个 editorArea 的字符串，这个字符串是一个标识，它将会应用待以下的代码中，比如在第三行代码中：

```

layout.addStandaloneView(NavigationView.ID,
    false, IPageLayout.LEFT,
    0.25f, editorArea);

```

Layout 将 NavigationView 元素放置在了 editorArea 的左边，这里的 editorArea 就是标识 layout 的编辑区域。第三个参数 0.25f 是表明，NavigationView 将会在 editorArea 左边所在整个布局的 25%大小。

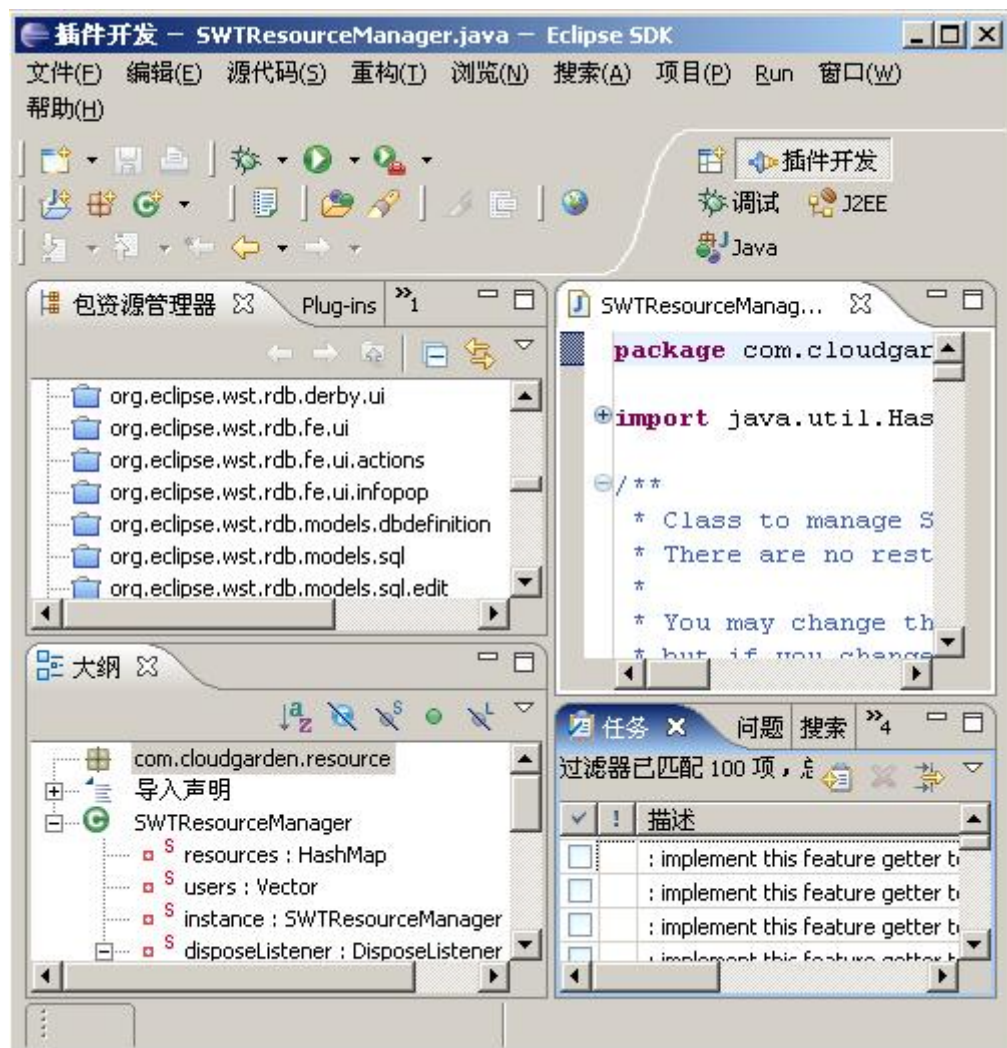
这里我们简单介绍了一下 Perspective，后续的开发中我们在逐步完善代码时将会再次讨论到它。

5. Eclipse Workbench 介绍

我们将通过整个 CRM 系统的制作过程来讲述 Eclipse RCP 的开发，在这个过程中，我们会涉及到工作台的操作以及它如何使用视图和编辑器来显示信息。

工作台是浏览由插件提供的所有功能的场合。通过使用工作台，我们可以浏览资源并且可以查看和编辑这些资源的内容和属性。

当在一组项目上打开工作台时，它将为如下所示。



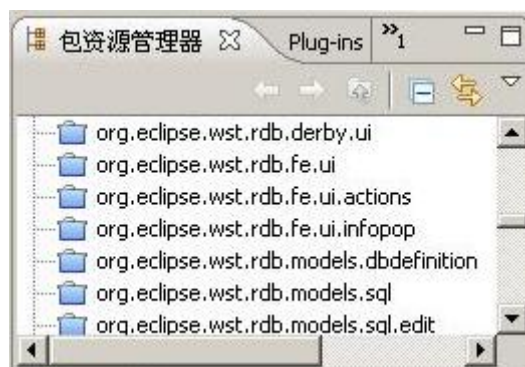
工作台只是一个可以提供各种可视部件的框架。这些部件属于两个主要类别：**视图**和**编辑器**。

- **编辑器**允许用户在工作台中编辑某些内容。编辑器是“以文档为中心的”，它很像文件系统编辑器。与文件系统编辑器相似，它们遵循“打开 — 保存 — 关闭”这一循环。与文件系统编辑器不同的是，它们与工作台紧密集成。

- **视图**提供了关于用户正在工作台使用的一些对象的信息。当用户选择工作台中的不同对象时，视图通常就会更改它们的内容。视图通常通过提供有关活动编辑器中的内容的信息来支持编辑器。

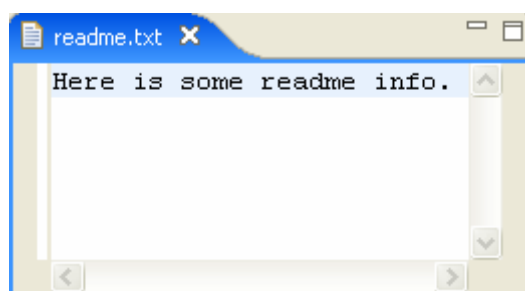
View （视图）

工作台提供了几个标准视图，它们允许用户浏览或者查看感兴趣的内容。例如，资源导航器允许用户浏览工作空间和选择资源。



Editor (编辑器)

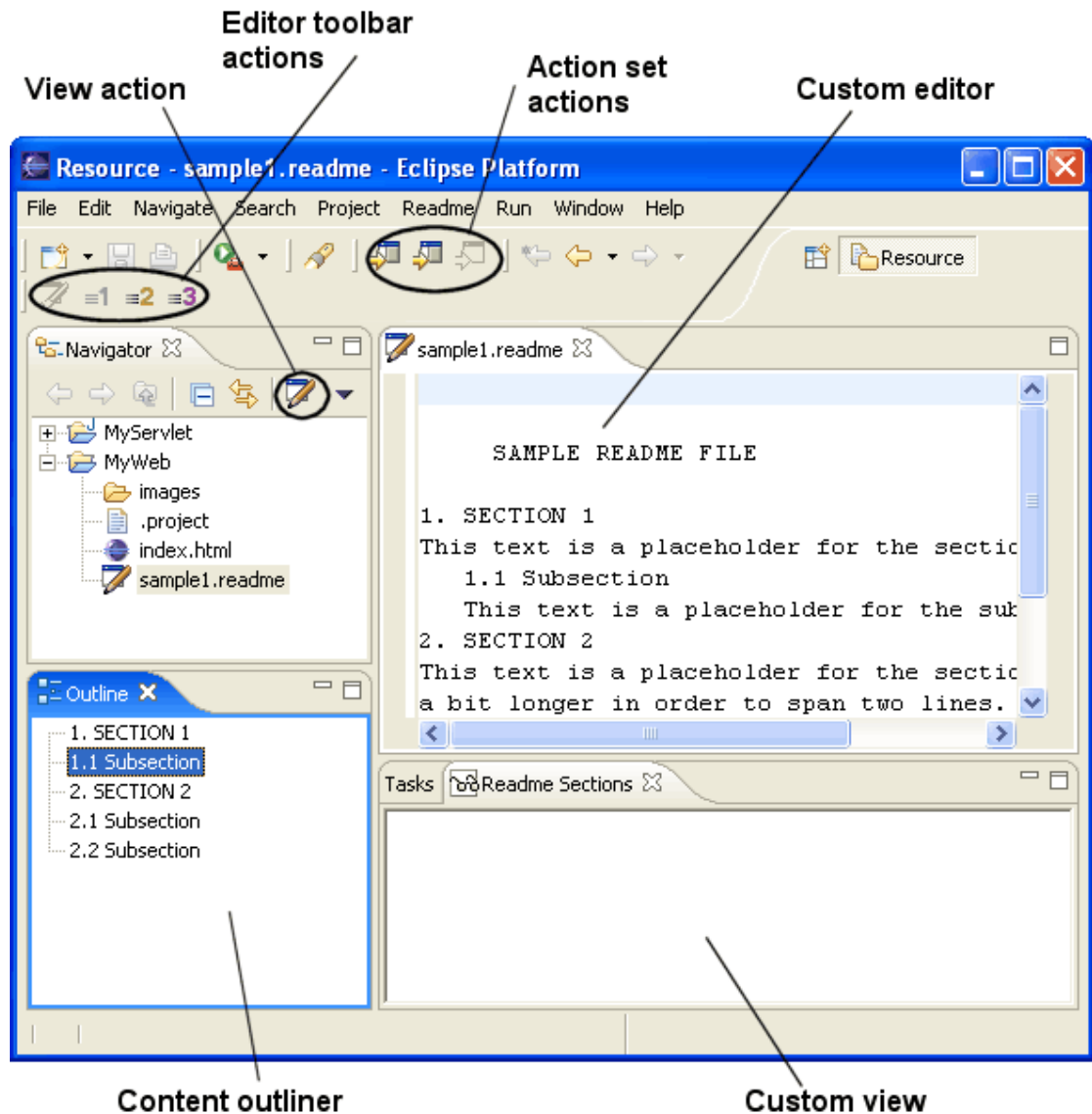
编辑器允许用户打开、编辑和保存对象。工作台为文本资源提供了标准编辑器。



可以由插件来提供附加编辑器，例如，Java 代码编辑器或者 HTML 编辑器。

5.1 基本工作台扩展点

工作台定义扩展点，这些扩展点允许插件将行为添加至现有视图和编辑器，或为新视图和编辑器提供实现。我们将从我们的 CRM RCP 中来学习这些扩展点的添加项。



5.2 扩展一个 view

我们在上面的介绍中提到了，我们的 CRM 系统中主要是针对客户数据做文章，在这里我们要实现一个表格，让它显示出客户的信息。

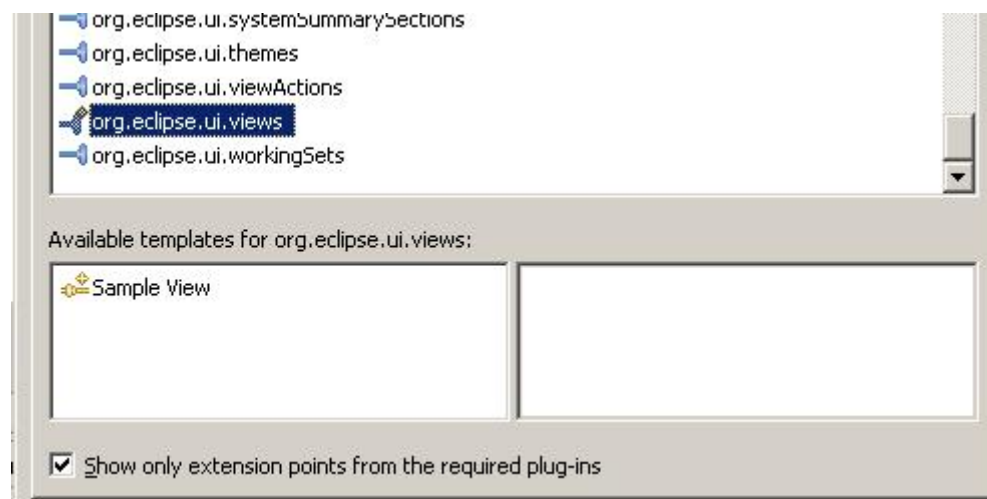
视图是可以浏览信息层次结构或显示对象属性的工作台部件。在一个工作台页面中，只能打开任何给定视图的一个实例。当用户在视图进行选择或其它更改，这些更改会立即显示在工作台中。提供视图通常是为了支持相应的编辑器。例如，大纲视图显示编辑器中的信息结构化视图。属性视图显示当前编辑的对象的属性。

扩展点 `org.eclipse.ui.views` 允许插件将视图添加到工作台中。添加视图的插件必须在它们的 `plugin.xml` 文件中注册该视图，并提供有关该视图的配置信息，例如，它的实现类、它所属的视图的类别（或组）以及应该用来在菜单和标签中描述该视图的名称和图标。

视图的接口是在 `IViewPart` 中定义的，但是插件可以选择扩展 `ViewPart` 类而不是根据暂存区来实现 `IViewPart`。

在 RCP 中，`View` 是一个很重要的显示部件，它几乎在 RCP 中起着主角的作用，很多相关的一些 UI 显示都是以 `View` 为载体而实现出来的。下面看看如何扩展一个 `View`。

我们打开 `MANIFEST.MF` 文件，选择 `Extentions` 页面，然后选择 `Add, Eclipse` 将会显示出目前可以扩展的扩展点，我们增加一个 `org.eclipse.ui.view` 扩展点：



然后在它上点击右键，选择 `view`，我们扩展了这个 `views` 扩展点。我们需要完善这个 `view` 扩展点的一些扩展点属性，比如 `id`, `name`, `class`。以下是 `Plugin.xml` 中的扩展点 XML 片断。

```
<extension
    point="org.eclipse.ui.views">
    <view
        class="org.uxteam.example.crm.ui.CustonmerViewPart"
```



```
        id="org.uxteam.example.crm.customerview"
        name="customerview"/>
    </extension>
```

这里 class 很重要，它是实例化这个 ViewPart 的入口类，我们将这个类命名为 CustommerViewPart，并让它继承 ViewPart 类，这样我们得到了以下代码：

```
public class CustommerViewPart extends ViewPart {
```

```
    /**
     *
     */
    public CustommerViewPart() {
        super();
        // TODO Auto-generated constructor stub
    }

    /* (non-Javadoc)
     * @see org.eclipse.ui.part.WorkbenchPart
     * #createPartControl(org.eclipse.swt.widgets.Composite)
     */
    public void createPartControl(Composite parent) {
        // TODO Auto-generated method stub
    }

    /* (non-Javadoc)
     * @see org.eclipse.ui.part.WorkbenchPart#setFocus()
     */
    public void setFocus() {
```

```
// TODO Auto-generated method stub

}

}
```

我们可以看到这里有一个叫 `createPartControl` 的方法，这个方法是创建整个 `ViewPart` 界面的方法体。我们在这个方法中加入以下代码：

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

label.setText("这是一个 ViewPart 的测试");
```

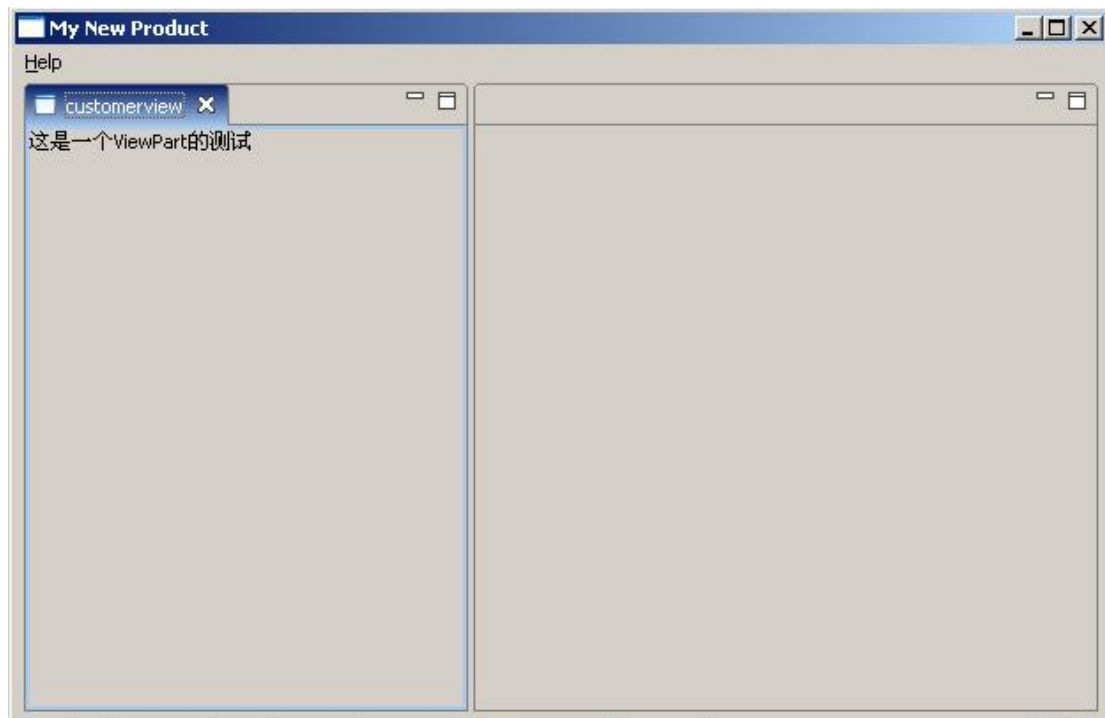
目前我们所做的工作是简单的在这个 `ViewPart` 上显示一个 `Label` 控件，但是目前这个 `ViewPart` 还没有办法显示在我们的 RCP 应用上，这是因为我们没有任何操作让它打开，这里我们就要回到前一节的 `Perspective` 上了。

我们在 `Perspective` 的 `createInitialLayout` 方法中加入以下代码：

```
String editorArea = layout.getEditorArea();

layout.addView("org.uxteam.example.crm.customerview",
IPageLayout.LEFT, 0.25f, editorArea);
```

然后我们运行一下我们修改过后的 RCP，得到下面的结果：



我们所设计的 ViewPart 显示了出来，上面也正是我们所创建的 Label 控件。

目前来看，我们已经搭建起来了这个 ViewPart，现在我们需要将 Customer 的表格搭建起来了。

在 RCP 中，构成 UI 元素的是由 SWT/JFace 提供的 UI 部件，这里我们将采用 JFace 的 TableViewer 来创建我们的 Customer 表格视图。

先看看以下几个概念：

➤ MVC

MVC（模型—视图—控制器模式）是一个比较古老的模式，它把显示和后台模型进行了解耦，利用控制器来控制视图并把视图的变化反应回模型或把模型的改变提交给视图。

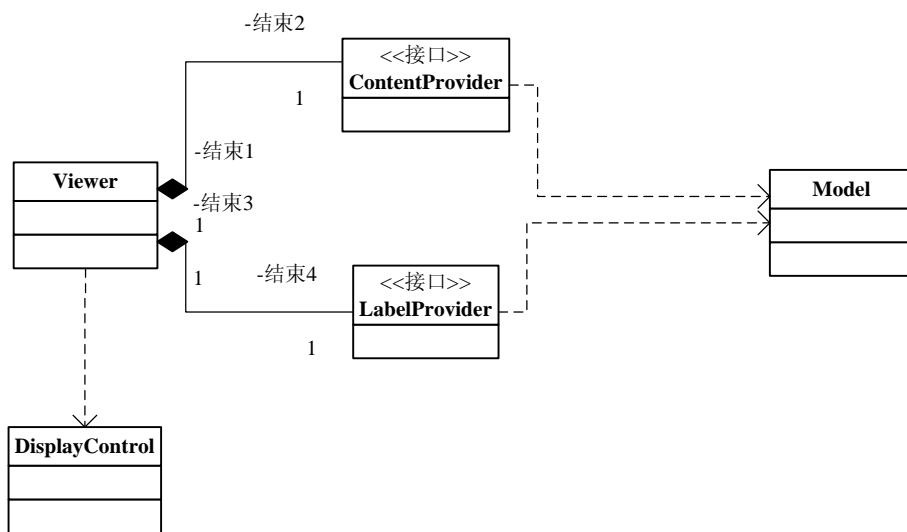
➤ ContentProvider

内容提供者。JFace 中的所有 View（包括 TableViewer、TreeViewer、ListViewer 等），都具有这么一个对象：ContentProvider，这个内容提供者其实就相当于 MVC 模式中的 M（模型），可是 JFace 中并没有完全依赖模型，而是提出了 Provider 来提供给 Viewer 模型，这样一来，模型和视图之间的耦合又大大降低了。

➤ LabelProvider

显示提供者。如同上面的内容提供者一样，显示提供者也是分担了 View 显示模型的工作，View 不关系模型该如何去显示，而是让显示提供者去返回该显示的字符串以及图标等显示元素。

在 JFace 中一个 Viewer 具有 LabelProvider 和 ContentProvider 为一身，一个负责显示提供者的维护一个负责模型提供者的维护，而真正去显示和加载模型的并不是 Viewer 本身。



回到我们的 CRM 系统中。

我们需要显示的是我们的 Customer 对象模型，我们现在就在刚才所创建的 CustomerViewPart 上创建一个 TableView:

```
viewer = new TableView(parent, SWT.MULTI
|SWT.FULL_SELECTION);

TableColumn nameColumn =
    new TableColumn(viewer.getTable(), SWT.NONE);
    nameColumn.setWidth(20);

TableColumn sexColumn =
    new TableColumn(viewer.getTable(), SWT.NONE);
    sexColumn.setWidth(20);
```

```

TableColumn bornDateColumn =
    new TableColumn(viewer.getTable(), SWT.NONE);
    bornDateColumn.setWidth(100);
TableColumn postColumn =
    new TableColumn(viewer.getTable(), SWT.NONE);
    postColumn.setWidth(20);
    viewer.getTable().setHeaderVisible(true);
    viewer.getTable().setLinesVisible(true);

    viewer.setContentProvider(
new CustommerContentProvider());

    viewer.setLabelProvider(
new CustommerLabelProvider());

```

我们构造了一个具有四列的表格，并且设置了 `CustommerContentProvider` 和 `CustommerLabelProvider`。

下面看看这两个类的具体实现。

➤ `CustommerContentProvider`

这个类实现了 `IStructuredContentProvider` 接口。

整个类最主要的方法是 `getElements` 方法，这个方法传入的参数 (`inputElement`) 即我们提供给 `TableViewer` 的模型，我们设这个模型是一个 `Customer` 的 `List`，得到下面的代码：

```

public Object[] getElements(Object inputElement) {
    if(inputElement instanceof List){
        return ((List)inputElement).toArray();
    }
    return null;
}

```

➤ CustommerLabelProvider

这个类实现了 ITableLabelProvider。

这个类中有 getColumnImage 和 getColumnText 两个方法，分别是返回对应列的图标以及显示字符串。我们先不考虑显示的图标，仅仅考虑返回的字符串：

```
public String getColumnText(Object element, int columnIndex) {  
    if(element instanceof Customer){  
        switch(columnIndex){  
            case 0:  
                return ((Customer)element).getName();  
            case 1:  
                return ((Customer)element).getSex();  
            case 2:  
                return ((Customer)element)  
                    .getBornDate().toGMTString();  
            case 3:  
                return ((Customer)element).getPost();  
            default : break;  
        }  
    }  
    return null;  
}
```

我们在创建 TableViewer 的时候，适当加入几个元素到其中，可以得到以下的结果：

张三	男	4 Apr 2006 17:53:04 GMT	开发工程师	..
李四	男	4 Apr 2006 17:53:04 GMT	项目经理	..
赵六	女	4 Apr 2006 17:53:04 GMT	售前工程师	..

注意：我们这里所得到的数据都是手动输入的，目前还没有一个持久化的数据源提供数据。在以后的开发中我们都会以外部 XML 文件作为我们的数据源。

5.3 扩展 Actions

当我们做完了以上工作的时候，会发现一个问题，就是如果我们要显示 Customer 表格 ViewPart 的时候，无法找到入口去打开它，就是说在 RCP 的界面上无法找到合适的动作让这个 ViewPart 打开，当然，在 Eclipse 本身的 IDE 中，我们可以通过 Windows->Show View 打开一个 ViewPart，但是我们的 CRM RCP 中没有这个选项。当然，我们可以将 Eclipse UI 中的 Show View menu 注册进来，这会在后面提到，这里我们只是想通过一个在 Toolbar 或者是 ActionBar 上的按钮打开这个 ViewPart。

Eclipse 的扩展点中，有一个名位 org.eclipse.ui.actionSets 的扩展点，这个扩展点是让我们在 Eclipse 的 workbench 上注册我们的 Action，即在 Toolbar 和 ActionBar 上注册一个按钮。

首先，我们在 plugin.xml 中加入以下的代码：

```

<extension
    point="org.eclipse.ui.actionSets">
    <actionSet
        id="org.uxteam.example.crm.actionSet"
        label="Open View Part Action Set"
        visible="true">
        <menu
            id="sampleMenu"
            label="Open View">
            <separator name="sampleGroup"/>
        </menu>
        <action
            class="org.uxteam.example.crm.actions.SampleAction"
            icon="icons/sample.gif"
            id="org.uxteam.example.crm.actions.SampleAction"
            label="&Open Customer View"
            menubarPath="sampleMenu/sampleGroup"
            toolbarPath="sampleGroup"
            tooltip="Open the customer table viewpart"/>
        </actionSet>
    </extension>

```

我们分析一下上面的 XML 片断：

➤ **actionSet**

actionSet 可以看作一个 actionSet 的根元素，它可以包含 menu、action 等元素，这些元素用来描述这个 actionSet 都扩展了哪些 action 和 menu，并且这些 action 和 menu 的显示文字，加载路径以及对应执行类是什么。

➤ **menu**

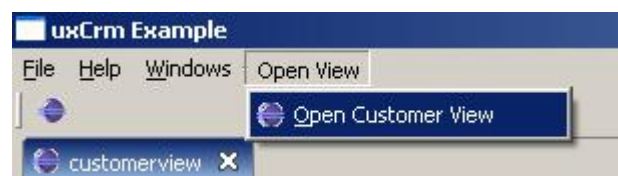
表示一个 menu 项，它将会出现在 Workbench 的 Menu 项中，在它属性中的 label 指明该 menu 的显示名字。

➤ action

action 不仅仅表示一个菜单项中的 menuitem，它可以表示一个在 toolbar 上的按钮，可以这么认为，action 是一个描述菜单项和 toolbar 上按钮对应的动作元素。

Class 属性表示该 action 执行时所对应的类；icon 表示它显示的图标；label 表示显示名；menubarPath 是指该 action 出现在菜单项的位置；toolbarPath 是指 action 出现在工具栏上的位置。

我们可以运行一下看看效果：



在 RCP 的菜单栏中我们发现新增了一个名为 Open View 的菜单项，它包含了一个名为 Open Customer View 的子菜单项，这正好和上面扩展点描述对应上：menu 即 Open View 项；action 则是 Open Customer View 项目，同样，我们会发现在工具栏中多出来一个按钮，这个按钮就是我们的 action，是我们制定的 toolbarPath 将它定位在此处的。

我们点击 Open Customer View 项和点击 toolbar 上的那个按钮所执行的代码则是在 action 元素中的 class 属性制定的类。

现在我们创建这个类。

新建一个名为 org.uxteam.example.crm.actions.SampleAction 的类，这个类需要实现 IWorkbenchWindowActionDelegate 接口：

```

public class SampleAction implements
IWorkbenchWindowActionDelegate {

    private IWorkbenchWindow window;

    public SampleAction() {
    }

    public void run(IAction action) {
        try {

            window.getActivePage().showView("org.uxteam.example.crm.custom
erview");

        } catch (PartInitException e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

    }

    public void selectionChanged(IAction action, ISelection
selection) {

    }

    public void dispose() {

    }

    public void init(IWorkbenchWindow window) {

        this.window = window;

    }

}

```

当我们点击菜单或者是工具栏上的按钮后，将会执行 SampleAction 的 run 方法。这个方法中，我们打开了一个 ViewPart，这个 View 的 ID 为

```
org.uxteam.example.crm.customerview:
```

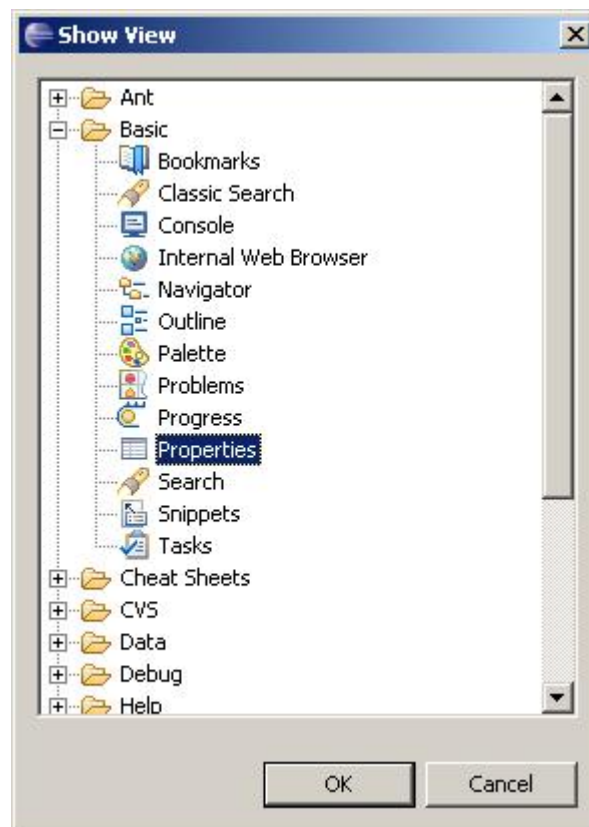
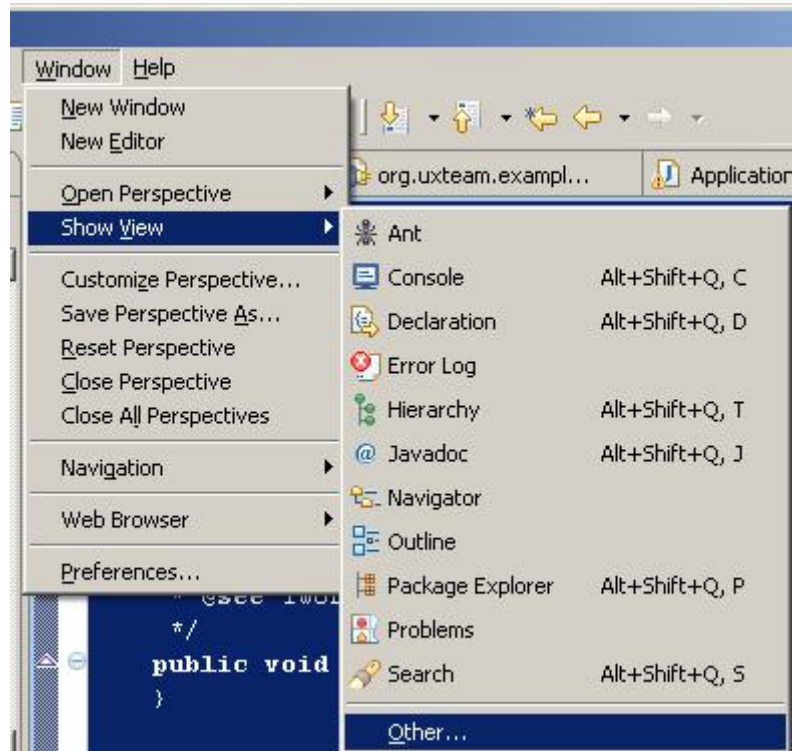
```
try {  
  
    window.getActivePage().showView("org.uxteam.example.crm.custom  
erview");  
  
    } catch (PartInitException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

这就是我们定义的 CustomerView 所对应的 ViewPart ID。当我们点击后 CustomerView 将会被激活并显示出来；如果 CustomerView 已经存在，那点击后 CustomerView 将会被设置焦点（forcus）。

5.4 扩展 PopMenu

我们通过上面的方法，实现了通过一个按钮去激活、打开我们的一个 ViewPart。

但是一般情况下，我们的 ViewPart 将会很多，并且不可能可能为每一个 View 都做一个 Action 去显示它，在 Eclipse 中，我们要打开一个 ViewPart 是可以通过 Windows->Show View 的方法，打开一个 View 的选择对话框的：



在我们所做的 RCP 中是没有这么一个选项的，但是并不是说它就不存在。我们是可以对一些代码，让它显示在我们的 RCP 中。

打开 `org.uxteam.example.crm.intro. ApplicationActionBarAdvisor` 类，我们会找到有一个方法，名为 `fillMenuBar`，这个方法是覆盖了父类 `ActionBarAdvisor` 的方法，它的作用是在 RCP 启动的时候去填充 Workbench 的菜单项。

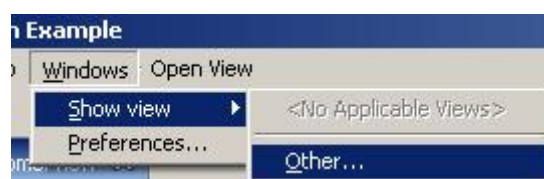
我们在 `makeActions` 方法中创建一个名为 `showList` 的 `IContributionItem` 对象：

```
showList = ContributionItemFactory.VIEWS_SHORTLIST.create(window);
```

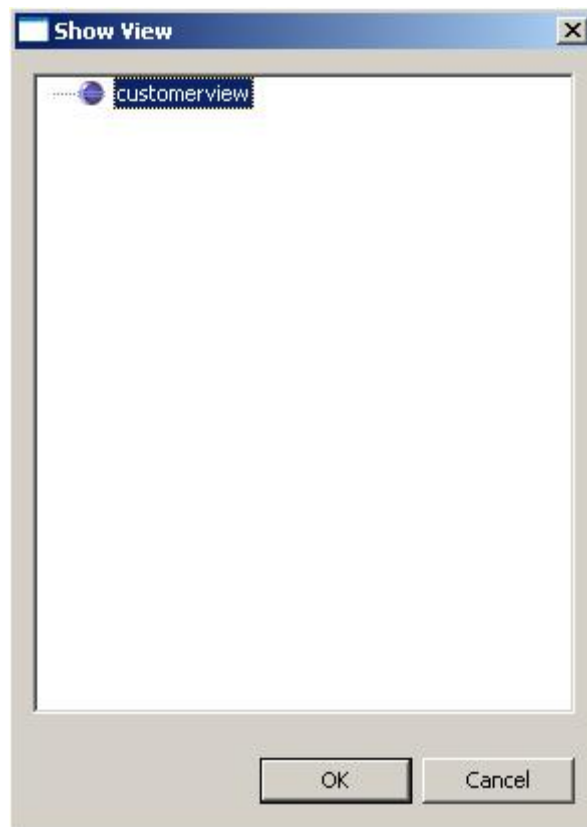
这个 `IContributionItem` 就是 workbench 上显示 `ViewDialog` 的菜单项，现在我们把它添加到一个新建的 menu 当中：

```
MenuManager showWindowsMenu = new MenuManager(
    "&Windows", IWorkbenchActionConstants.M_WINDOW);
MenuManager showViewMenu = new MenuManager(
    "&Show view", IWorkbenchActionConstants.SHOW_EXT);
showWindowsMenu.add(showViewMenu);
showViewMenu.add(showList);
```

从代码中我们可以看出，我们新增加了一个 `showWindowsMenu` 和一个 `showViewMenu` 菜单，并把新生成的 `IContributionItem` 项加入到了其中，这样我们在 CRM RCP 中就可以得到一个带有 `Windows->Show View` 的菜单项，并且它可以显示出 `ViewDialog` 供我们选择要显示的 `ViewPart`：



我们点击 Other 后会出现一个 ViewDialog:



这样一来我们就可以通过它来选择我们要显示的 ViewPart 了。

5.5 扩展 editor

编辑器是一个工作台部件，它允许用户编辑对象（通常是文件）。编辑器的运行方式类似于文件系统编辑工具，只不过它们紧密集成到平台工作台用户界面中。编辑器总是与输入对象（`IEditorInput`）相关联。可以将输入对象看作是正在编辑的文档或文件。在用户保存在编辑器中所作的更改之前，不会落实这些更改。

只能打开一个编辑器以供工作台页面中的任何特定编辑器输入使用。例如，如果用户正在工作台中编辑 `readme.txt`，则在同一透视图再次打开它时将激活同一编辑器。（可以在不同的工作台窗口或透视图对同一文件打开另一编辑器）。但是，与视图不同，在一个工作台页面中，可以对不同输入多次打开同一编辑器类型（例如，文本编辑器）。

插件使用工作台扩展点 `org.eclipse.ui.editors` 来将编辑器添加到工作台中。添加编辑器的插件必须在它们的 `plugin.xml` 文件中注册编辑器扩展以及编辑器的配置

信息。某些编辑器信息（例如，实现类以及要在工作台菜单和标签中使用的名称和图标）类似于视图信息。另外，编辑器扩展指定编辑器理解的文件类型的文件扩展名或文件名模式。编辑器还可以定义 `contributorClass` 类，当编辑器活动时，该类将操作添加到工作台菜单和工具栏中。

编辑器的接口是在 `IEditorPart` 中定义的，但是插件可以选择扩展 `EditorPart` 类而不是根据暂存区来实现 `IEditorPart`。

我们在 CRM 中需要用到 `Editor`，设想，每一个 `Customer` 具有一个 `Summry` 属性，这个属性是关于 `Customer` 的一些描述或者是摘要信息，所以我们需要一个专门的编辑器来编辑这一类的大文本对象。这个用户的 UI 体验也是不一样的，如果仅仅是一个简单的 `Text` 控件，当用户输入的时候往往会造成一些不便。

我们首先需要在 `plugin.xml` 中扩展 `org.eclipse.ui.editors` 扩展点：

```
<extension
    point="org.eclipse.ui.editors">
    <editor
        class="org.uxteam.example.crm.editors.CustomerSummryEditor"
        default="false"
        icon="icons/sample.gif"
        id="org.uxteam.example.crm.editor1"
        name="org.uxteam.example.crm.editor1"/>
</extension>
```

我们看看这个扩展点中的一些必要属性：

➤ **class**

`class` 在上面已经提到过，这是编辑器的入口类，需要实现 `IEditorPart` 接口。

➤ **icon**

编辑器对应的图标。

➤ **name**

编辑器显示的名字

每一个 Editor 在打开的时候是需要我们输入一个 EditorInput 对象的，这个对象必须实现 IEditorInput 接口，这是由于在 workbench 创建 Editor 的时候需要在 init 方法中带入这个 Input 作为参数。

所以我们在实现这个 Editor 之前需要实现这个 Editor 的 Input。

看看下面的代码，我们创建了一个类 TextEditorInput，它实现了 IEditorInput 接口，并且它聚合了一个 Customer 对象。当我们在初始化 Editor 的时候才能够得到这个 Customer 对象，便于我们对它操作：

```
public class TextEditorInput implements IEditorInput {  
    Customer customer;  
  
    public TextEditorInput(Customer customer){  
        this.customer = customer;  
    }  
  
    public boolean exists() {  
        return customer != null;  
    }  
  
    public ImageDescriptor getImageDescriptor() {  
        return null;  
    }  
  
    public String getName() {  
        return "";  
    }  
  
    public IPersistableElement getPersistable() {  
        return null;  
    }  
}
```



```

    public String getToolTipText() {
        return "";
    }

    public Object getAdapter(Class adapter) {
        return null;
    }

    public Customer getCustomer() {
        return customer;
    }

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }
}

```

在这些接口方法中, `exists` 表示该 `EditorInput` 是否存在, 我们这里判断输入参数 `Customer` 是否为空来设置它的返回值。

`getToolTipText` 和 `getName` 方法是表示这个 `Input` 的一些显示信息, 他们是不能为空的, 否则 `Workbench` 在创建 `Editor` 的时候会抛出异常。

当我们创建好这个 `EditorInput` 后, 就可以开始创建这个 `Editor` 了。

```

public class CustomerSummryEditor extends EditorPart {
    private Customer customer;
    private boolean dirty;
    private Text text;

    public void doSave(IProgressMonitor monitor) {
        customer.setSummry(text.getText());
        setDirty(false);
        firePropertyChange(PROP_DIRTY);
    }
}

```

```

    public void doSaveAs() {

    }

    public void init(IEditorSite site, IEditorInput input)
throws PartInitException {

        if(input instanceof TextEditorInput){

            customer = ((TextEditorInput)input).getCustomer();

            this.setSite(site);

            this.setInput(input);

        }

    }

    public boolean isDirty() {

        return dirty;

    }

    public boolean isSaveAsAllowed() {

        return false;

    }

    public void createPartControl(Composite parent) {

        text = new Text(parent, SWT.NONE);

        text.addModifyListener(new ModifyListener(){

            public void modifyText(ModifyEvent e) {

                setDirty(true);

                firePropertyChange( PROP_DIRTY );

            }

        });

        text.setText(customer.getSummary() ==

```

```

    null?"":customer.getSummry());
    }

    public void setFocus() {
    }

    public void setDirty(boolean dirty) {
        this.dirty = dirty;
    }
}

```

在这个类中，我们设置了一个 Dirty 属性，这个属性是表示该 Editor 的内容是否已经被更改，如果我们发射出被更改的信号，即调用 `firePropertyChange(PROP_DIRTY)` 方法后，Workbench 就会将这个编辑器的显示名前方加上一个“*”，表示这个编辑已经被修改，可以进行保存。正如下图的 Java 编辑器一样：



而当我们做完了保存操作后，把 `dirty` 属性设置为 `false`，即内容没有更改，然后发射出信号，Workbench 就会自动把“*”取消，并且保存按钮也被设置为 `enable`。

我们的 `CustomerSummryEditor` 类中，在执行完毕 `doSave` 后就开始这上述操作：

```

public void doSave(IProgressMonitor monitor) {
    customer.setSummry(text.getText());
    setDirty(false);
    firePropertyChange(PROP_DIRTY);
}

```

```
}
```

当然，这不是唯一的方法，这只是比较常见的方法，开发人员可以根据需要自己进行一些修改。

`createPartControl` 方法是创建编辑的 UI 部分的。我们在这里创建了一个 `Text` 对象，它用来显示 `Customer` 的 `Summry` 属性的。`Text` 加入了一个监听器，每当被修改后就会发射出被修改的信号。

在 `init` 方法中，我们得到了一个 `EditorInput` 对象，这个对象就是我们刚才所创建的，我们把它携带的 `Customer` 对象获得后设置为编辑器的 `Customer` 对象，这样便于操作。这里需要注意：`editor` 的 `site` 和 `editorinput` 属性在打开编辑器的时候是不能为空的，所以我们在 `init` 的时候把这两个对象都设置给了 `CustomerSummryEditor` 对象。

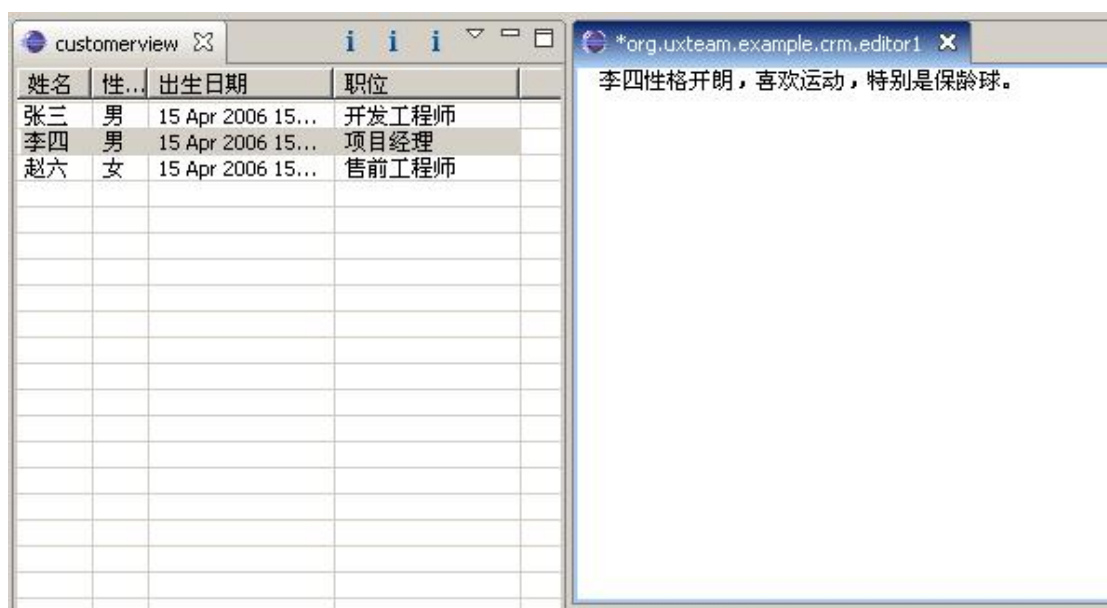
我们在 `CustomerViewer` 的 `AddAction run` 方法中加入以下代码：

```
public void run() {  
    Object customer =  
        ((StructuredSelection)viewer.getSelection())  
        .getFirstElement();  
    TextEditorInput in =  
        new TextEditorInput((Customer)customer);  
    try {  
  
        getSite().getWorkbenchWindow().getActivePage()  
            .openEditor(  
                in, "org.uxteam.example.crm.editor1");  
    } catch (PartInitException e) {  
        // TODO 自动生成 catch 块  
        e.printStackTrace();  
    }  
}
```

}

首先，我们获得了当前选中的 `Customer` 对象，然后创建好一个 `TextEditorInput`，再通过 `ActiveWindow` 对象的 `openEditor` 方法打开编辑器。`openEditor` 方法中具备两个参数，一个就是 `TextEditorInput`，另一个是 `Editor` 对应的 `ID`，这个 `ID` 是在 `plugin.xml` 文件中设定好的。

下面是运行结果:



5.6 扩展 EditorActions

每一个 Editor 都可以对应自己的一组 Action 按钮，这和我们即将讲到的 View 的 Action 一样的，当我们打开一个编辑器的时候，对应的 Editor 的 Action 就会出现在 Workbench 的 ActionBar 上，方便用户操作。

插件使用 `editorActions` 扩展点来为自述文件编辑器添加的菜单添加附加操作。现在, `plugin.xml` 中的定义应该看起来相当熟悉。

```

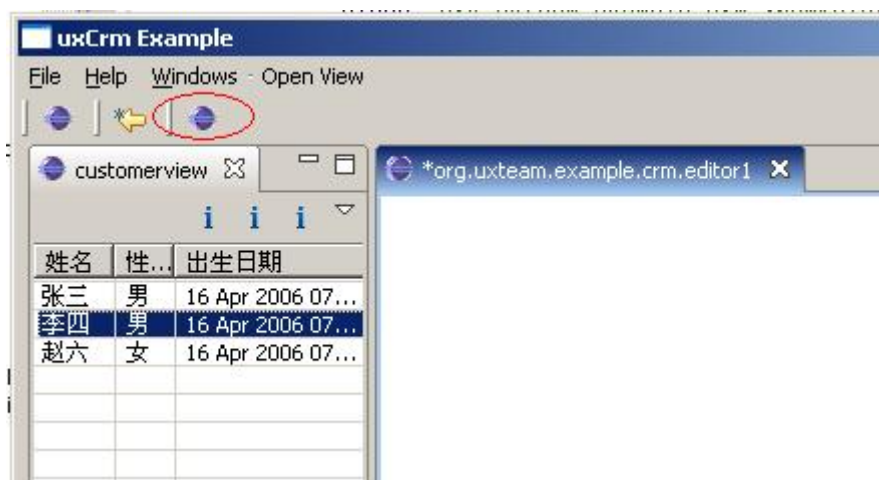
<extension
    point="org.eclipse.ui.editorActions">
    <editorContribution
        id="org.uxteam.example.crm.editorContribution1"
        targetID="org.uxteam.example.crm.editor1">
        <menu
            id="org.uxteam.example.crm.menu1"
            label="Actions"
            path="addtions">
            <separator name="TestActions"/>
        </menu>
        <action
            class="org.uxteam.example.crm.EditorActionDelegate1"
            icon="icons/sample.gif"
            id="org.uxteam.example.crm.editoraction1"
            label="TestAction"
            menubarPath="org.uxteam.example.crm.menu1/TestActions"
            style="push"
            toolbarPath="TestActions"/>
        </editorContribution>
    </extension>

```

与视图操作类似，扩展必须指定它正对其添加操作的编辑器的 `targetID`。该操作本身与视图操作（标识、标签、图标和 `toolbarPath...`）非常相似，但指定类必须实现 `IEditorActionDelegate`。

注意，此标记中未指定菜单栏路径。因此，当编辑器活动时，该操作将出现在工作台工具栏中，而不是出现在工作台菜单栏中。

运行一下，当编辑器活动时，我们会看到编辑器操作出现在由编辑器本身添加的操作旁边的工具栏上。



除了这种方法以外我们还可以指定一下 Editor 扩展点中的 EditorActionContributor 类，这也可以达到上述效果。我们回到 plugin.xml 中 Editor 的扩展点：

```
<editor
  class="org.uxteam.example.crm.editors.CustomerSummryEditor"
  contributorClass="CustomerEditorActionBarContributor"
  default="false"
  icon="icons/sample.gif"
  id="org.uxteam.example.crm.editor1"
  name="org.uxteam.example.crm.editor1"/>
```

我们把 contributorClass 指定为新生成的 CustomerEditorActionBarContributor：

```
public class CustomerEditorActionBarContributor extends
    EditorActionBarContributor {
```

```

        private Action testAction;

        public CustomerEditorActionBarContributor() {
            super();

            testAction = new Action(){

                public void run(){

                    MessageDialog.openInformation(null,"Information",

                        "Customer 编辑器的 TestAction");

                }

            };

            testAction.setText("TestAction2");

```

```

                testAction.setToolTipText("This is a test Action for
customer Editor");

```

```

                testAction.setImageDescriptor(
CrmPlugin.getImageDescriptor("icons/sample.gif"));
            }

            public void contributeToMenu(IMenuManager manager) {

                IMenuManager menu = new MenuManager("Customer Editor
&Menu");

                manager.prependToGroup(
IWorkbenchActionConstants.MB_ADDITIONS, menu);

                menu.add(testAction);

            }

            public void contributeToToolBar(IToolBarManager manager) {

                manager.add(new Separator());

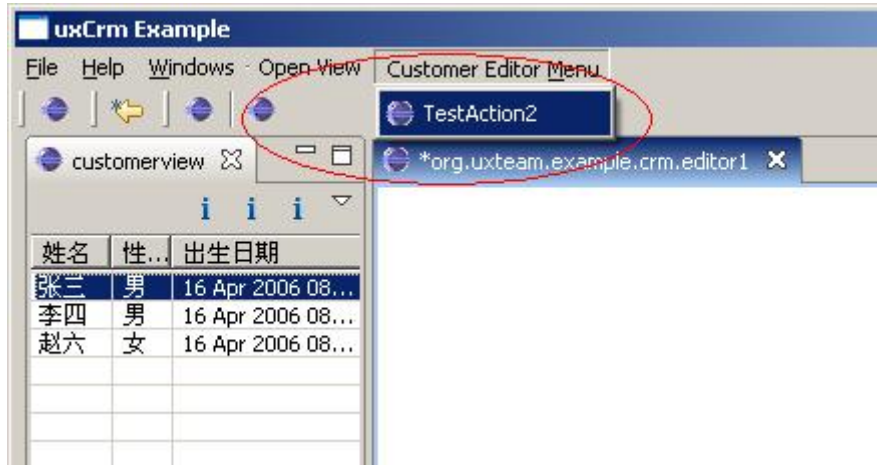
                manager.add(testAction);

            }

        }

```

我们在 Editor 中的 ActionBar 和 Menu 中分别注册了 testAction，运行就会发现，比刚才的运行界面多出了一个 Action：



这两种为 Editor 增加 Action 的方法都很实用，开发者根据需要选择不同的方式即可，只是在扩展点上直接增加的方式比较简单也比较容易，而且扩展性也很强。

5.7 View 的 Action 和 PopMenu

我们的 CustomerView 是一个仅仅有一个 TableView 的 ViewPart。其实在 ViewPart 上，我们还可以为它添加一些 toolBar 和 menuBar，并且可以创建一个 PopMenu。

我们假设 CustomerView 上将会有这些操作：

- 新增一个 Customer
- 删除一个 Customer
- 修改一个 Customer

我们将这三个操作都放置在 toolBar 和 menuBar 上，并且让制作一个 PopMenu，让它显示出这三个操作来。

首先我们要定义三个 Action，这里的 Action 其实就是一些操作，它并不仅仅代表一个 toolbar 上的按钮或者是 menuBar 上的选项，我们会把这三个 Action 注册到 toolbar 和 menuBar 上。

定义这三个 Action：

```
private Action deleteAction;

private Action addAction;

private Action modifyAction;
```

我们在 `createActions` 方法中实例化这三个 `Action`，并用弹出对话框来测试他们的动作。

```
private void showMessage(String message) {
    MessageDialog.openInformation(
        viewer.getControl().getShell(),
        "Sample View",
        message);
}

private void makeActions() {
    deleteAction = new Action() {
        public void run() {
            showMessage("deleteAction executed");
        }
    };

    deleteAction.setText("Delete");
    deleteAction.setToolTipText("Delete tooltip");
    deleteAction.setImageDescriptor(
        PlatformUI.getWorkbench().getSharedImages().
        getImageDescriptor(ISharedImages.IMG_OBJ_INFO_TSK));

    addAction = new Action() {
        public void run() {
            showMessage("addAction executed");
        }
    };
}
```

```

        }

};

addAction.setText("Add");

addAction.setToolTipText("Add tooltip");

addAction.setImageDescriptor(
PlatformUI.getWorkbench().getSharedImages().
getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));

modifyAction = new Action() {

    public void run() {

        showMessage("modifyAction executed");

    }

};

modifyAction.setText("Modify");

modifyAction.setToolTipText("Modify tooltip");

modifyAction.setImageDescriptor(
PlatformUI.getWorkbench().getSharedImages().
getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));

}

```

创建好以上的三个 Action 后，我们还需要将他们分别注册到 toolBar、MenuBar、PopupMenu 上。

先看看注册到 ToolBar 和 ActionBar 上的代码：

```

private void contributeToActionBars() {

    IActionBars bars = getViewSite().getActionBars();

    fillLocalPullDown(bars.getMenuManager());

    fillLocalToolBar(bars.getToolBarManager());

}

```

```

private void fillLocalPullDown(IMenuManager manager) {
    manager.add(deleteAction);
    manager.add(new Separator());
    manager.add(addAction);
    manager.add(modifyAction);
}

private void fillLocalToolBar(IToolBarManager manager) {
    manager.add(deleteAction);
    manager.add(addAction);
    manager.add(modifyAction);
}

```

我们先好获得改 ViewPart 上的 IActionBars 的对象句柄，改句柄是在创建好 ViewPart 后就有的，然后通过 fillLocalPullDown 和 fillLocalToolBar，分别把三个 Action 注册到它上面去，下面是运行结果：



我们现在还需要为 CustomerView 增加一个 PopMenu:

```

private void hookContextMenu() {
    MenuManager menuMgr =
    new MenuManager("#PopupMenu");
}

```

```

        Menu menu =
        menuMgr.createContextMenu(viewer.getControl());

        fillContextMenu(menuMgr);

        viewer.getControl().setMenu(menu);

        getSite().registerContextMenu(menuMgr, viewer);
    }

    private void fillContextMenu(IMenuManager manager) {

        manager.add(deleteAction);

        manager.add(addAction);

        manager.add(modifyAction);

        // Other plug-ins can contribute there actions here
        manager.add(
            new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
    }

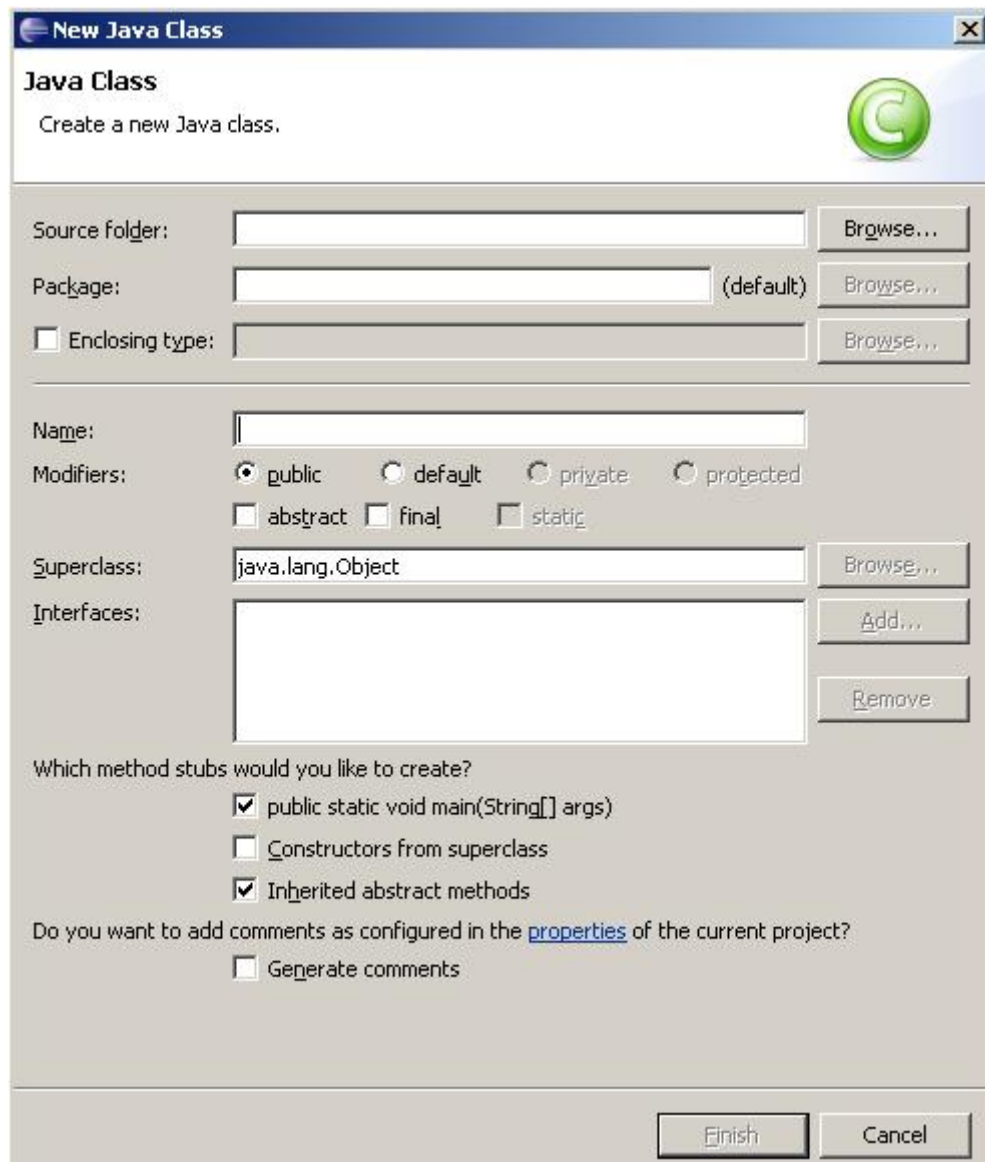
```

当我们在 CustomerView 上点击右键的时候，就会出现一个 PopMenu，并且上面的按钮即为我们所定义的三个按钮：



6. 扩展一个向导页

我们一般在新建或者是修改的时候常常使用到向导页，向导页能够引导用户，按照步骤完成一些操作，比如我们新建一个 Java 类，Eclipse 的 Java 类创建向导能够让我们根据一定步骤来创建这个 Java 类：



我们在新建一个 Customer 的时候也项通过向导来引导用户创建一个新的 Customer，这样一来我们就能够规范用户的一些输入，并且透明化一些繁琐的操作，这给用户的体验是完全不一样的。

首先考虑一下新建一个 Customer 需要哪些必要信息：

➤ **Customer 姓名**

- **Customer** 性别
- **Customer** 的职位
- **Customer** 的出生日期

依照上面的信息，我们可以设计出输入 **Customer** 信息的控件：



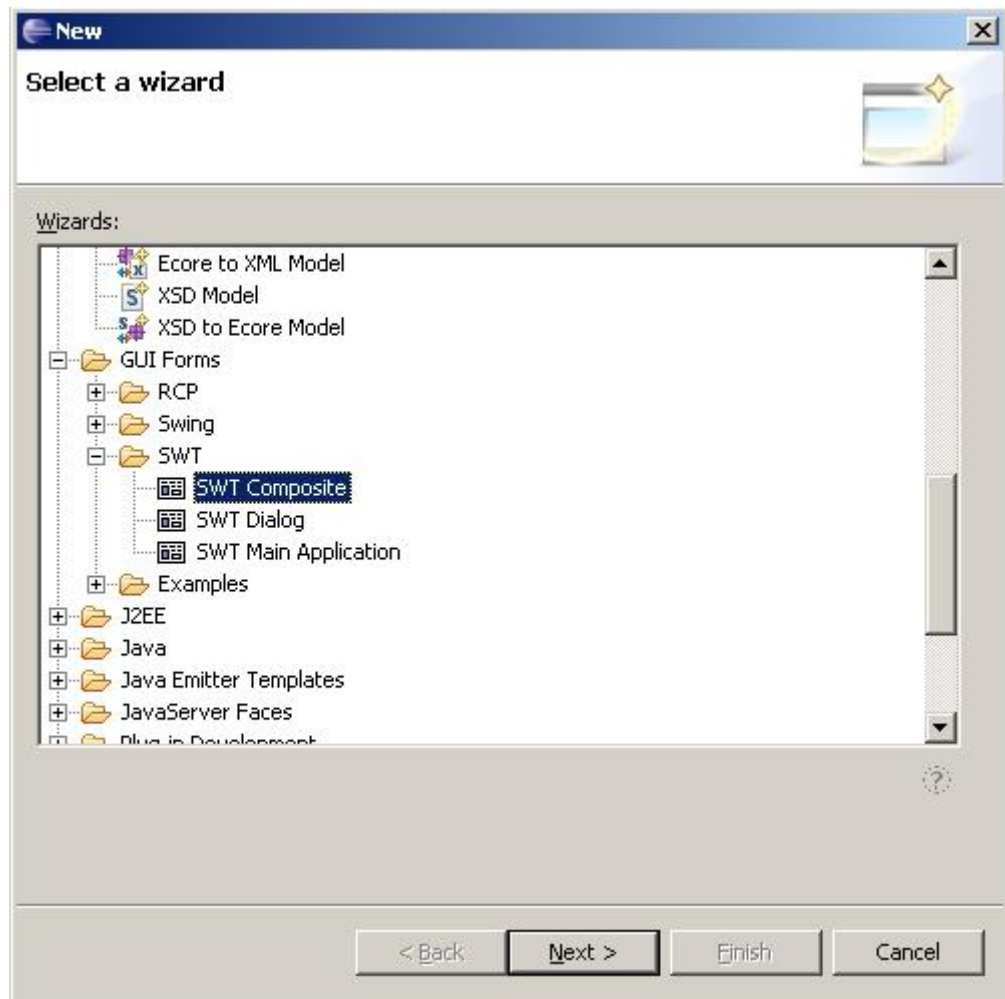
注意，这只是一个简单的控件，我们并没有把它加载到向导中。

我们通过这个控件输入的必要信息，然后在向导结束后利用这些信息创建出我们的 **Customer** 对象，并把这个对象加载到预先创建好的根容器中（这里涉及到了 **EMF** 的一些知识，不过由于我们只讲解 **RCP** 的应用，所以 **EMF** 的知识可以略过），那在 **CustomerView** 中就会显示出新建的这个 **Customer** 对象来。

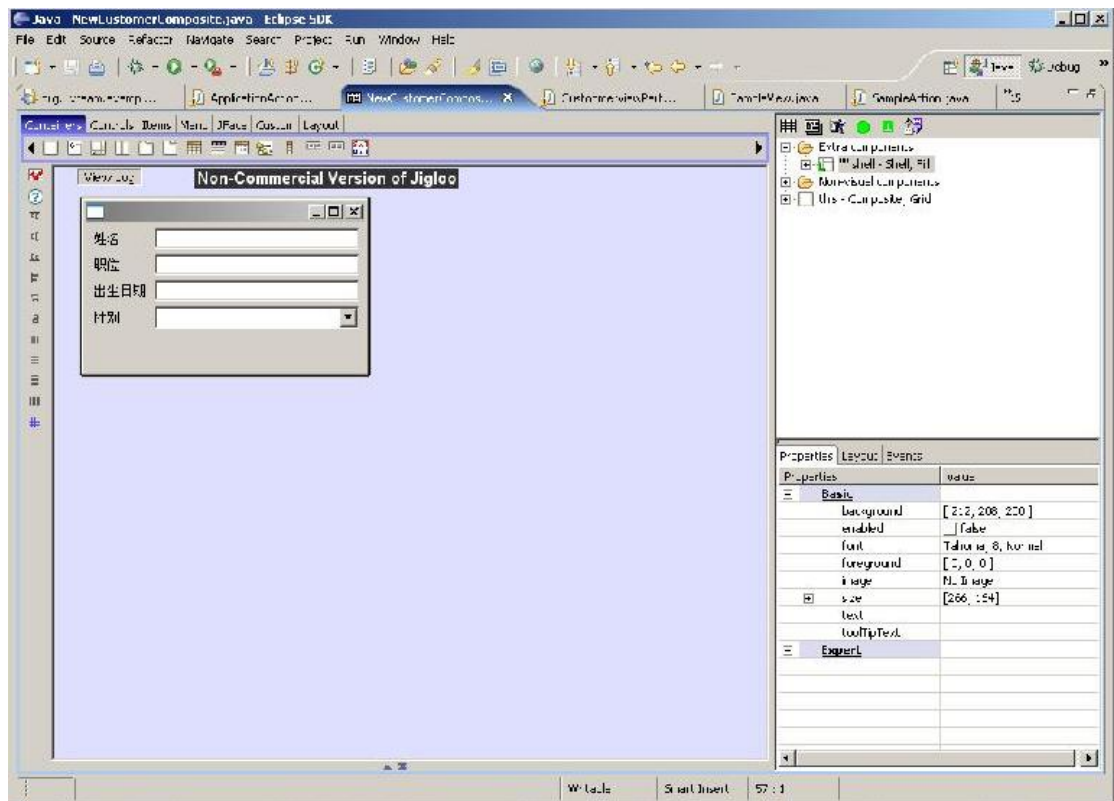
6.1 生成新建 **Customer** 对象的控件(Composite)

我们使用了一些辅助工具在创建我们的 **Composite**，这个工具叫做 **jigloo**，是免费的，我们可以在 <http://www.cloudgarden.com> 获得该工具。在以后的开发中，我们常常会使用到这个工具。

先利用工具创建一个新的 **Composite**：



然后我们生成了一个名为 NewCustomerComposite 的 Composite，用工具打开后进行图形化的编辑：



我们先看看这个输入 Customer 信息控件的代码：

```
private void initGUI() {
    try {
        GridLayout thisLayout = new GridLayout();
        this.setLayout(thisLayout);
        thisLayout.numColumns = 2;
        this.setSize(258, 137);
        {
            cLabel1 = new CLabel(this, SWT.NONE);
            cLabel1.setText("\u59d3\u540d");
        }
        {
            text1 = new Text(this, SWT.BORDER);
            GridData text1LData = new GridData();
```

```

        text1LData.heightHint = 13;

        text1LData.grabExcessHorizontalSpace = true;
        text1LData.horizontalAlignment = GridData.FILL;
        text1.setLayoutData(text1LData);
    }

    {
        cLabel2 = new CLabel(this, SWT.NONE);
        cLabel2.setText("\u804c\u4f4d");
    }

    {
        text2 = new Text(this, SWT.BORDER);
        GridData text2LData = new GridData();
        text2LData.grabExcessHorizontalSpace = true;
        text2.setLayoutData(text2LData);
        text2LData.horizontalAlignment = GridData.FILL;
        text2LData.grabExcessHorizontalSpace = true;
    }

    {
        cLabel3 = new CLabel(this, SWT.NONE);
        cLabel3.setText("\u51fa\u751f\u65e5\u671f");
    }

    {
        text3 = new Text(this, SWT.BORDER);
        GridData text3LData = new GridData();
        text3LData.grabExcessHorizontalSpace = true;
        text3.setLayoutData(text3LData);
        text3LData.horizontalAlignment = GridData.FILL;
        text3LData.grabExcessHorizontalSpace = true;
    }

    {

```

```

        cLabel4 = new CLabel(this, SWT.NONE);
        cLabel4.setText("\u6027\u522b");
    }
    {
        combo1 = new Combo(this, SWT.NONE);
        GridData combo1LData = new GridData();
        combo1LData.grabExcessHorizontalSpace = true;
        combo1.setLayoutData(combo1LData);
        combo1LData.horizontalAlignment = GridData.FILL;
        combo1LData.grabExcessHorizontalSpace = true;
        combo1.add("男");
        combo1.add("女");
    }
    this.layout();
} catch (Exception e) {
    e.printStackTrace();
}
}

```

新生成的控件可以获得一些必要信息，我们把它的这些能得到信息的控件暴露出来，供创建新 Customer 时取得信息使用：

```

public Date getBorn() {
    return new Date(text3.getText());
}

public void setBorn(Date born) {
    this.born = born;
}

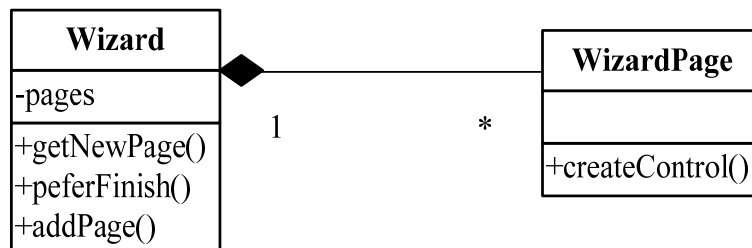
```

```
public String getName() {  
    return text1.getText();  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public String getSex() {  
    return combol.getText();  
}  
  
public void setSex(String sex) {  
    this.sex = sex;  
}  
  
public String getPost() {  
    return text2.getText();  
}  
  
public void setPost(String post) {  
    this.post = post;  
}
```

通过以上代码我们可以轻松地从中获得 Customer 的一些必要信息。我们现在需要的不是整个 Composite 类，而是它的部分代码，所以在后面的开发中，我们会将代码 Copy 到创建向导页的类中。

6.2 创建向导页和向导

向导其实只是一个载体，真正显示出来的部分被成为向导页。向导通过加载向导页，在显示的时候通过向导页的加载顺序，逐一加载到当前的页面中，通过向导的 New 按钮，一个一个显示出来。下面是向导页和向导直接的关系图：



我们先生新建一个 Wizard 类，命名为 NewCustomerWizard:

```
public class NewCustomerWizard extends Wizard implements
INewWizard {
    NewCustomerWizardPage page;

    public NewCustomerWizard() {
        super();
    }

    /* (non-Javadoc)
     * @see org.eclipse.jface.wizard.Wizard#performFinish()
     */
    public boolean performFinish() {
        return true;
    }

    public void init(IWorkbench workbench, IStructuredSelection
selection) {
```

```
    }  
}
```

我们看看它的方法含义：

➤ **performFinish**

这是当点击 Wizard 的 Finish 按钮时执行的代码方法

➤ **init**

在创建 Wizard 后初始化该向导的时候调用的代码方法

Wizard 是通过 addPage 方法加向导页的。

我们再新建一个向导页 (WizardPage)：

```
public class NewCustomerWizardPage extends WizardPage {  
  
    protected NewCustomerWizardPage(String pageName, String  
title) {  
        super(pageName, title,  
CrmPlugin.getImageDescriptor("icons/sample.gif"));  
    }  
  
    public void createControl(Composite parent) {  
        this.setControl(parent);  
    }  
  
}
```

WizardPage 的构造函数中可以制定该向导页的名称以及对应的显示的图标，在 createControl 方法中是创建该向导页控件的地方，我们只要把刚才新建的 Composite

代码加入到其中即可。

现在我们有了 Wizard 和 WizardPage 后, 先在 Wizard 的构造函数中实例化这个 WizardPage, 并把它加入到 Wizard 的 Pages 当中:

```
public NewCustomerWizard() {  
    super();  
    page = new NewCustomerWizardPage("New Customer", "New  
Customer Wizard Page");  
    this.addPage(page);  
}
```

然后我们把刚才生成的 Composite 的代码加入到 WizardPage 的 createControl 当中。

在获得 Customer 的信息后, 我们会在 performFinish 代码中加入创建用户的代码:

```
public boolean performFinish() {  
    Customer c = CoreFactory.eINSTANCE.createCustomer();  
    c.setName(page.getNameText().getText());  
    c.setBornDate(new  
Date(page.getBornDateText().getText()));  
    c.setSex(page.getSexCombo().getText());  
    c.setPost(page.getZhiweiText().getText());  
    CrmPlugin.getDefault().getRoot().getCustomer().add(c);  
    return true;  
}
```

通过上面这些步骤后, 我们可以得到一个简单的新建 Customer 向导。

6.3 将向导加入到新建目录中

建好向导后，我们需要显示它，在 Eclipse 中我们可以通过 File->New 去选择要显示的向导，但是我们的 RCP 中还没有这个选项，我们可以通过 `org.eclipse.jface.wizard.WizardDialog` 类将我们的向导显示出来：

```
WizardDialog dialog =  
    new WizardDialog(this.getShell(),new NewCustomerWizard());  
dialog.open();
```

但是这样做的话就会引起创建 `CustomerView` 类似的情况，所以我们还是需要将这个 Wizard 加入到 Eclipse 的 File-New 目录中去。

首先我们创建一个 `org.eclipse.ui.wizard` 扩展点：

```
<extension  
    point="org.eclipse.ui.newWizards">  
    <wizard  
        category="org.uxteam.example.crm.customer_category"  
        class="org.uxteam.example.crm.wizard.NewCustomerWizard"  
        icon="icons/sample.gif"  
        id="org.uxteam.example.crm.newcustomerwizard"  
        name="New Customer" />  
    <category  
        id="org.uxteam.example.crm.customer_category"  
        name="Customers" />  
    </extension>
```


➤ **wizard**

这个元素表示我们的 wizard，其中包括对应的 wizard 类、图标、显示名、所在目录等信息

➤ **category**

wizard 所在的目录。

现在我们在 ApplicationActionBarAdvisor 类中将 File->New 这样的目录加入到其中。

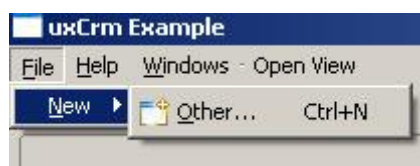
首先我们新建一个名为 newFileAction 的 IWorkbenchAction:

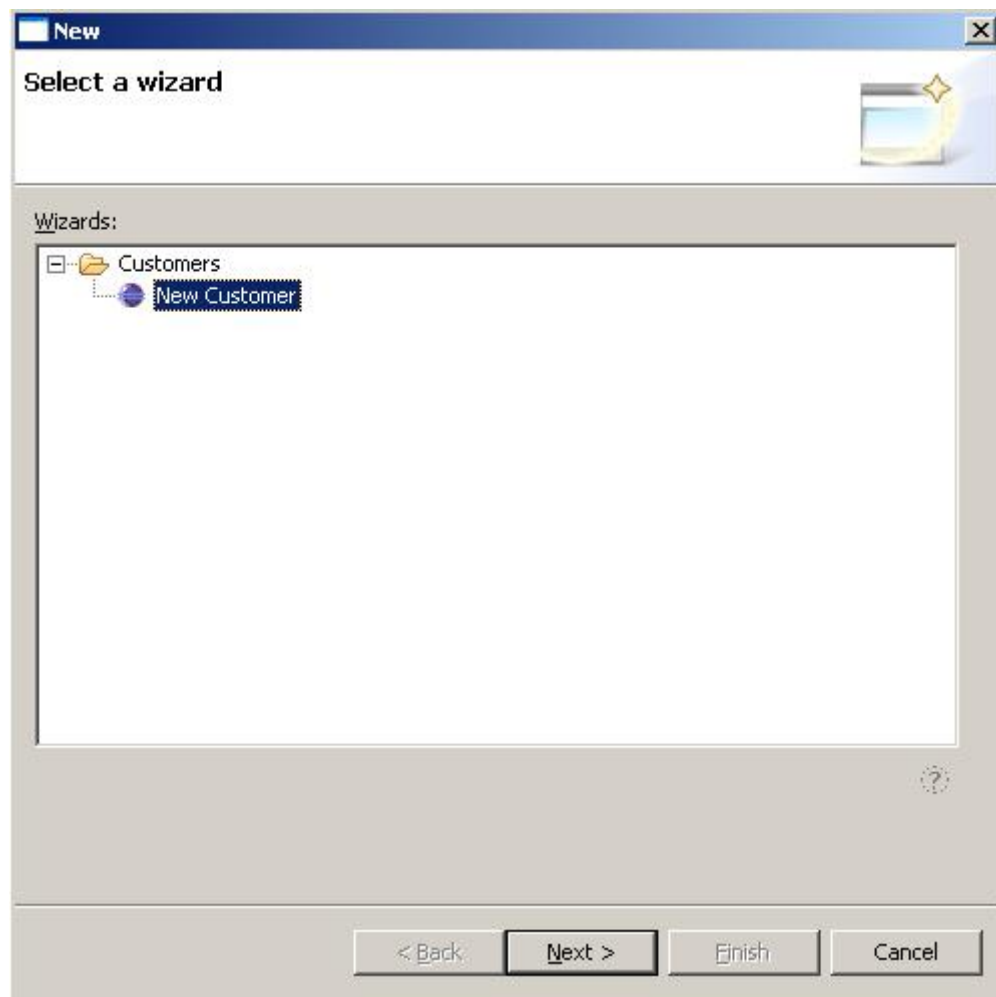
```
newFileAction = ActionFactory.NEW.create(window);  
register(newFileAction);
```

然后增加菜单项:

```
MenuManager fileMenu =  
new MenuManager("&File", IWorkbenchActionConstants.M_FILE);  
menuBar.add(fileMenu);  
  
MenuManager newMenu =  
new MenuManager("&New", ActionFactory.NEW.getId());  
newMenu.add(newFileAction);  
fileMenu.add(newMenu);
```

这样我们就能如同在 Eclipse IDE 那样，获得一个 Wizard 的对话框了:





选择 New Customer 选项后点击 Next，即可得到我们刚才创建的 Wizard：

The image shows a Windows-style dialog box titled "New Customer Wizard Page". It has a standard title bar with minimize, maximize, and close buttons. The main area contains four input fields with labels in Chinese: "姓名" (Name), "职位" (Position), "出生日期" (Date of Birth), and "性别" (Gender). The "出生日期" field is currently selected. At the bottom of the dialog, there are four buttons: "< Back" (disabled), "Next >" (disabled), "Finish" (active), and "Cancel".

6.4 Wizard 的提示信息

我们在输入 Customer 信息的时候，有时候会有一些信息对于新建 Customer 时是错误的，比如出生日期，在我们没有使用一个特别的控件时，只能时手动输入一个字符串，并且字符串的格式为“YYYY/MM/DD”，但是如果用户并不这样输入的话，那新建的 Customer 信息就会出现不可预知的错误了，所以我们需要规范用户的操作。

我们可以利用 Wizard 提供的一些方法，来捕获错误，并且显示出来，让用户在输入错误时无法点击 Finish。

我们就“出生日期”这一项来进行讨论。

首先我们创建一个名为 `upDataWizardErrorMessage` 的方法。该方法判断输入的出生日期字符串是否符合规定，如果不符合将会调用 `setErrorMessage` 去提示用户，并且如果该错误信息不为空，即有错误发生时候，我们将认定该 Page 没有完成，Wizard 就会

去确定 Next 或者 Finish 按钮是否可用。

```
protected void upDataWizardErrorMessage(){
    String date = bornDateText.getText();
    String errorMes = null;
    if(date == null){
        errorMes = "出生日期不能为空";
    }
    try{
        new Date(date);
    }catch(Exception e){
        errorMes = "出生日期必须按照 YYYY/MM/DD 的格式进行输入";
    }
    setErrorMessage(errorMes);
    this.setPageComplete(errorMes == null);
}
```

首先，打开 NewCustomerWizarPage 类，在该控件（bornText）上加入一个监听器：

```
bornDateText.addModifyListener(new ModifyListener(){
    public void modifyText(ModifyEvent e) {
        upDataWizardErrorMessage();
    }
});
```

这样，每当用户输入日期的时候监听器就会去调用 upDataWizardErrorMessage 方法去更新 wizard 的错误提示信息。

我们运行后可以得到以下结果：

我们输入的出生日期不符合规范，wizard 就会在上方显示出提示错误信息，并且将 Finish 按钮变为了灰色不可用。

直到我们输入了正确格式的出生日期后，Finish 才将能够使用。

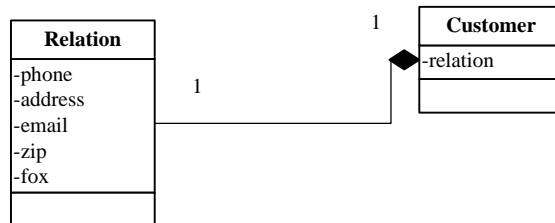
在我们创建完了一个 Customer 后，CustomerViewPart 可以监听到该 Customer 的增加的，于是就开始刷新表格。这些操作都和 EMF 有关联，不过我们这里不对 EMF 做讨论。

7. 创建“联系方式”ViewPart

我们的需求中针对客户信息是有一项联系方式的。联系方式是一个聚合在 Customer 类中的一个类，这个类具有一些常见的属性，比如电话、传真、邮箱、地址等，我们利用这个类来描述 Customer 的联系方式。

7.1 类图结构

正如上面所说的，Relation 类是聚合在 Customer 里的一个类，它专门负责维护客户的一些联系方式信息，类图如下：



我们要显示 Relation 的这些熟悉需要做得就是制作一个 ViewPart 来显示这些信息。

7.2 创建 RelationViewPart

我们利用工具构建这个 ViewPart。

首先，Relation 需要显示的信息有以下五个：

- 电话
- 地址
- 电子邮件
- 传真
- 邮编

那我们制作传来的 ViewPart 如下图所示：



我们采用 CLabel 控件来显示它 Relation 的属性信息。当我们创建完这个 ViewPart 后，就利用 ViewPart 的接口方式向它传递 Relation 对象，这样 RelationViewPart 就能够更新这个 CLabel 以达到显示的目的。

UI 代码：

```
public void createPartControl(Composite parent) {  
    {  
        GridLayout parentLayout = new GridLayout();  
        parentLayout.numColumns = 2;  
        parent.setLayout(parentLayout);  
        parent.setSize(416, 270);  
        parent.setBackground(  
SWTResourceManager.getColor(255, 255, 255));  
        {  
            cLabel1 = new CLabel(parent, SWT.NONE);  
            GridLayout cLabel1Layout = new GridLayout();
```

```

        cLabel1Layout.makeColumnsEqualWidth = true;

        cLabel1Layout.numColumns = 2;

        cLabel1.setLayout(cLabel1Layout);

        cLabel1.setText("电话: ");

        cLabel1.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        phoneLabel = new CLabel(parent, SWT.NONE);

        GridData cLabel2LData = new GridData();

        cLabel2LData.grabExcessHorizontalSpace = true;

        cLabel2LData.horizontalAlignment = GridData.FILL;

        phoneLabel.setLayoutData(cLabel2LData);

        phoneLabel.setBackground(SWTResourceManager.getColor(255, 255,
255));
    }
    {
        cLabel3 = new CLabel(parent, SWT.NONE);

        cLabel3.setText("地址: ");

        cLabel3.setBackground(SWTResourceManager.getColor(255, 255,
255));
    }
    {
        addressLabel = new CLabel(parent, SWT.NONE);

        GridData cLabel4LData = new GridData();

        cLabel4LData.grabExcessHorizontalSpace = true;

        cLabel4LData.horizontalAlignment = GridData.FILL;

        addressLabel.setLayoutData(cLabel4LData);
    }

```



```

        addressLabel.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        cLabel5 = new CLabel(parent, SWT.NONE);
        cLabel5.setText("电子邮箱: ");
        cLabel5.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        mailLabel = new CLabel(parent, SWT.NONE);
        GridData cLabel6LData = new GridData();
        cLabel6LData.grabExcessHorizontalSpace = true;
        cLabel6LData.horizontalAlignment = GridData.FILL;
        mailLabel.setLayoutData(cLabel6LData);
        mailLabel.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        cLabel7 = new CLabel(parent, SWT.NONE);
        cLabel7.setText("邮编: ");
        cLabel7.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        zipLabel = new CLabel(parent, SWT.NONE);
        GridData cLabel8LData = new GridData();
        cLabel8LData.grabExcessHorizontalSpace = true;
        cLabel8LData.horizontalAlignment = GridData.FILL;
        zipLabel.setLayoutData(cLabel8LData);
    }

```

```

        zipLabel.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        cLabel9 = new CLabel(parent, SWT.NONE);
        cLabel9.setText("传真: ");
        cLabel9.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
    {
        foxLabel = new CLabel(parent, SWT.NONE);
        GridData cLabel10LData = new GridData();
        cLabel10LData.grabExcessHorizontalSpace = true;
        cLabel10LData.horizontalAlignment = GridData.FILL;
        foxLabel.setLayoutData(cLabel10LData);
        foxLabel.setBackground(
SWTResourceManager.getColor(255, 255, 255));
    }
}
}

```

当我们传入 Relation 对象后, ViewPart 就更新这些 Label:

```

public void setRelation(Relation relation){
    if(relation != this.relation){
        this.phoneLabel.setText(relation.getPhone());
        this.mailLabel.setText(relation.getEmail());
        this.addressLabel.setText(relation.getAddress());
        this.zipLabel.setText(relation.getZip());
        this.foxLabel.setText(relation.getFax());
    }
}

```

```
}
```

7.3 为 CustomerViewPart 的 PopMenu 增加 Action

我们需要显示 Customer 的 Relation，那当我们打开 CustomerView 的时候就需要有个机制，让它能显示出来。我们这样规定：当我们选中 CustomerViewPart 上的某一个 Customer 对象的时候，点击右键，在弹出菜单项中有一项“Show Detail”子项，然后在这个子项下面有一个“打开关系视图”的按钮。

由于我们在前面的章节已经说了怎么为一个 ViewPart 增加一个 PopMenu，这里不再复述。现在我们需要为这个 PopMenu 增加一些按钮。

普通的方式就是在原有的代码基础上为这个 PopMenu 增加一个 Action:

```
showRelationAction = new Action() {  
    public void run() {  
        showMessage("addAction executed");  
    }  
};  
  
showRelationAction.setText("Add");  
showRelationAction.setToolTipText("Add tooltip");  
.....  
  
manager.add(showRelationAction);
```

但是这样的话对代码修改的话会对本身 CustomerViewPart 污染，而且如果当弹出的 Action 增加的时候，代码也会随之增加。

另一个方法就是对 Plugin.xml 进行修改。

我们在 Plugin.xml 上增加一个 org.eclipse.ui.popMenus 扩展点:

```
<extension  
    point="org.eclipse.ui.popupMenus">  
    <objectContribution  
        adaptable="false"
```

```

        id="org.uxteam.example.crm.objectContribution1"

        objectClass="org.eclipse.emf.ecore.EObject">

        <menu

        id="org.uxteam.example.crm.menu2"

        label="Show Details"

        path="addtions">

        <groupMarker name="new" />

        </menu>

        <action

        class="org.uxteam.example.crm.ShowRelationAction"

        id="org.uxteam.example.crm.action2"

        label="查看联系方式"

        menubarPath="org.uxteam.example.crm.menu2/new" />

        </objectContribution>

    </extension>

```

分析一下这个 XML 片断：

➤ **ObjectContribution**

这是表示该弹出菜单将会在那个对象上进行显示。

当我们选中一个对象的时候（比如 Customer 对象），Workbench 会将这个 Selection 记录下来，然后我们点击右键的时候就会进行分析，如果该 Selection 和 objectContribution 对应的 objectClass 相同或者是它的子类，那么就会显示出 XML 指定的 menu 和 action。

所以 我们 这里 把 objectClass 的值 设置 为了 org.eclipse.emf.ecore.EObject，这个类是 Customer 类的父类。

➤ **Menu**

显示的 menu 项。里面的 groupMark 表示的是显示的菜单项的子项集。

➤ **Action**

对应的 Action。Action 有一个对应的 Action 类，这个类需要实现 IObjectActionDelegate 接口，并且要指定该 Action 所在的路径 menubarPath。

看看这个类的实现代码:

```
public class ShowRelationAction implements IObjectActionDelegate
{
    ISelection selection;
    IWorkbenchPart window;
    .....

    public void run(IAction action) {
        if (selection != null && window != null) {
            if (selection instanceof IStructuredSelection) {
                Customer c = (Customer)
                ((IStructuredSelection) selection)
                                .getFirstElement();

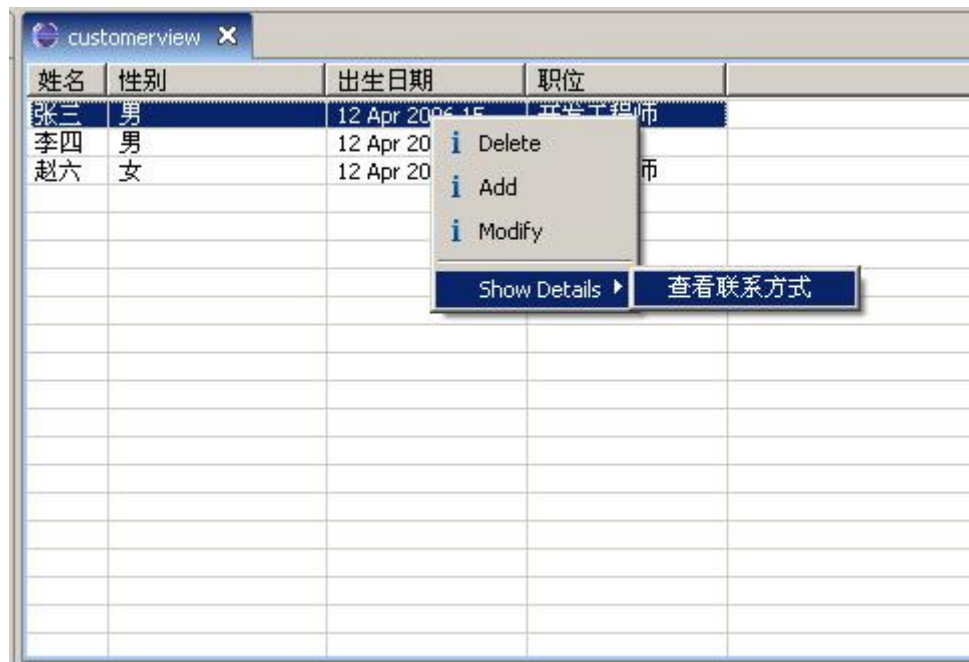
                try {
                    RelationViewPart view =
                    (RelationViewPart) window.getSite()
                                .getWorkbenchWindow().getActivePage().showView(
                                "org.uxteam.example.crm.relationview");

                    view.setRelation(c.getRelation());
                } catch (PartInitException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    .....
}
```

在 Action 的 Run 方法中,我们将 RelationViewPart 实现出来,并且将 Customer 的 Relation 对象传递给它,这样在显示出这个 RelationViewPart 后, Label 上就会

把 Customer 的 Relation 的所有属性显示出来。

下面是运行结果：



RelationViewPart 的显示效果：



8. 深入 JFace

8.1 JFace 介绍

在 Eclipse RCP 中，JFace 提供的 UI 组件相当丰富，比如对话框、进度条还有上面我们所提到的 TableViewer 以及 TreeViewer。这些组件都为我们构建一个友好的用户界面提供了支持。

我们已经见到工作台定义扩展点来为使插件向平台添加用户界面功能。许多这些扩展点，特别是向导扩展，是通过使用 `org.eclipse.jface.*` 包中的类来实现的。有什么区别？

JFace 是一个用户界面工具箱，它提供很难实现的、用于开发用户界面功能部件的 **helper** 类。**JFace** 在原始的窗口小部件系统的级别之上运行。它提供用于处理常见的用户界面编程任务的类：

- 查看器负责处理填充、排序、过滤和更新窗口小部件等最辛苦的工作。
- 操作和添加项介绍用于定义用户操作的语义，并指定在何处提供它们。
- 图像和字体注册表提供用于处理用户界面资源的常见模式。
- 对话框和向导定义用于构建与用户进行复杂交互的框架。

JFace 使您可以专注于实现特定插件的功能，而不必花费精力来处理底层窗口小部件系统或者解决几乎在任何用户界面应用程序中都很常见的问题。

我们将以 CRM 系统的后续开发为例，讲解这些常用的 UI 组件。

8.1.1 JFace 和 Workbench

何处是 **JFace** 结束而工作台开始的位置？有时候界线并不是这样明显。通常，**JFace** API（来自于包 `org.eclipse.jface.*`）独立于工作台扩展点和 API。可以想象，根本不必使用任何工作台代码就可以编写 **JFace** 程序。

工作台使用 **JFace**，但是又试图尽可能减少依赖项。例如，工作台部件模型（**IWorkbenchPart**）被设计为独立于 **JFace**。我们很早就知道可以直接使用 **SWT** 窗口小部件来实现视图和编辑器，而不必使用任何 **JFace** 类。工作台尽可能保持“**JFace** 中立”，允许程序员使用他们觉得有用的 **JFace** 的部件。实际上，在工作台的大多数实现中都使用了 **JFace**，并且在 API 定义中引用了 **JFace** 类型。（例如，**IMenuManager**、**IToolBarManager** 和 **IStatusLineManager** 的 **JFace** 接口显示为工作台 **IActionBar** 方法中的类型。）

8.1.1 JFace 和 SWT

SWT 和 JFace 之间的界线是很明显的。SWT 根本不依赖于任何 JFace 或平台代码。许多 SWT 示例说明可以如何构建独立的应用程序。

JFace 用来在 SWT 库顶部提供常见的应用程序用户界面功能。JFace 并不试图掩盖 SWT 或者替换它的功能。它提供一些类和接口，以处理与使用 SWT 来对动态用户界面编程相关联的许多常见任务。

通过了解查看器 (Viewer) 及它们与 SWT 窗口小部件之间的关系，就可以更清楚地了解 JFace 和 SWT 之间的关系。

8.2 对话框

8.2.1 标准对话框

我们在显示了用户信息后，对这些信息的操作中包括有一个“删除”操作，在删除的时候，我们会提示用户，是否执行该操作。因为在一般情况下，信息删除后是不能恢复的，所以在删除的时候需要比较慎重处理，有可能是用户点击错误，这样的情况下就需要我们去提示用户。

JFace 中提供了一系列的对话框，这些对话框都是由一个提示图标，加上对话框信息和一组按钮组成，不同的对话框具有不同的提示图标和按钮，而且根据点击按钮的不同，对话框对象的返回值也不同，比如问题对话框，点击“是”和“否”分别返回 true 和 false。

➤ Information Dialog

信息对话框。对话框显示一个信息图标，该对话框只有一个按钮，当点击后对话框就会关闭：



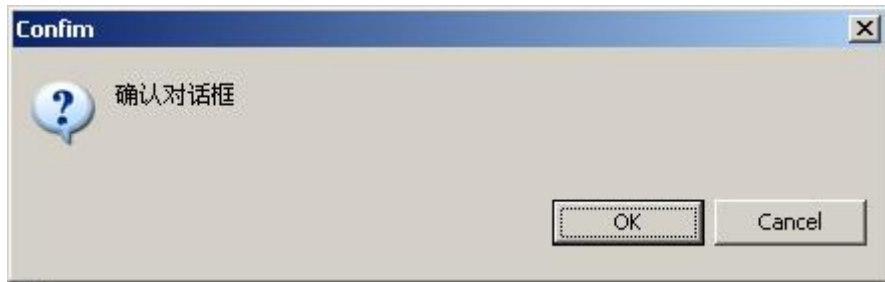
➤ **Error Dialog**

错误对话框。对话框显示一个错误图标，话框有一个按钮，当点击后对话框就会关闭：



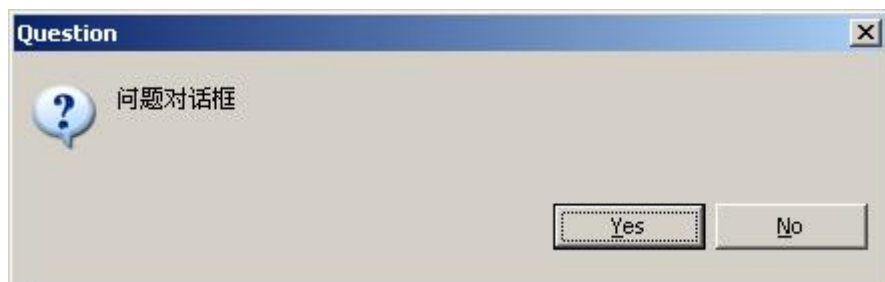
➤ **Confirm Dialog**

确认对话框。对话框显示一个提示图标，话框有两个按钮，点击 OK 返回 true，点击 Cancel 返回 false，操作完毕后话框就会关闭：



➤ **Question Dialog**

问题对话框。该对话框显示一个问题图标，点击 Yes 后返回 true，点击 No 后返回 false，然后对话框关闭。



➤ **Warning Dialog**

警告对话框。警告对话框显示一个警告图标，话框有一个按钮，当点击后对话框就会关闭：



上面一组对话框，都是由一个 `MessageDialog` 类的静态方法生成的，分别是：

`openInformation`, `openError`, `openConfirm`, `openQuestion`, `openWarning`。

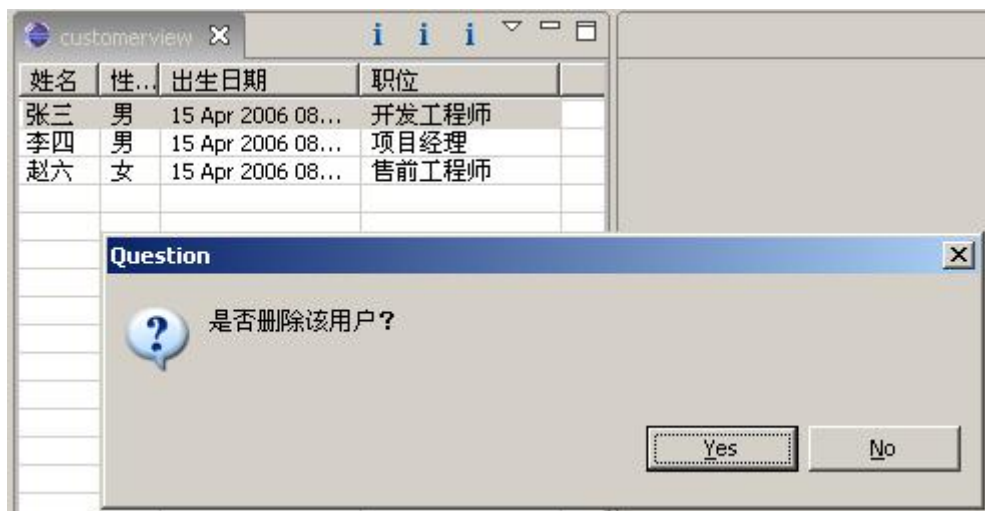
我们可以根据自己的需要来选择我们所需要的对话框。

回到我们的 CRM 上来。当我们点击 `deleteAction` 的时候，我们提示用户是否删除该 `Customer`，然后根据用户点击的按钮返回值来判断是否删除该用户。

我们在 `deleteAction` 的 `run` 方法中加入以上代码：

```
delateAction = new Action() {  
    public void run() {  
  
        boolean canDelete = MessageDialog.openQuestion(  
viewer.getControl().getShell(),  
            "Question", "是否删除该用户? ");  
        if(canDelete){  
            Object customer =  
((StructuredSelection)viewer.getSelection())  
.getFirstElement();  
  
CrmPlugin.getDefault().getRoot().getCustomer()  
.remove(customer);  
        }  
    }  
};
```

当我们点击删除按钮的时候，就会弹出对话框来提示用户是否删除该 `Customer`：



点击 Yes 后用户即被删除掉。

8.2.2 自定义对话框

JFace 除了提供了上述的那些 `StandardDialog` 外,主要是给出了一个 `Dialog` 类。这个类是供用户自定义对话框的基类。

`Dialog` 类是一个抽象类,我们需要继承它才行,在 `Dialog` 类中有一个方法:`createDialogArea`,这个方式是创建对话框主体的方法体,我们只要在子类中覆盖这个方法,即可创建出定制的对话框。

当然 `Dialog` 还有其他可以覆盖的方法,比如 `createButtonArea` 方法,这里我们就可以根据需要来创建我们想要的按钮。

下面是一个测试性的代码:

```
Dialog dialog = new Dialog(viewer.getControl().getShell()){
    protected Control createDialogArea(
        Composite parent) {
        Composite composite = (Composite)
            super.createDialogArea(parent);
        Label l = new Label(composite, SWT.NONE);
```

```

        l.setText("Test 对话框");

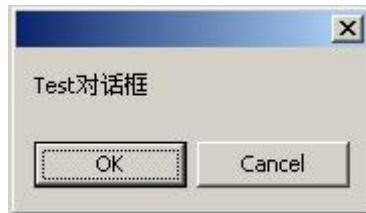
        return composite;

    }

};

dialog.open();

```



8.3 Viewer (查看器)

其实在我们上面的讲述中已经使用到了 Viewer，比如 CustomerViewPart 以及 RelationViewerPart 中，我们都使用到了 TableViewer，这里我们详细介绍这些 Viewer 以及他们的结构原理。

我们已经知道可以利用 SWT 窗口小部件来直接实现工作台用户界面添加项（例如，视图、编辑器、向导和对话框），为什么您还要使用查看器？

查看器允许您在创建窗口小部件时仍然可以使用模型对象。如果直接使用 SWT 窗口小部件，则需要将对象转换成 SWT 期望的字符串和图像。查看器充当 SWT 窗口小部件上的适配器，处理用于处理窗口小部件事件的常见代码，否则，将需要由您自己来实现。

我们首先在 CustomerViewPart 的客户信息的视图添加项中使用到了查看器。

```

public void createPartControl(Composite parent) {

    viewer = new TableViewer(parent, SWT.MULTI
|SWT.FULL_SELECTION);

    ...

}

```

注意：可以使用查看器来同时为工作台视图和编辑器提供实现。术语“查看器”并不意味着它们只能用于实现视图。例如，TextViewer 用于实现许多工作台和插件编辑器。

8.3 标准查看器 (Viewer)

JFace 为 SWT 中大多数重要的窗口小部件提供查看器。查看器通常用于列表、树、表和文本窗口小部件。

每个查看器都有相关联的 SWT 窗口小部件。可以通过在方便的查看器构造函数中提供父组合体来隐式地创建此窗口小部件，或者通过首先创建它，然后将它提供给它的构造函数中的查看器，从而显式地创建它。

8.3.1 面向列表的查看器 (TableViewer, ListView, TreeViewer)

列表、树和表共享许多从用户的观点来说较为常见的一些功能，例如，填充对象、选择、排序和过滤。

这些查看器保存域对象（称为元素）的列表，并将这些列表显示在其相应的 SWT 窗口小部件中。列表查看器知道如何从列表中的任何元素中获取文本标签。它从可以在查看器上设置的 `ILabelProvider` 中获取标签。列表查看器知道如何从窗口小部件回调映射回到查看器客户所知道的元素中。

使用平面 SWT 窗口小部件的客户需要在这样一个 SWT 级别上运行 — 在该级别中，项是字符串，而事件通常与字符串列表中的索引相关。查看器提供了更高级的语义。对于客户，是使用它们提供给查看器的元素来通知对列表的选择和更改的。查看器处理下列各项事务中的所有繁琐工作：将索引映射回至元素、调整对象的过滤视图以及在必要时重新排序。

过滤和排序功能是通过为查看器指定查看器排序器 (ViewerSorter) 和 / 或查看器过滤器 (ViewerFilter) 来处理的。（除了列表查看器之外，还可以为树查看器和表查看器指定这些。）客户只需要提供可以比较或过滤列表中的对象的类。查看器根据指定顺序和过滤器来处理填充列表的详细信息，并在添加和除去元素时维护顺序和过滤器。

查看器并不是由客户扩展的。要定制查看器，可以用您自己的内容和标签提供程序来配置它。

➤ **ListViewer** 将列表中的元素映射至 SWT 列表控件。

➤ **TreeViewer** 显示 SWT 树窗口小部件中的分层对象。它处理关于展开和折叠各项的详细信息。不同 SWT 树控件（普通树、表树和复选框树）具有几种不同类型的树查看器。

➤ **TableViewer** 非常类似于列表查看器，但添加了查看表中每个元素的多列信息的功能。表查看器通过引入编辑单元这一概念显著地扩展了 SWT 表窗口小部件的功能。可以使用特殊单元编辑器来允许用户通过使用组合框、对话框或文本窗口小部件来编辑表单元。当用户编辑需要时，表查看器会处理这些窗口小部件的创建和布置。这是通过使用 `CellEditor` 类来完成的，例如，`TextCellEditor` 和 `CheckboxCellEditor`。对于仅当查看时才填充的虚拟表，表查看器仅运行指定数目的结果，而不考虑实际所创建的内容。

8.3.2 文本查看器 (TextViewer)

文本窗口小部件具有许多常见语义，例如，双击行为、撤销、着色以及按索引或者行来导航。`TextViewer` 是 `SWT StyledText` 窗口小部件的适配器。文本查看器为客户提供了文档模型，并管理从文档至由文本窗口小部件提供的样式文本信息的转换。

我们将会在高级应用中详细讲解文本查看器。

8.4 Viewer 的(查看器)体系结构

要了解查看器，您必须熟悉查看器的输入元素、它的内容、它的选项以及在它正在处理的窗口小部件中实际显示的信息之间的关系。

8.4.2 输入元素(Model)

输入元素是查看器正在显示（或编辑）的主要对象。从查看器的角度来看，输入元素可以是任何对象。不建议由输入元素来实现任何特定接口。（我们将在讨论内容提供程序时了解一下原因。）

查看器必须能够处理输入元素的更改。如果在查看器中设置了新的输入元素，则它必须根据新元素来重新填充它的窗口小部件，并取消它自己与前一输入元素的关联。对于每种类

型的查看器，注册为输入元素上的侦听器的语义和根据元素填充窗口小部件的语义是不同的。

8.4.2 内容查看器(ContentProvider)

内容查看器是这样一种查看器：具有严格定义的协议，用来获取来自于它的输入元素的信息。内容查看器使用两个专门的 helper 类 `IContentProvider` 和 `ILabelProvider` 来填充它们的窗口小部件和显示关于输入元素的信息。

`IContentProvider` 提供了基本的生存期协议，以将内容提供程序与输入元素进行关联以及处理输入元素的更改。为不同类型的查看器实现了多个专门的内容提供程序。最常用的内容提供程序是 `IStructuredContentProvider`，它可以提供给定输入元素的对象列表。它用于类似列表的查看器，例如，列表、表或树。通常，内容提供程序知道如何在输入元素和期望的查看器内容之间进行映射。

`ILabelProvider` 更加深入。给定查看器的内容（派生自输入元素和内容提供程序），它可以生成在查看器中显示内容所需的特定用户界面元素，例如，名称和图标。标签提供程序可以帮助保存图标资源，因为它们可以确保图标的相同实例用于查看器中所有相似类型。

注意：特定内容和标签提供程序的实例并不会在多个查看器之间共享。即使所有查看器都使用相同类型的内容或标签提供程序，也应该利用每个查看器自己的提供程序类的实例来进行初始化。提供程序有效期协议被设计为用于提供程序与它的查看器之间的“一对一”关系。

输入元素、内容提供程序和标签提供程序允许查看器隐藏有关填充窗口小部件的实现的绝大部分详细信息。查看器的客户只需要关心是否使用了正确类型的输入和内容提供程序来填充查看器。标签提供程序必须知道如何从查看器内容中派生出用户界面信息。

标签提供程序不仅仅可以显示文本和图像。`JFace` 提供了几个类和接口来支持最常用的附加功能。`TableViewer`、`AbstractTreeViewer` 和 `TableTreeViewer` 支持下列类。

➤ **IColorProvider**。如果标签提供程序实现了 `IColorProvider`，它将在返回颜色时设置查看器中某个项的前景色和背景色。除非使用了系统颜色，否则建议对这些颜色进行高速缓存以最大程度地降低使用的系统资源量。

➤ **IFontProvider**。如果标签提供程序实现了 `IFontProvider`，它将在返回颜色时设置查看器中某个项的字体。也应该对字体进行高速缓存。由于每次调用 `Control#getFont()` 时都会创建 `Font` 的新实例，所以应该尽量减少调用它的次数。

➤ **ILabelDecorator**。`ILabelDecorator` 是一个对象，它可以使用图像或文本并对它们添加修饰。

➤ **DecoratingLabelProvider**。`DecoratingLabelProvider` 是一个复合对象，它接收标签提供程序和 `ILabelDecorator`。这使标签提供程序能够与修饰机制（如工作台提供的修饰机制）相关联。

➤ **IViewerLabelProvider**。`IViewerLabelProvider` 是一个标签提供程序，它允许外部对象（如修饰符）构建标签。`DecoratingLabelProvider` 是一个 `IViewerLabelProvider`。

➤ **IDelayedLabelDecorator**。`IDelayedLabelDecorator` 是一个 `ILabelDecorator`，它支持延迟的修饰（如在 `Thread` 中进行修饰的 `IDecoratorManager`）。`IDecoratorManager` 是 `IDelayedLabelDecorator`。可以通过调用 `IWorkbench#getDecoratorManager()` 获取工作台 `IDecoratorManager`。

➤ **IColorDecorator**。`IColorDecorator` 是一个对象，它可以支持修饰前景色和背景色。工作台 `DecoratorManager` 以内部方式支持此对象。

➤ **IFontDecorator**。`IFontDecorator` 是一个对象，它可以支持修饰字体。工作台 `DecoratorManager` 以内部方式支持此对象。

对于 Eclipse 3.1 来说，可以通过两种方法来影响视图的颜色 — 通过它自己的标签提供程序或者通过用于设置颜色和字体的修饰符。通常，最好使用标签提供程序中的颜色和字体支持来作为修饰符，从而影响每个显示特定类型的视图。如果确实使用了颜色或字体修饰符，则确保可以在“颜色和字体”首选项页面上设置它的值。

8.4.3 查看器和工作台

可以通过观察工作台如何使用查看器、内容提供程序和标签提供程序来体现它们的灵活性。

`WorkbenchContentProvider` 是一个结构化内容提供程序，它通过询问输入元素的子代来获取输入元素的内容。再次使用适配器的概念以实现通用功能。当从它的输入元素中请求元素列表时，`WorkbenchContentProvider` 将获取输入元素的 `IWorkbenchAdapter`。如果已经为输入元素注册了 `IWorkbenchAdapter`，则内容提供程序可以放心地假定可对元素的子代来查询该元素。`WorkbenchContentProvider` 在工作空间更改时还执行必需的工作来保持它的查看器是最新的。

`WorkbenchLabelProvider` 是一个标签提供程序，它从对象中获取 `IWorkbenchAdapter`，以查找它的文本和图像。标签提供程序的概念对于工作台对象特别有帮助，因为它允许单个标签提供程序高速缓存查看器中常用的图像。例如，一旦 `WorkbenchLabelProvider` 获取要用于 `IProject` 的图像，则它可以高速缓存该图像，并将它用于查看器中显示的所有 `IProject` 对象。

通过定义常见适配器 `IWorkbenchAdapter`，并将它向许多平台类型注册，就可以在许多常见查看器和包含它们的工作台视图中正确地表示这些类型。

8.5 操作和添加项

操作类允许您定义用户命令而不考虑它们在用户界面中的表示。这使您能够灵活地更改操作在插件中的表示，不必在选择某个命令时更改实际执行该命令的代码。添加项类用来管理表示命令的实际用户界面项。您不需要对添加项类编程，但是，您将在某些工作台和 `JFace` API 中看到它们。

我们在上面的讲述中已经不至一次讲到了操作（`Action`）以及添加项，例如在创建 `CustomerViewerPart` 的时候，我们就为它添加了很多 `Action`，现在我们来详细了解一下 `Action`。

8.5.1 操作

操作（`IAction`）表示可以由最终用户触发的命令。操作通常与按钮、菜单项和工具栏中的项相关联。

尽管操作本身不放置在用户界面中，但是它们确实具有面向用户界面的属性，例如，工具提示文本、标签文本和图像。这允许其它类为操作的表示构造窗口小部件。

当用户触发用户界面中的操作时，就会调用该操作的 `run` 方法来执行实际工作。`run` 方法中的常见模式是查询工作台选择并处理所选的对象。另一种常见模式是在选择操作时启动向导或对话框。

您不必直接实现 `IAction` 接口。而是应该对 `Action` 类划分子类。浏览此类的子类，以便了解操作的许多常见模式。以下代码实现了“关于”操作。它是工作台最简单的操作之一。

```
public void run() {  
    new AboutDialog(workbenchWindow.getShell()).open();  
}
```

以前我们了解了工作台接口 `IViewActionDelegate` 和 `IEditorActionDelegate`。在向工作台添加视图操作或编辑器操作时，就会使用这些接口。工作台操作代表是利用与它们相关联的视图或编辑器来初始化的。借助此认知，它们可以浏览工作台页面或窗口，访问执行操作所需要的选择或任何其它信息。

每当您想要在插件中定义命令时，就将实现您自己的操作类。如果您正在向其它视图和编辑器添加操作，则将实现操作代表。

8.5.2 添加项

添加项（`IContributionItem`）表示操作的用户界面部分。更准确的说，它表示添加到共享用户界面资源的项，例如，菜单或工具栏。

添加项知道如何利用表示该添加的相应 `SWT` 项来填充特定的 `SWT` 窗口小部件。

当您正在为工作台用户界面添加操作时，不必担心创建添加项。当工作台为已定义的操作创建用户界面项时，以您的名义完成了此任务。

我们曾经在 Application 的 ActionBarContributor 中为 Wizard 和 Viewer 增加了 Workbench 提供的添加项。

8.5.3 添加项管理器

添加项管理器(IContributionManager)表示将在用户界面中提供的添加项的集合。可以通过使用已命名的添加标识来添加和插入添加项，以便按照相应的顺序来放置这些添加项。还可以根据标识来查找添加项和除去各个项。

IContributionManager 的每种实现都知道如何使用 SWT 窗口小部件的项来填充该窗口小部件。JFace 为菜单 (IMenuManager)、工具栏 (IToolBarManager) 和状态行 (IStatusLineManager) 提供了添加项管理器。

作为插件开发者，您不需要实现这些接口，但是，在 API 方法中，您将看到对其中某些管理器的引用。

8.6 进度操作 (Progress)

org.eclipse.jface.operations 包为需要进度指示器或允许用户取消操作的长时间运行的操作定义接口。将在工作台进程对话框和视图的实现中使用这些接口。

通常，插件应将 IProgressService 中提供的工作台支持用于长时间运行的操作，以便所有插件都将具有一致的进度表示。有关进度对话框和视图的可用支持的完整讨论，请参阅 工作台并行支持。此讨论的其余部分重点描述了由工作台使用的 JFace 操作基础结构的详细信息。

8.6.1 可运行程序和进度

平台运行时定义常见接口 IProgressMonitor，在运行长时间运行的操作时，它用来向用户报告进度。在为用户显示进度很重要的情况下，客户可以提供监视器来作为许多平台 API 方法中的参数。

JFace 为实现进度监视器的用户界面的对象定义了多个特定接口。

`IRunnableWithProgress` 是用于长时间运行的操作的接口。此接口的运行方法具有 `IProgressMonitor` 参数，该参数用来报告进度并检查用户的取消操作。

`IRunnableContext` 是用户界面中可报告进度的不同位置的接口。实现此界面的类可选择使用不同技巧来显示进度和运行操作。例如，`ProgressMonitorDialog` 通过显示进度对话框来实现此界面。`IWorkbenchWindow` 通过在工作台窗口的状态行中显示进度来实现此界面。`WizardDialog` 实现此界面以在向导状态行中显示长时间运行的操作。

注意：工作台用户界面为 `WorkspaceModifyOperation` 中的操作提供了附加支持。这个类简化了长时间运行的修改工作空间的操作的实现。它在 `IRunnableWithProgress` 与 `IWorkspaceRunnable`之间进行映射。

我们以一段代码为例子：

```
IRunnableWithProgress runnable = new IRunnableWithProgress () {

    public void run(IProgressMonitor monitor) throws
InvocationTargetException, InterruptedException {

        int i =0;

        monitor.beginTask("Creating...",100);

        while(i< 100){

            i++;

            monitor.worked(i);

            monitor.setTaskName("Creating.." + i);

            try{

                Thread.sleep(50);

            }catch(Exception e){

            }

        }

    }

}
```

```

        monitor.done();
    }

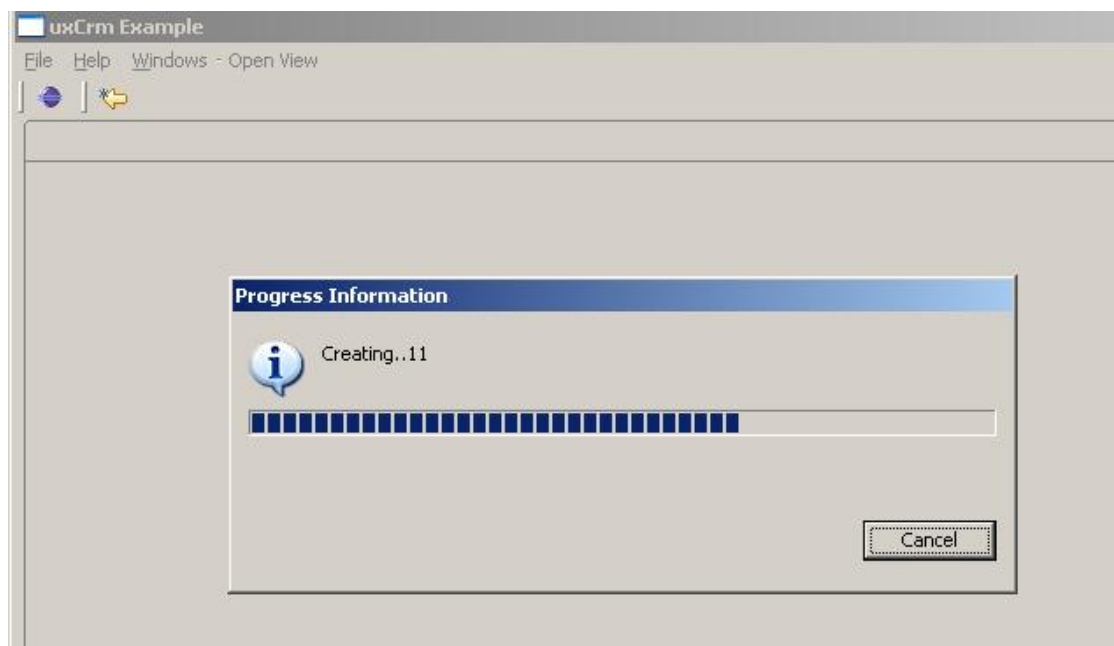
};

ProgressMonitorDialog dialog =
new ProgressMonitorDialog(getSite().getShell());

try {
    dialog.run(true,true,runnable);
} catch (InvocationTargetException e) {
    // TODO 自动生成 catch 块
    e.printStackTrace();
} catch (InterruptedException e) {
    // TODO 自动生成 catch 块
    e.printStackTrace();
}

```

我们模拟一个简单的操作，让一个任务从 1 执行到 100，中间等待 50 毫秒，下面是运行结果。



8.6.1 模态操作

提供 `ModalContext` 类用来运行客户机代码的透视图是模态的操作。它用于 `IRunnableContext` 的不同实现。如果插件需要等待长时间运行的操作完成才能继续执行，则可以使用 `ModalContext` 来完成此任务，并且仍然让用户界面保持响应。

在模态上下文中运行操作时，可以选择将该操作分配到不同的线程中。如果 `fork` 为 `false`，则操作将在调用的线程中运行。如果 `fork` 为 `true`，则操作将在新线程中运行，调用线程将停止，并且运行用户界面事件循环，直到该操作终止为止。

9. 深入 SWT（标准窗口小部件工具箱）

标准窗口小部件工具箱（SWT）是 Java 开发者的窗口小部件工具箱，它提供可移植的 API，并与底层本机操作系统图形用户界面平台紧密集成。

许多低级别的用户界面编程任务是在 Eclipse 平台的较高层处理的。例如，`JFace` 查看器和操作实现了应用程序与窗口小部件之间的普通交互作用。然而，SWT 的知识对于了解平台的其它部分是如何工作的是很重要的。

9.1 窗口小部件

SWT 包含许多功能部件，但是系统的核心的基础知识：窗口小部件、布局和事件，都是实现有用而且功能强大的应用程序所需要的。

当使用平台工作台扩展来添加用户界面元素时，用于启动 SWT 的机制是由工作台来处理的。

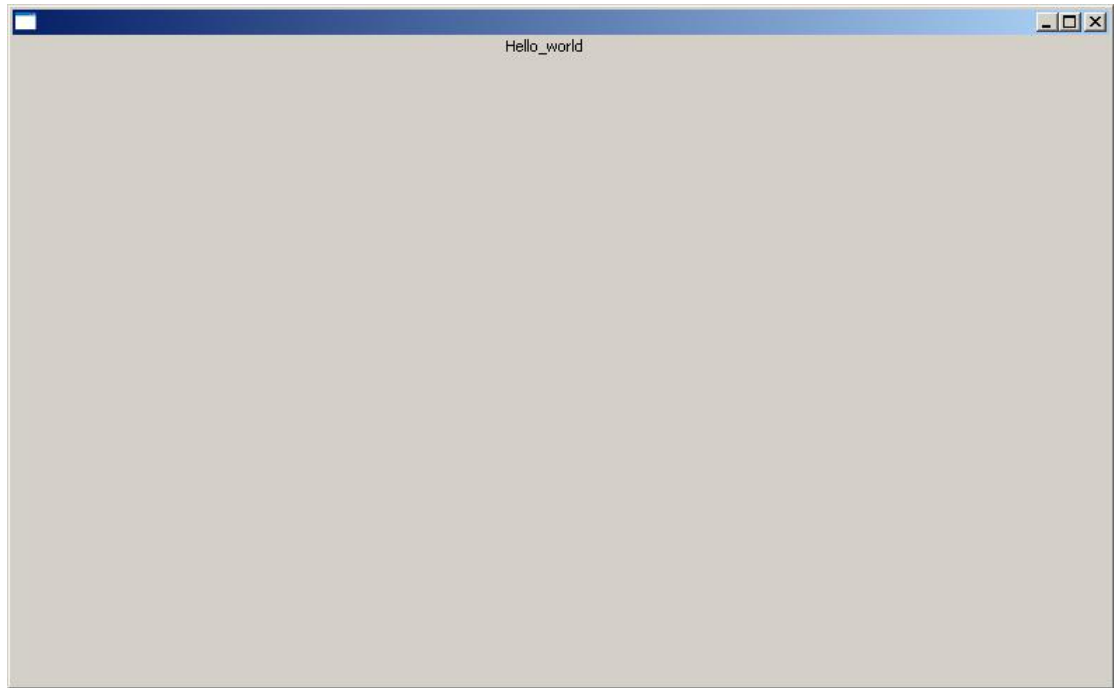
典型的独立 SWT 应用程序具有下列结构：

- 创建显示，它表示 SWT 会话。
- 创建一个或多个 `Shell`，它充当应用程序的主窗口。
- 创建 `shell` 内部需要的任何其它窗口小部件。
- 初始化窗口小部件的大小和其它必需的状态。为需要处理的窗口小部件事件注册侦听器。

- 打开 shell 窗口。
- 运行事件调度循环，直到发生退出情况为止（通常是在用户关闭主 shell 窗口的情况下）。
- 除去显示。

以下代码段改编自 `org.eclipse.swt.examples.helloworld.HelloWorld2` 应用程序。由于该应用程序只显示字符串“Hello World”，因此不需要向任何窗口小部件事件注册。

```
public static void main (String [] args) {  
    Display display = new Display ();  
    Shell shell = new Shell (display);  
    Label label = new Label (shell, SWT.CENTER);  
    label.setText ("Hello_world");  
    label.setBounds (shell.getClientArea ());  
    shell.open ();  
    while (!shell.isDisposed ()) {  
        if (!display.readAndDispatch ()) display.sleep ();  
    }  
    display.dispose ();  
}
```



9.1.1 Display

显示表示 SWT 与底层平台的图形用户界面系统之间的连接。显示主要用来管理平台事件循环和控制用户界面线程与其它线程之间的通信。对于大多数应用程序，可以遵循以上所使用的模式。在创建任何窗口之前必须创建显示，并且当关闭 `shell` 时，您必须除去显示。除非您正在设计多线程应用程序，否则，不需要太多考虑显示。

9.1.2 Shell

`Shell` 是由操作系统平台窗口管理器管理的一个“窗口”。顶级 `shell` 是作为显示的子代创建的那些 `shell`。这些窗口是在用户使用应用程序时移动、调整大小、最小化和最大化的窗口。辅助 `shell` 就是作为另一个 `shell` 的子代创建的那些 `shell`。这些窗口通常用作只存在于另一个窗口的上下文中的对话框窗口或其它瞬时窗口。

9.1.3 父代和子代

不是顶级 `shell` 的所有窗口小部件都必须具有父代。顶级 `shell` 没有父代，但是，它们是通过与特定显示进行关联而创建的。可以使用 `getDisplay()` 来访问它的 `Display`。所有其它窗口小部件都是作为顶级 `shell` 的子代（直接或间接）来创建的。

组合体窗口小部件是可以具有子代的窗口小部件。

当看到应用程序窗口时，可以将它看作窗口小部件树，或者看作层次结构，其根目录就是 `shell`。根据应用程序的复杂性，可以具有 `shell` 的单个子代、几个子代或者具有子代的组合体的嵌套层。

9.1.4 样式位

某些窗口小部件属性必须在创建窗口小部件时设置，并且不能进行后续更改。例如，列表可以是单个或多个选择，并且可以具有滚动条也可以没有滚动条。

这些属性(称为样式)是在构造函数中设置的。所有窗口小部件构造函数都采用 `int` 自变量，该自变量指定所有期望样式的位宽 OR。在某些情况下，认为特定样式是一种提示，这意味着它可能在所有平台上都不可用，但是在不支持它的平台上将适当地忽略它。

样式常量作为公共静态字段存在于 `SWT` 类中。`SWT` 的“API 参考”中包含了每个窗口小部件类的可应用常量的列表。

9.1.5 资源处理

在 `SWT` 下面的平台需要显式地分配和释放操作系统资源。为了配合反映窗口小部件工具箱中的平台应用程序结构的 `SWT` 设计原理，`SWT` 要求您显式地释放已经分配的任何操作系统资源。在 `SWT` 中，`Widget.dispose()` 方法用来释放与特定工具箱对象相关联的资源。

经验法则是，如果您创建对象，则您必须除去它。以下进一步说明此原理的特定规则：

- 如果使用构造函数来创建图形对象或窗口小部件，使用完时必须显式地将其除去。
- 当 `Composite` 被除去时，将递归地除去该组合体及其所有子窗口小部件。在此情况下，不需要除去窗口小部件本身。然而，必须释放与那些窗口小部件一起分配的所有图形资源。
- 如果在未使用构造函数（例如 `Control.setBackground()`）的情况下获取图形对象或窗口小部件，则不要将其除去，这是因为您未分配它。

➤ 如果将对窗口小部件或图形对象的引用传送至另一个对象，则一定要小心，仍在使用它时一定不要除去它。（与在 使用图像的插件模式中所描述的规则相似。）

➤ 如果创建图形对象以便在其中一个窗口小部件的生命周期内使用它，则在除去窗口小部件时必须除去图形对象。这可以通过向窗口小部件注册除去侦听器，并在接收到除去事件时释放图形对象来实现。

这些规则有一个例外。简单的数据对象（例如，矩形和 点）不使用操作系统资源。它们没有 `dispose()` 方法，您也不需要释放它们。

9.2 布局

通常，处理简单窗口小部件定位的最佳方法是在大小调整事件侦听器中进行处理。然而，当放置窗口小部件时，应用程序使用公共模式。这些模式可以构造为可配置的布局算法，许多不同的应用程序都可以重用该算法。

SWT 定义了布局，它提供组合体中子窗口小部件的通用定位和缩放。布局是抽象类 布局的子类。可以在 `org.eclipse.swt.layout` 包中找到 SWT 标准布局。

在对窗口小部件进行大小调整和定位时，将使用一些一般的定义：

- 窗口小部件的位置是它在父代窗口小部件中的 `x` 和 `y` 坐标位置。
- 窗口小部件的首选大小是显示其内容所需要的最小大小。每一类窗口小部件的计算方法都不相同。
- `clientArea` 是可以放置子代而不必对该子代进行裁剪的区域。
- 修剪是窗口小部件的客户区域与它的实际边界之间的距离。修剪被窗口小部件的边框或边框边缘的额外空间占用。修剪的大小和外观与窗口小部件和平台有关。

这些概念与应用程序有关，不管是否使用了布局。可以认为布局是打包调整大小功能以便重用的便捷方法。

布局还引入了一些附加概念：

- 某些布局支持布局中的各个窗口小部件之间存在间距。
- 某些布局支持布局的边缘与和边缘相邻的窗口小部件之间存在页边距。

有关进一步的讨论和用来演示这些概念的图片，请参阅 [了解 SWT 中的布局](#)。

以下代码段说明了一种简单情况，应用程序使用调整大小回调来将标签的大小调整为它的父代 shell 的大小：

```
Display display = new Display ();

Shell shell = new Shell (display);

Label label = new Label (shell, SWT.CENTER);

shell.addControlListener(new ControlAdapter() {

    public void controlResized(ControlEvent e) {

        label.setBounds (shell.getClientArea ());

    }

});
```

以下代码段使用布局来获得相同的效果：

```
Display display = new Display ();

Shell shell = new Shell (display);

Label label = new Label (shell, SWT.CENTER);

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

即使对于此简单示例，使用布局也减少了应用程序代码。对于更复杂的布局，简化更大。

下表总结了 SWT 提供的标准布局。

下面的几节我们将按照 Eclipse 官方的教程进行讲解。

布局	作用
填充布局	将控件安排成一行或一列，并强制使它们大小相同。
表单布局	通过使用 <code>FormAttachments</code> 可选地配置每个子代的左边、顶边、右边和底边来定位子代。
网格布局	将子代放置成行和列。
行布局	将子代放置成水平的行或垂直的列。

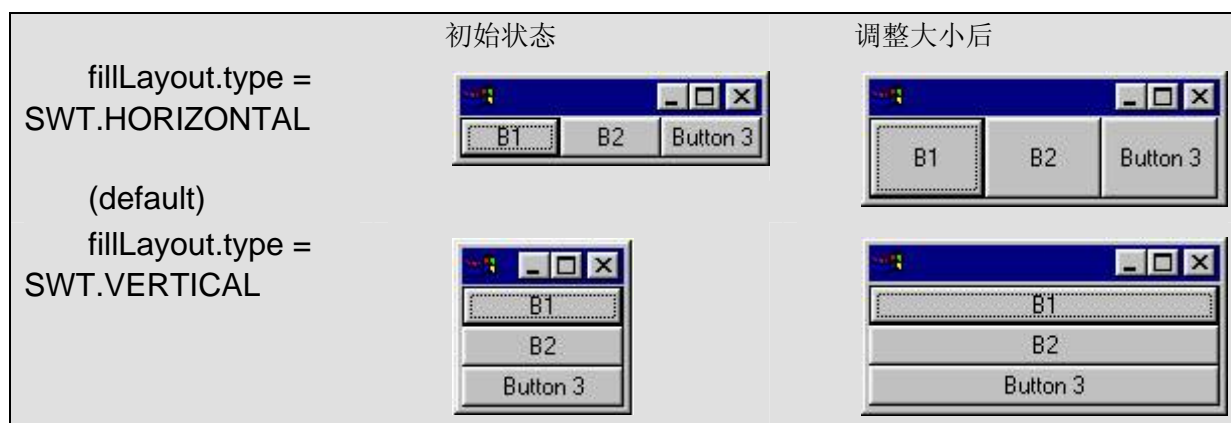
9.2.1 FillLayout

`FillLayout` 是最简单的布局类，它把组件摆放在一行或者一列，并强制组件大小一致。一般的，组件的高度与最高组件一致，宽度与最宽组件一致。`FillLayout` 不折行，不能设置边界距离和间距。可以使用它布局任务栏或工具栏，或者在 `Group` 中的一组选择框。当容器只有一个子组件时也可以使用它。例如如果一个 `Shell` 只有一个 `Group` 子组件，`FillLayout` 将使 `Group` 完全充满 `Shell`。

以下是相关代码。首先创建了一个 `FillLayout`，然后设置它的 `type` 域值为 `SWT.VERTICAL`，再把它设置到容器上（一个 `Shell`）。示例的 `Shell` 有三个按钮，`B1`，`B2`，和 `Button 3`。注意在 `FillLayout` 中，所有的子组件尺寸都相同，并且充满了全部可用的空间：

```
FillLayout fillLayout = new FillLayout();
fillLayout.type = SWT.VERTICAL;
shell.setLayout(fillLayout);
new Button(shell, SWT.PUSH).setText("B1");
new Button(shell, SWT.PUSH).setText("Wide Button 2");
new Button(shell, SWT.PUSH).setText("Button 3");
```

下图显示了水平和垂直布局时，以及在初始状态和调整大小之后，`FillLayout` 的不同表现：



9.2.2 RowLayout

RowLayout 比 FillLayout 更常用，它可以提供折行显示，以及可设置的边界距离和间距。它有几个可设置的域。另外可以对每个组件通过 `setLayoutData` 方法设置 RowData，来设置它们的大小。

9.2.2.1 RowLayout 的可设置域

➤ **type** (*2.0 新添加*)

type 域控制 RowLayout 是水平还是垂直布局组件。默认为水平布局。

➤ **wrap**

wrap 域控制 RowLayout 在当前行没有足够空间时是否折行显示组件。默认折行显示。

➤ **pack**

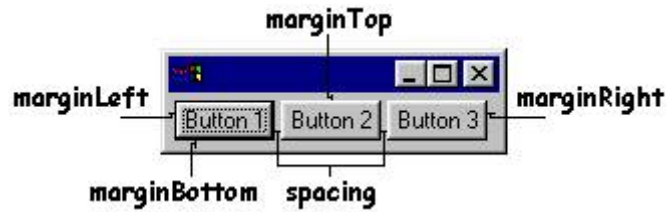
pack 域为 true 时，组件使用他们的原始尺寸，并且排列时尽量远离左边（and they will be aligned as far to the left as possible? ）。如果 pack 域为 false，组件将填充可用的空间，跟 FillLayout 类似。默认为 true。

➤ **justify**

justify 域为 true 时，组件将在可用的空间内从左到右伸展。如果容器变大了，那么多余的空间被平均分配到组件上。如果 pack 和 justify 同时设为 true，组件将保持它们的原始大小，多余的空间被平均分配到组件之间的空隙上。默认 false。

MarginLeft, MarginTop, MarginRight, MarginBottom 以及 Spacing

这些域控制组件之间距离（spacing，均以像素记），以及组件与容器之间的边距。默认的，RowLayout 保留了 3 个像素的边距和间距。下图示意了边距和间距：



9.2.2.2 RowLayout 示例

下面的代码创建了一个 RowLayout，并设置各个域为非默认值，然后把它设置到一个 Shell：

```
RowLayout rowLayout = new RowLayout();
rowLayout.wrap = false;
rowLayout.pack = false;
rowLayout.justify = true;
rowLayout.type = SWT.VERTICAL;
rowLayout.marginLeft = 5;
rowLayout.marginTop = 5;
rowLayout.marginRight = 5;
rowLayout.marginBottom = 5;
rowLayout.spacing = 0;
shell.setLayout(rowLayout);
```

如果使用默认设置，只需要一行代码即可：

```
shell.setLayout(new RowLayout());
```

下图显示了设置不同域值的结果：

初始状态	调整尺寸后
------	-------

```

wrap = true
pack = true
justify =
false
type =
SWT.HORIZONTAL

```

(默认)

```

wrap =
false
(没有足够空
间时裁边)

```

```

pack =
false
(所有组件尺
寸一致)

```

```

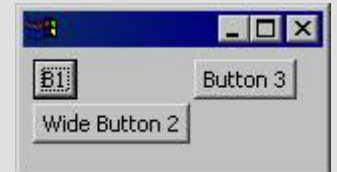
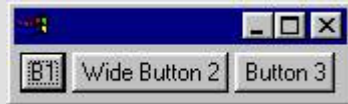
justify =
true
(组件根据可
用空间进行伸展)

```

```

type =
SWT.VERTICAL
(组件垂直按
列排列)

```



9.2.2.3 在 RowLayout 上配合使用 RowData

每个由 RowLayout 控制的组件可以通过 RowData 来设置其的原始的尺寸。以下代码演示了使用 RowData 改变一个 Shell 里的按钮的原始尺寸：

```

import org.eclipse.swt.*;

import org.eclipse.swt.widgets.*;

import org.eclipse.swt.layout.*;

public class RowDataExample {

    public static void main(String[] args) {

```

```

Display display = new Display();
Shell shell = new Shell(display);
shell.setLayout(new RowLayout());
Button button1 = new Button(shell, SWT.PUSH);
button1.setText("Button 1");
button1.setLayoutData(new RowData(50, 40));
Button button2 = new Button(shell, SWT.PUSH);
button2.setText("Button 2");
button2.setLayoutData(new RowData(50, 30));
Button button3 = new Button(shell, SWT.PUSH);
button3.setText("Button 3");
button3.setLayoutData(new RowData(50, 20));
shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) display.sleep();
}
}

```

以下是运行结果:



9.2.3 GridLayout

GridLayout 可能是最常用的、功能最强大的标准布局类了，当然它也最复杂。GridLayout 把容器里的组件摆放在一个格子里，它有许多可设置的域，并且同 RowLayout 类似，组件可以有相应的布局数据，称作 GridData。GridLayout 的强大在于它可以通过 GridData 来设置每一个控件。

9.2.3.1 GridLayout 的可设置域

➤ numColumns

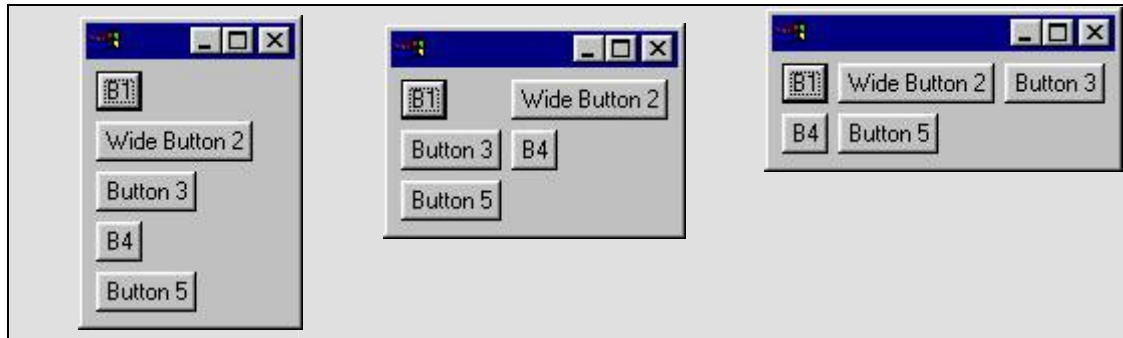
numColumns 域是 GridLayout 的最重要的域，并且通常是第一个需要设置的域。组件从左到右摆放在列里，当 numColumns + 1 个组件添加到容器中时，将创建一个新行。默认只有一列。以下代码创建了由 GridLayout 管理的含有 5 个具有不同宽度的按钮的 Shell，随后的列表显示了当 numColumns 设为 1, 2 或 3 时的效果。

```
Display display = new Display();
Shell shell = new Shell(display);
GridLayout gridLayout = new GridLayout();
gridLayout.numColumns = 3;
shell.setLayout(gridLayout);
new Button(shell, SWT.PUSH).setText("B1");
new Button(shell, SWT.PUSH).setText("Wide Button 2");
new Button(shell, SWT.PUSH).setText("Button 3");
new Button(shell, SWT.PUSH).setText("B4");
new Button(shell, SWT.PUSH).setText("Button 5");
shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) display.sleep();
}
```

numColumns = 1

numColumns = 2

numColumns = 3



➤ **makeColumnsEqualWidth**

makeColumnsEqualWidth 域强制各列具有相同的宽度。默认为 false。

➤ **MarginWidth, MarginHeight, HorizontalSpacing, 以及 VerticalSpacing**

GridLayout 边距和间距域与 RowLayout 的类似，不同的是左边距和右边距统一成 marginWidth，上边距和下边距统一成 marginHeight。同样可以分别设置 horizontalSpacing 和 verticalSpacing (RowLayout 中的间距根据它的 type 类型设置水平间距或者垂直间距)。

9.2.3.2 GridData 对象的域

GridData 是 GridLayout 对应的布局数据，可以通过 setLayoutData 设置组件的布局数据。例如，可以采用如下代码设置按钮的 GridData：

```
Button button1 = new Button(shell, SWT.PUSH);
button1.setText("B1");
button1.setLayoutData(new GridData());
```

以上代码创建了一个含有默认值的 GridData 对象，其效果和没有设置布局数据是一样的。有两种方式可以创建含有指定域值的 GridData 对象。第一种方式就是直接设置各个域值，例如：

```
GridData gridData = new GridData();
gridData.horizontalAlignment = GridData.FILL;
```

```
gridData.grabExcessHorizontalSpace = true;

button1.setLayoutData(gridData);
```

第二种方式是通过利用便利的 API 来设置 GridData 的风格位:

```
button1.setLayoutData(new
GridData(GridData.HORIZONTAL_ALIGN_FILL |
GridData.GRAB_HORIZONTAL));
```

实际上, 为了方便还提供了一些风格位的组合, 例如:

```
button1.setLayoutData(new
GridData(GridData.FILL_HORIZONTAL));
```

注意 FILL_ 风格同时设置对齐方式和占位方式。GridData 的风格位只对布尔值和枚举值有效, 数字域需要直接设置。

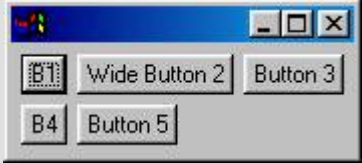
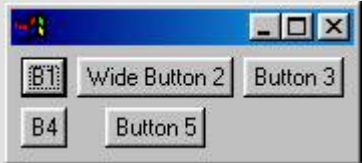

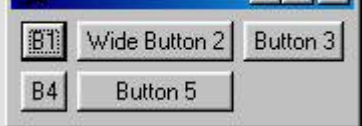
➤ HorizontalAlignment 及 VerticalAlignment

alignment 域指定组件在格子里按照水平或者垂直方式摆放。可以设置以下值:

- BEGINNING
- CENTER
- END
- FILL

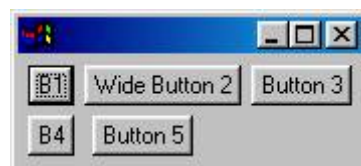
默认 horizontalAlignment 设为 BEGINNING (左对齐), verticalAlignment 设为 CENTER。

参考前面的五个按钮的例子, 设置三列, 并对 Button 5 设置不同的 horizontalAlignment, 如图:

<pre>horizontalAlignment = GridData.BEGINNING</pre> <p>(默认)</p>	
<pre>horizontalAlignment = GridData.CENTER</pre>	
<pre>horizontalAlignment = GridData.END</pre>	
<pre>horizontalAlignment = GridData.FILL</pre>	

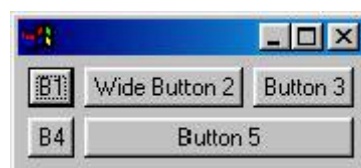
➤ **HorizontalIndent**

`horizontalIndent` 域可以使组件向右移动指定的像素值。只有当 `horizontalAlignment` 设为 `BEGINNING` 时有效。不能通过风格位设置缩进，以下代码演示了把 `Button 5` 缩进 4 个像素：



```
GridData gridData = new GridData();
gridData.horizontalIndent = 4;
button5.setLayoutData(gridData);
HorizontalSpan 及 VerticalSpan
```

`span` 域可以使控件跨越几个格子，常常与 `FILL` 风格一起使用。以下代码把 `Button 5` 扩展到了两个格子：



```
GridData gridData = new GridData();
```

```
gridData.horizontalAlignment = GridData.FILL;  
gridData.horizontalSpan = 2;  
button5.setLayoutData(gridData);
```

如果要把 Button 2 扩展到两个格子，可以这样写：



```
GridData gridData = new GridData();  
gridData.horizontalAlignment = GridData.FILL;  
gridData.horizontalSpan = 2;  
button2.setLayoutData(gridData);
```

还可以把 Button 3 垂直扩展两个格子：



```
GridData gridData = new GridData();  
gridData.verticalAlignment = GridData.FILL;  
gridData.verticalSpan = 2;  
button3.setLayoutData(gridData);  
  
GrabExcessHorizontalSpace 及 GrabExcessVerticalSpace
```

`grabExcessHorizontalSpace` 和 `grabExcessVerticalSpace` 主要用在象 `Text`，`List` 及 `Canvas` 这样的重量级组件上，当它们所在的容器增大时，可以使它们自动增大。例如如果设置一个 `Text` 组件可以横向扩展，那么当用户调整 `Shell` 的宽度时，`Text` 组件会自动扩展填满新的横向空间，而同一行上的其他组件保持宽度不变。当然，当 `Shell` 变小时，设置了此属性的组件也是首先收缩的。在可以调整容器大小的环境中，很

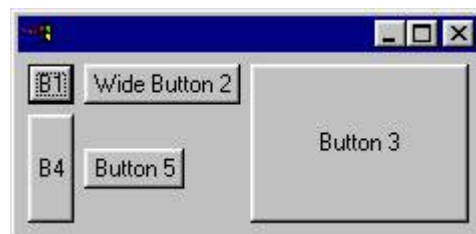
容易的就能想到设置 `grabExcessSpace` 域。作为例子，我们仍然采用前面 Button 3 垂直扩展两个单元格的例子，如下：



如果我们调整窗口的大小，那么只有窗口变大了：



现在设置 Button 3 可以横向和纵向扩展，B1 和 B4 仅纵向填充（不扩展），重新调整之后，如图：



这一次，Button 3 在两个方向上变化了，B4 只在纵向上变化，其它保持不变。这是由于设置了 Button 3 在纵向上扩展，最后一行变高了的的结果。注意 B1 没有变化，尽管对它设置了纵向填充，因为它所在的行没有变化。并且 Button 3 还设置了横向扩展和填充，它所在的列变宽了，所以它也变宽了。以下是代码：

```
Button button1 = new Button(shell, SWT.PUSH);
button1.setText("B1");

GridData gridData = new GridData();
gridData.verticalAlignment = GridData.FILL;
button1.setLayoutData(gridData);
```

```

new Button(shell, SWT.PUSH).setText("Wide Button 2");

Button button3 = new Button(shell, SWT.PUSH);
button3.setText("Button 3");

gridData = new GridData();
gridData.verticalAlignment = GridData.FILL;
gridData.verticalSpan = 2;
gridData.grabExcessVerticalSpace = true;
gridData.horizontalAlignment = GridData.FILL;
gridData.grabExcessHorizontalSpace = true;
button3.setLayoutData(gridData);

Button button4 = new Button(shell, SWT.PUSH);
button4.setText("B4");

gridData = new GridData();
gridData.verticalAlignment = GridData.FILL;

button4.setLayoutData(gridData);

new Button(shell, SWT.PUSH).setText("Button 5");

```

在典型的应用程序窗口中，通常会设置至少一个组件可以扩展。如果有多于一个的组件可以扩展，那么它们平均分配扩展的空间，如图所示：



最后需要注意的是，如果一个组件可以横向扩展，并且它的父容器变宽了，那么组件所在的整列都变宽了。同样如果组件可以纵向扩展，并且其父容器变高了，那么组件所在的整行都变高了。这样如果相应的行或列里有其它的组件设置了填充（fill）属性，那么这些组件也会被拉伸。具有左对齐、居中对齐、右对齐的组件不会被拉伸，它们仍然在所在的行或列里左对齐、居中对齐或右对齐。

➤ **WidthHint 及 HeightHint**

`widthHint` 和 `heightHint` 域指示了希望组件可以具有的宽度和高度，前提是不与 `GridLayout` 其他要求约束矛盾。参见前面五个按钮、三列的例子，假设要设置 `Button 5` 宽 70 像素、高 40 像素，代码如下：

```
GridData gridData = new GridData();  
gridData.widthHint = 70;  
gridData.heightHint = 40;  
button5.setLayoutData(gridData);
```

`Button 5` 自然尺寸如左图所示，右图是 70 像素宽、 40 像素高的图示：



注意，如果 `Button 5` 的 `horizontalAlignment` 设置为 `FILL`，那么 `GridLayout` 不能满足其宽为 70 像素的请求。

最后一点，在一个平台上表现好的设置，在另一个平台上可能会有差别。由于不同平台之间的字体大小和组件的自然大小不一样，因此硬编码像素值不是布局窗体的最好方法。因而，除非万不得已，尽量少用 `size hint`。

9.2.1 定制布局

有时，可能需要编写您自己的定制 `Layout` 类。当您具有用于应用程序中的许多不同位置的复杂布局时，这样作最合适。注意，除非您正在编写将供一些组合体窗口小部件使用的很普通的布局，否则，计算大小和定位调整大小侦听器中的子代有时更简单且更容易。

布局负责实现两种方法：

- 一旦根据布局算法来确定了组合体的所有子代的大小和位置，

`computeSize(...)` 就会计算打包组合体的所有子代的矩形的宽度和高度。提示参

数允许限制宽度和 / 或高度。例如，如果布局在一个维中受到约束，则布局可以选择在另一个维中增大。

➤ `layout(...)` 定位组合体的子代并调整其大小。布局可以选择高速缓存与布局相关的信息，例如，每个子代的首选范围。`flushCache` 参数告诉 布局将高速缓存的数据清仓，当已经更改了除了组合体的大小之外的其它因素时（例如，创建或删除子代，或者更改了窗口小部件的字体），就需要这样做。

可以选择实现第三个方法 `flushCache(...)` 以清除与特定控件相关联的任何高速缓存数据。通常，窗口小部件的 `computeSize()` 方法的成本较高，因此布局可以对结果进行高速缓存以提高性能。

三. Eclipse RCP 开发 (2)

从这一章开始，我们将介绍一些 RCP 的高级应用，诸如文本编辑器、`IAdaptable` 的使用、`Selection Service` 等。

1. 选择服务

1.1 Eclipse 中的“选择服务”

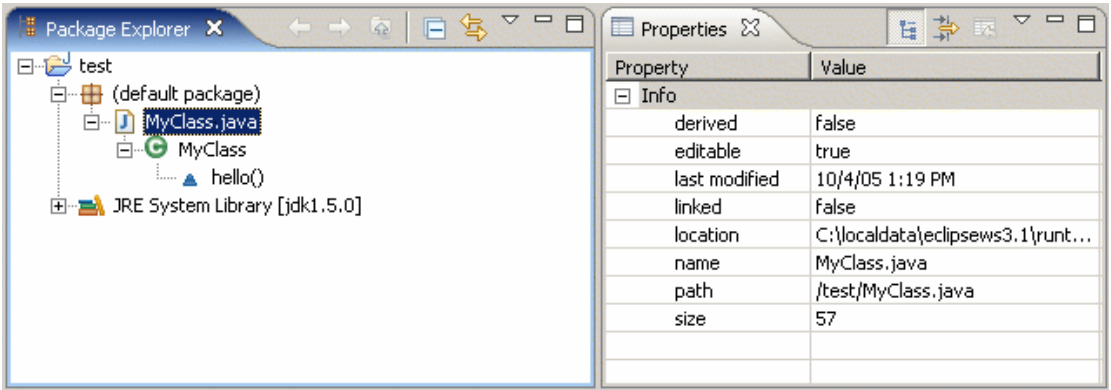
`EclipseWorkbench` 作为一个能构建 RCP 和其他 `Application` 的强大 UI 框，为高度集成和灵活扩展用户接口提供了许多服务，其中由一个服务方面，就是 `ViewPart` 的整合。

在我们前面所提到的 CRM 开发中，其中有一节提到了，当选择到某一个 `Customer` 的时候，点击右键利用菜单中的 `MenuItem` 显示一个 `Customer` 的 `RelationView`。在 `Eclipse Workbench` 中，有一个监听机制，它会将一些可发送事件的事件源的动作分发都注册在 `Workbench` 中的监听器。这一类事件源被称作 `SelectionProvider`，监听器则是 `SelectionChangeListener`。

因为一般情况下，在我们的 RCP 中，常常会出现改变当前选择项的动作，例如我们点击一个 `Table` 上的某项，这时候就会产生一个 `selectionChanged` 的事件，这一个动作

是由类似于查看器一类的 UI 组件发出的，比如 TableViewer，TreeViewer 等。

所谓的这一类整合，是为了对于在 Workbench 中选择了某个模型后，ViewPart 中的模型自动更新时，提供更多的支持。最经典的例子莫过于属性页（Property），当我们在 Workbench 中选择一个模型时，Property 页面上就会显示出该模型对应的属性信息。



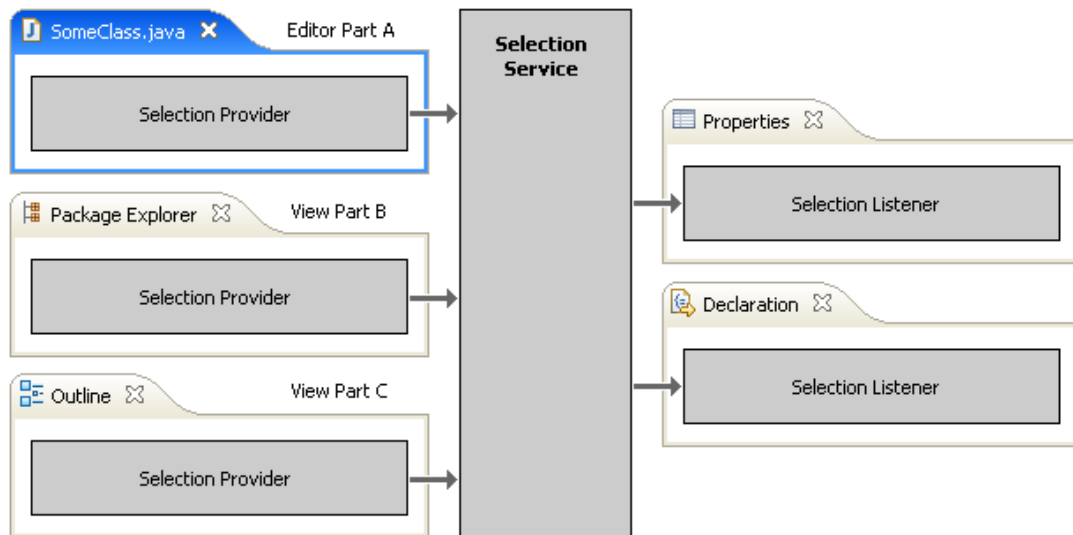
这种“选择服务”的另一个方面，是针对那些 Action 的，每当选中的 Workbench 中的元素时候，这些 Action 可以根据选择的 Element 来判断自身当前是否可用（enable）。

1.2 Selection Service 的整体架构

每个 Workbench window 都具有属于自己的 Selection Service（选择服务）实例，这些服务（Service）跟踪在当前激活页面中所选择元素的改变情况，然后把这些信息发放给注册在服务实例中的那些监听器。

每当在当前页面下的选择元素发生改变，或者当另一个页面被激活，这一类的选择性事件就会发生。

这两种改变方式所激发出来的事件，开发人员是可以利用起来做到页面之间的挥动以及交互方案的。



在很多情况下，我们会试图去监听或者跟踪一些选择变化，利用 Selection Service 的这一机制这些元素或者是文本被选中的 ViewPart 就没有必要去知道谁对这些改变做了跟踪，或者说对目前的这些选择改变相当关注，所以，我们如果想创建一个新的 ViewPart，并且这个 ViewPart 依赖于一个已有的 View 的选择元素的话，只需要修改这个已有 ViewPart 上的一两行代码。

1.3 能被选中的元素

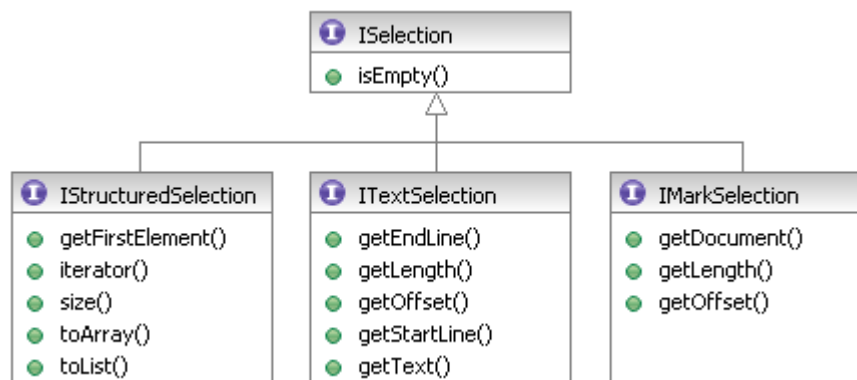
从用户角度来看，一个被选择元素，是一组在 Table 或者 Tree 控件中高亮现实的一组实例，另一种被选择元素也可以是一段在编辑器中被选中文本。

这谢可视化的元素在 Workbench 平台后面，都是一个个的 Java 对象，这种 JFace 的 MVC 实现对这些模型对象和可视化的元素进行了有效地映射。

这些可被选择的元素，是一组在 Workbench 中被选中的图形元素对应的模型的数据结构，正如前面我们提出的两种不同的 Selection：

- 一组对象
- 一段文本

它们是可以为 NULL 的，比如，一段空的列表或者字符长度为 0 的文本字符串。Eclipse 在将它们这种数据结构描述成了一些接口：



上图所示的 **IStructuredSelection** 对应了一组对象，而 **ITextSelection** 和 **IMarkSelection** 则事描述了一段被选中的文本。

Eclipse 在实现的时候给出了一组默认的实现类：

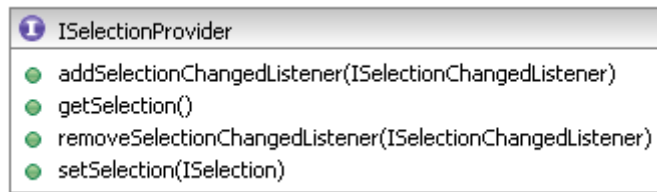
- org.eclipse.jface.viewers.StructuredSelection
- org.eclipse.jface.text.TextSelection
- org.eclipse.jface.text.MarkSelection

这些实现类经常应用在将低级别的 SWT 事件转化到 **ISelection** 对象中，这些实现类在某些需要将被选择元素利用代码表示出来的时候也相当有用，例如：

```
ISelection sel = new StructuredSelection(presetElement);
treeview.setSelection(sel);
```

1.4 通知 Workbench

所有的 JFace 查看器 (Viewer) 都能可以被称作“选择提供者” (Selection Provider)，一个选择提供者必须实现接口：**ISelectionProvider**：



我们在前面的介绍中其实已经提到了这些问题。

不同的 JFace 查看器所对应的选择对象类型是不同的,我们给出一个简单的列表展示一下:

查看器类型	选择类型
ComboViewer	IStructuredSelection
ListViewer	IStructuredSelection
TreeViewer	IStructuredSelection
+ -CheckboxTreeViewer	IStructuredSelection
TableViewer	IStructuredSelection
+ - CheckboxTableViewer	IStructuredSelection
TextViewer	ITextSelection, IMarkSelection
+ - SourceViewer	ITextSelection, IMarkSelection
+ - ProjectionViewer	ITextSelection, IMarkSelection

一个用户定制的查看起也可以去实现 ISelectionProvider 接口来变成一个 Selection Provider。

当一个 Workbench Part 得到一个 Viewer 的时候,需要把这个已经成为了 Selection Provider 的查看器利用 View Site 注册进去:

```
getSite().setSelectionProvider(tableviewer);
```

这样 View 本身就不必实现 ISelectionProvider 接口了,但是实现的效果同上面的方式实现的是一样的。

作为事件监听的另一端,则更为简单一些。只需要实现 ISelectionListener 接口,并注册在 Site 中:

```
site.getPage().addSelectionListener(this);
```

然后实现 `public void selectionChanged(IWorkbenchPart part, ISelection selection) {}` 方法即可。这样，当 `SelectionProvider` 中的选择发生改变时，这个视图中的 `selectionChanged()` 方法就会被调用。

注意 `SelectionProvider` 和 `SelectionListener` 并不直接对应。这个地方有一点容易混淆，就是 Eclipse 实际上提供有两套与 `Selection` 相关的事件与接口：

➤ `ISelectionChangedListener` 对应 `ISelectionProvider`

这个是 `JFace` 中的选择监听机制，对试图或者控件而言，它提供对原始的选择事件的通知与响应。`ISelectionProvider` 需要注册在 `Site` 上，以供 `ISelectionService` 使用，将选择事件扩散到其他的 `ISelectionListener` 中。

➤ `ISelectionListener` 对应 `ISelectionService`

这个是在 `Site` 中使用的，`ISelectionService` 不需要自己实现，已经实现好了，`ISelectionListener` 则需要注册到 `ISelectionService` 上，以对其它 `SelectionProvider` 的事件响应进行监听。

在 Eclipse 本身的实现中，`PropertySheet` 和 `Outline` 都使用了这种机制。不过需要注意的是，缺省的 `PropertySheet` 需要接受一个 `IStructuredSelection`，而不仅仅是一个 `ISelection`。因此，如果 `ISelectionProvider` 需要提供一个能够让 `PropertySheet` 进行显示的对象的话，除了要实现 `ISelection` 接口外，还需要对其进行封装成一个 `IStructuredSelection`。这个比较简单，直接调用 `StructuredSelection` 构造函数即可。

下面我们来具体进行一个简单得实现，这样更能清楚地认识这个机制得应用方法。

我们首先建立一个 `ViewPart`，该 `View` 上有一个 `TableViewer`，然后我们新建一个 `ViewPart`，让他显示 `TableViewer` 上被选中的 `TreeObject` 名称。

```
public class NavigationView extends ViewPart {  
    public void createPartControl(Composite parent) {
```

```

        viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL
| SWT.V_SCROLL | SWT.BORDER);

        viewer.setContentProvider(new ViewContentProvider());

        viewer.setLabelProvider(new ViewLabelProvider());

        viewer.setInput(createDummyModel());

        this.getSite().setSelectionProvider(viewer);
    }
}

```

我们在创建这个 NavigationView 得 TreeViewer 时，将它加入到了 Site 得 SelectionProvider 中，此时它所发出的 selection 事件就可以扩散给其他监听器。

```

public class View extends ViewPart implements ISelectionListener
{
    .....

    public void createPartControl(Composite parent) {

        this.getSite().getPage().addSelectionListener(this);

        Composite top = new Composite(parent, SWT.NONE);

        GridLayout layout = new GridLayout();

        layout.marginHeight = 0;

        layout.marginWidth = 0;

        top.setLayout(layout);

        // top banner

        Composite banner = new Composite(top, SWT.NONE);

        banner.setLayoutData(new
GridData(GridData.HORIZONTAL_ALIGN_FILL,
        GridData.VERTICAL_ALIGN_BEGINNING, true, false));

        layout = new GridLayout();

        layout.marginHeight = 5;
    }
}

```

```

        layout.marginWidth = 10;

        layout.numColumns = 2;

        banner.setLayout(layout);

        // setup bold font
        Font boldFont = JFaceResources.getFontRegistry().getBold(
            JFaceResources.DEFAULT_FONT);

        l = new Label(banner, SWT.WRAP);
        l.setText("Subject:");
        l.setFont(boldFont);
        l = new Label(banner, SWT.WRAP);

        // message contents
    }

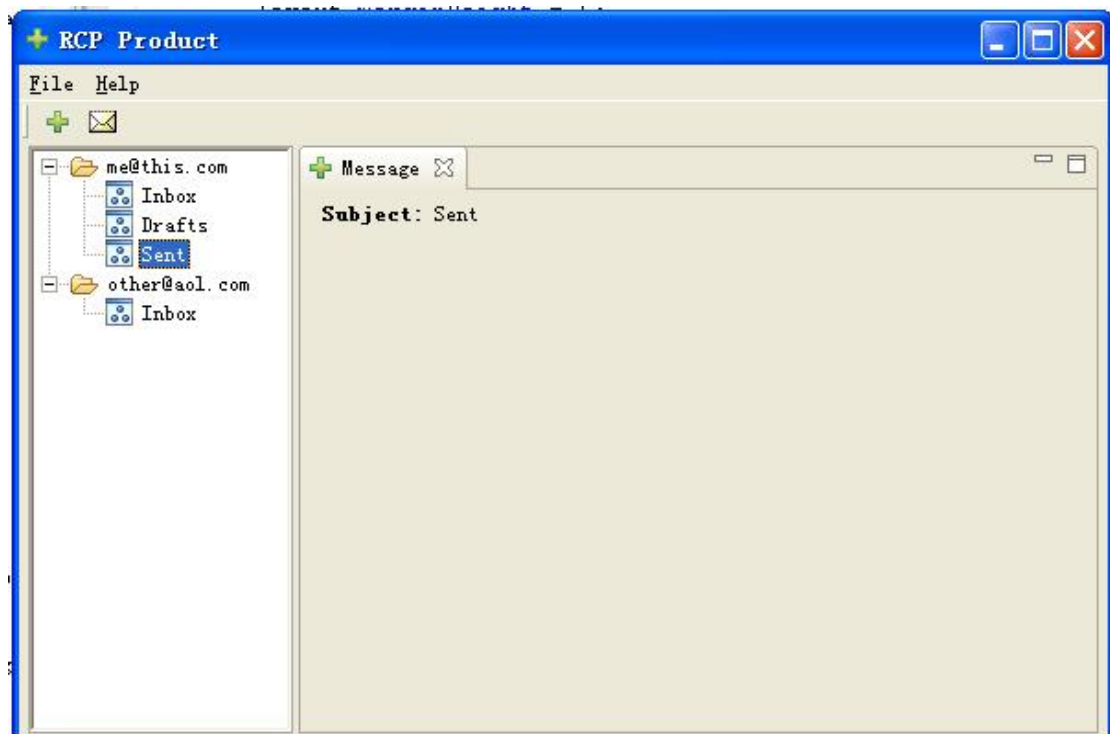
    public void selectionChanged(IWorkbenchPart part, ISelection
selection) {
        NavigationView.TreeObject object =
(NavigationView.TreeObject)((IStructuredSelection) selection)
            .getFirstElement();

        l.setText(object.getName());
    }
}

```

View 类实现了 ISelectionListener 接口，我们在 createControl 得时候把它注册到了 Page 的 SelectionChangeListener 中。

在 ISelectionListener 中，我们每当监听到 selection 改变后，就会把这个 selection 所携带的 TreeObject 的 name 属性现实在 Label 上：



2. Eclipse RCP 中的线程

2.1 RCP GUI 线程简介

各个操作系统之间的线程机制是不一样的,但是大致是相同的.当用户使用 GUI 程序时,如果点鼠标或按下键盘上的键等时,操作系统会产生对应的 GUI 事件,它来决定哪个窗口或程序来接受每一个事件并且放到程序的事件队列中.

任何 GUI 程序的底层结构就是一个事件循环.程序首先初始化事件循环,并开始循环,这个循环会从事件队列依次接收 GUI 事件并一一做出相应的反应.程序应该对事件做出快速的反应使程序一直对用户有响应,举个例子,用户点了一下程序里的一个按钮结果程序没有反应了,说明这个程序在控制 UI 线程上存在缺陷。

如果某个 UI 事件引发了某个需要长时间的事务,那么应该把它放到一个另外的单独的线程中,这样程序的那个事件循环就能够马上回来响应用户的下一个操作

eclipse 为我们开发插件提供了一个方便的 UI 线程包,大大的简化了很多底层复杂的东西。

2.2 SWT UI 线程

SWT用的是操作系统直接支持的线程模式,程序会在主程序里运行一个时间循环并依次在这个线程里响应事件.看下面这段代码,UI 线程就是我们创建的 Display 的这个线程.

```
public static void main (String [] args) {  
    Display display = new Display ();  
    Shell shell = new Shell (display);  
    shell.open ();  
    // 开始事件循环  
    // 关掉窗口后  
    while (!shell.isDisposed ()) {  
        if (!display.readAndDispatch ())  
            display.sleep ();  
    }  
    display.dispose ();  
}
```

如果是长时间的操作,最好不要用 UI 线程来做一些业务上的逻辑,这样给用户的感觉是不好的,经常会出现 UI 由于等待一些业务逻辑代码执行,而造成“假死”现象。

2.3 Job

如果需要用到多线程去处理一些非 UI 的逻辑,可以交给 Job 去做.它其实就是另外启动的线程。

Job 类由 org.eclipse.core.runtime 插件提供.它能够让客户程序员轻松的在另外的线程中执行代码.

看一个小例子

```

Job job = new Job("My First Job") {
    protected IStatus run(IProgressMonitor monitor) {
        System.out.println("Hello World (from a background job)");
        return Status.OK_STATUS;
    }
};
job.setPriority(Job.SHORT);
job.schedule(); // start as soon as possible

```

Job 的默认优先级是 Job.Long, 这里例子中的优先级要比它高.

只要调用 Job#schedule(), 它就会尽快在另外的线程中运行 run() 中的代码.

再看一个小例子:

```

final Job job = new Job("Long Running Job") {
    protected IStatus run(IProgressMonitor monitor) {
        try {
            while(hasMoreWorkToDo()) {
                // do some work
                // ...
                if (monitor.isCanceled()) return Status.CANCEL_STATUS;
            }
            return Status.OK_STATUS;
        } finally {
            schedule(60000); // start again in an hour
        }
    }
};
job.addJobChangeListener(new JobChangeAdapter() {
    public void done(IJobChangeEvent event) {

```

```

        if (event.getResult().isOk())
            postMessage("Job completed successfully");
        else
            postError("Job did not complete successfully");
    }
});
job.setSystem(true);

job.schedule(); // start as soon as possible

```

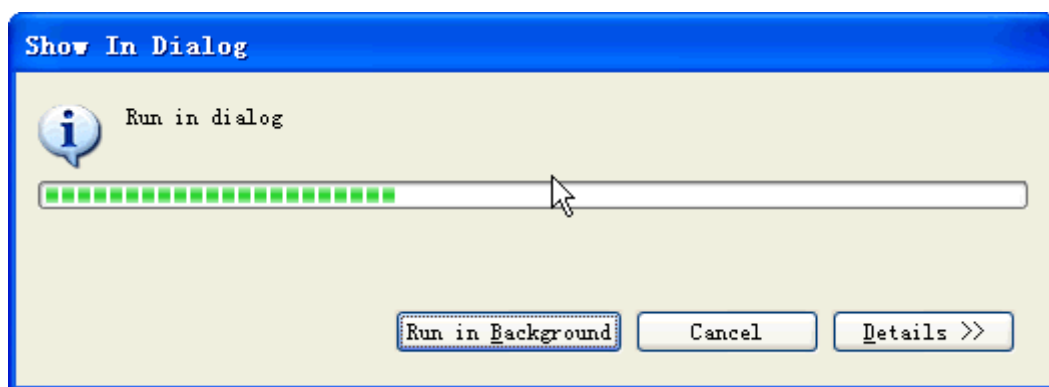
monitor 是一个进度显示条,它会在运行 job 时自动显示,如果任务成功运行完成,返回 Status.OK_STATUS, 如果中途被用户在进度显示条那里中断,就返回 Status.CANCEL_STATUS.上面 schedule(60000);它是让 job 每过 1 小时就自动运行,Job 又一个非常强大的功能.

然后后面是可以给 job 添加监听器.

job.setSystem(true);这一句是把这个 job 设置为系统级别的.如果调用 setUser(true),那么就被定义为用户级别的,用户级别和默认级别的 job

在运行时会以 UI 形式反映出来,如果是用户 job,那么会弹出一个进度显示窗口,能让用户选择在后台里运行.

下图是一个 job 自动运行时的效果:



job 常常用到的一个方法:join().

系统调用到某个 job,调用它的 run()方法:

```

class TrivialJob extends Job {

```

```
public TrivialJob() {  
    super("Trivial Job");  
}  
  
public IStatus run(IProgressMonitor monitor) {  
    System.out.println("This is a job");  
    return Status.OK_STATUS;  
}  
}
```

job 的创建和计划如下所示:

```
TrivialJob job = new TrivialJob();  
System.out.println("About to schedule a job");  
job.schedule();  
System.out.println("Finished scheduling a job");
```

他们的执行是和时间没关系的,输出可能如下:

```
About to schedule a job  
  
This is a job  
  
Finished scheduling a job
```

也可能是:

```
About to schedule a job  
  
Finished scheduling a job  
  
This is a job
```

如果希望某个 job 运行完成后在继续时,可以使用 `join()` 方法.

`join()` 会一直阻塞到该 job 运行完:

```
TrivialJob job = new TrivialJob();  
System.out.println("About to schedule a job");  
job.schedule();  
job.join();  
if (job.getResult().isOk())  
    System.out.println("Job completed with success");  
else  
    System.out.println("Job did not complete successfully");
```

上面的代码执行后,输出应该就是这样:

```
About to schedule a job  
  
This is a job  
  
Job completed with success
```

2.4 干扰 UI 线程

如果我们在非 UI 线程中去访问 UI 线程中的变量或者打断 UI 线程,就会产生异常, SWT 就会抛出一个 `SWTException` 异常。这是以为 SWT 在设计的时候是不允许非 UI 线程去打断 UI 线程的。

但是不是说在非 UI 线程中就无法去访问到 UI 线程了,比如我们在一个非 UI 线程中连接数据库,我们希望将连接的过程利用某 UI 控件(比如 `Label`)显示出来,那我们就可以利用 `Display` 提供的同步、异步方法:

- `Display#syncExec(Runnable)`
- `Display#asyncExec(Runnable)`

或者可以自己编写了另外一种 `Job`,就是 `UIJob`,可以直接在它里面运行改变 UI 的代码,其实它就是在 SWT 的 `asyncExec()` 方法里运行的。所有继承 `UIJob` 的类应。

2.5 关于进度显示

在 eclipse 插件和 RCP 开发中用户级别的 job 是互操作性最强的,它不仅能够让用户用 Cancel 键取消 job, 而且可以在 Detail 中展示具体情况,但是注意:

Detail 只会在下面两种方法中出现:

`IProgressService#busyCursorWhile` 或

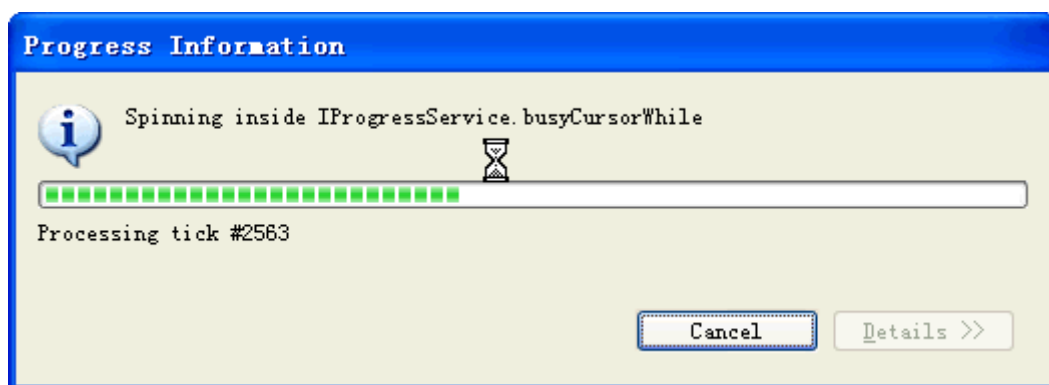
`IProgressService#runInUI`

➤ **`IProgressService#busyCursorWhile`**

注意这里的 `run()` 中做些和 UI 无关的事

```
IProgressService progressService = PlatformUI.getWorkbench().
    getProgressService();
progressService.busyCursorWhile(new IRunnableWithProgress(){
    public void run(IProgressMonitor monitor) {
        //do non-UI work
    }
});
```

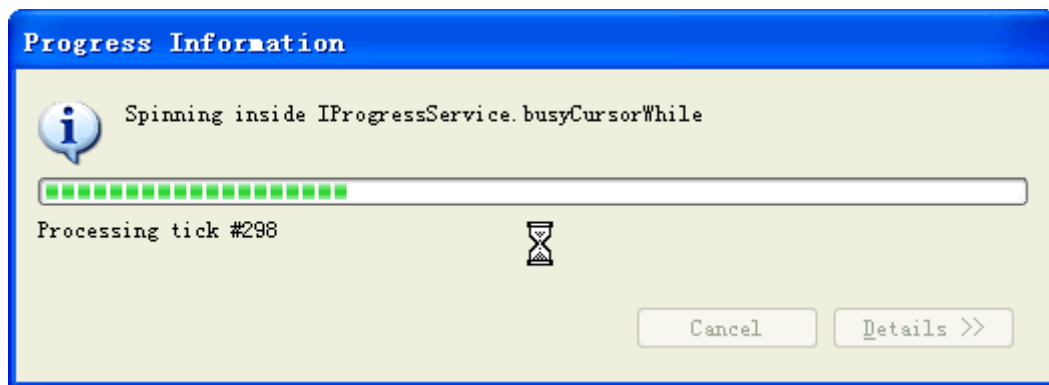
运行效果:



➤ **`IProgressService#runInUI`**

注意这里的 `run()` 中可以做些和 UI 有关的事.

```
progressService.runInUI(  
    PlatformUI.getWorkbench().getProgressService(),  
    new IRunnableWithProgress() {  
        public void run(IProgressMonitor monitor) {  
            //do UI work  
        }  
    },  
    Platform.getWorkspace().getRoot());
```



上面只是简单介绍了一下 Eclipse 中多线程以及 ProgressMonitor 的使用，一般情况下开发人员用得最多得也是上面所介绍得几种方法。

3. SWT 绘图

3.1 SWT 绘图介绍

SWT 具有一个 GC 对象，它是专门负责绘制的一个类，由于 SWT 是以调用操作系统底层 API 进行控件显示的，所以 GC 具有很强的作图能力，特别是在 Windows 平台上。

和一般的绘图类一样，GC 如同一只画笔，能够在设定好的画布上任意绘制图形。一帮情况下，我们要用 SWT 绘制一个图形，需要在某个控件对象上加载一个监听器，这个监听器能够获得 GC 对象，这个 GC 对象是针对它所在的控件的，开发人员利用这个 GC 对象就能够在该控件上绘制图形了。

下面有一段小程序，将展示一个简单的绘图过程。

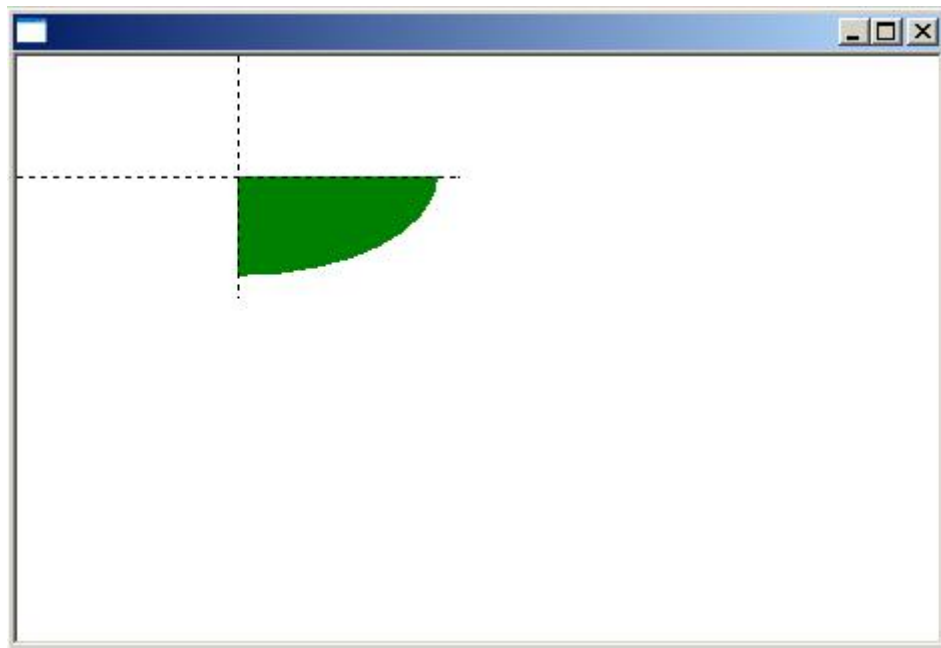
```
public class Drawings {  
    Display display = new Display();  
    Shell shell = new Shell(display);  
    public Drawings() {  
        shell.setLayout(new FillLayout());  
        Canvas canvas = new Canvas(shell, SWT.BORDER);  
        canvas.setBackground(  
display.getSystemColor(SWT.COLOR_WHITE));  
        canvas.addPaintListener(new PaintListener() {  
            public void paintControl(PaintEvent e) {  
                e.gc.setLineStyle(SWT.LINE_SOLID);  
                e.gc.setBackground(  
display.getSystemColor(SWT.COLOR_DARK_GREEN));  
                e.gc.fillArc(10, 10, 200, 100, 0, -90);  
                e.gc.setLineStyle(SWT.LINE_DOT);  
                e.gc.drawLine(0, 60, 220, 60);  
                e.gc.drawLine(110, 0, 110, 120);  
            }  
        });  
        shell.pack();  
        shell.open();  
        // Set up the event loop.  
        while (!shell.isDisposed()) {  
            if (!display.readAndDispatch()) {  
                // If no more entries in event queue  
                display.sleep();  
            }  
        }  
        display.dispose();  
    }  
}
```

```
    }  
  
    public static void main(String[] args) {  
        new Drawings();  
    }  
}
```

从上面代码看出，我们是创建了一个 Canvas 控件，然后给它安装上一个 PaintListener，这个 PaintListener 整是我们绘图所需要的监听器。

当我们创建好这个控件后，SWT 在显示的时候，就会回掉 PaintListener 的 paintControl 方法，并传入一个 PaintEvent 事件对象，而这个对象就拥有该控件的 GC 句柄。

我们运行看看结果：



运行显示出我们想要的图形。

虽然上面的例子很简单，但是 SWT 的绘图能力不仅仅如此，我们下面将介绍 SWT 的 GC 高级应用。

3.2 GC 的一些高级特性

3.2.1 异或

异或是图形绘制中比较常用的一个功能。我们这里创建一个程序，绘制三个不同颜色的圆，让他们相交，利用异或功能可以发现他们的交接处是半透明的。

```
public class XOR {
.....

    public XOR() {
        shell.setLayout(new FillLayout());

        final Canvas canvas = new Canvas(shell, SWT.NULL);
        canvas.setBackground(display.getSystemColor(SWT.COLOR_WHITE))
;

        canvas.addPaintListener(new PaintListener() {
            public void paintControl(PaintEvent e) {
                e.gc.setXORMode(true);

                e.gc.setBackground(display.getSystemColor(SWT.COLOR_GREEN
));

                e.gc.fillOval(60, 10, 100, 100); // Top
                e.gc.setBackground(display.getSystemColor(SWT.COLOR_RED))
;

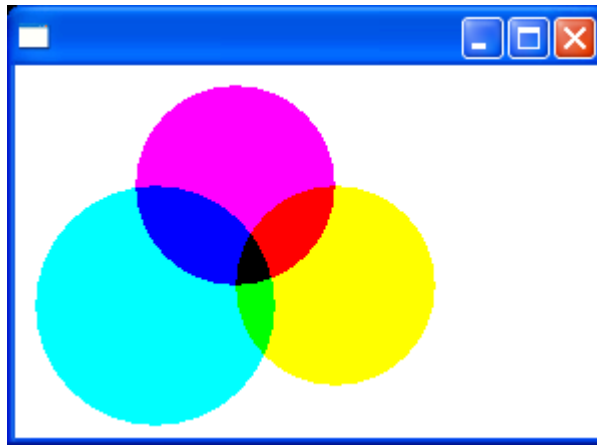
                e.gc.fillOval(10, 60, 120, 120); // left bottom
                e.gc.setBackground(display.getSystemColor(SWT.COLOR_BLUE)
);

                e.gc.fillOval(110, 60, 100, 100); // right bottom
            }
        });

.....
    }

    public static void main(String[] args) {
```

```
new XOR();  
}
```



我们注意到代码中，将 GC 的 XORMode 设置为了 true，这样它就具有异或的功能了。

3.2.2 截取屏幕

GC 能够和 Image 对象配合，截取整个屏幕。

我们看看以上代码片断：

```
button.addListener(SWT.Selection, new Listener() {  
    public void handleEvent(Event event) {  
        GC gc = new GC(display);  
        final Image image =  
new Image(display, display.getBounds());  
        gc.copyArea(image, 0, 0);  
        gc.dispose();  
  
        Shell popup = new Shell(shell, SWT.SHELL_TRIM);  
        .....  
        .....  
        Canvas canvas = new Canvas(sc, SWT.NONE);  
        .....  
        canvas.addPaintListener(new PaintListener() {  
            public void paintControl(PaintEvent e) {
```

```

        e.gc.drawImage(image, 0, 0);

    }

});

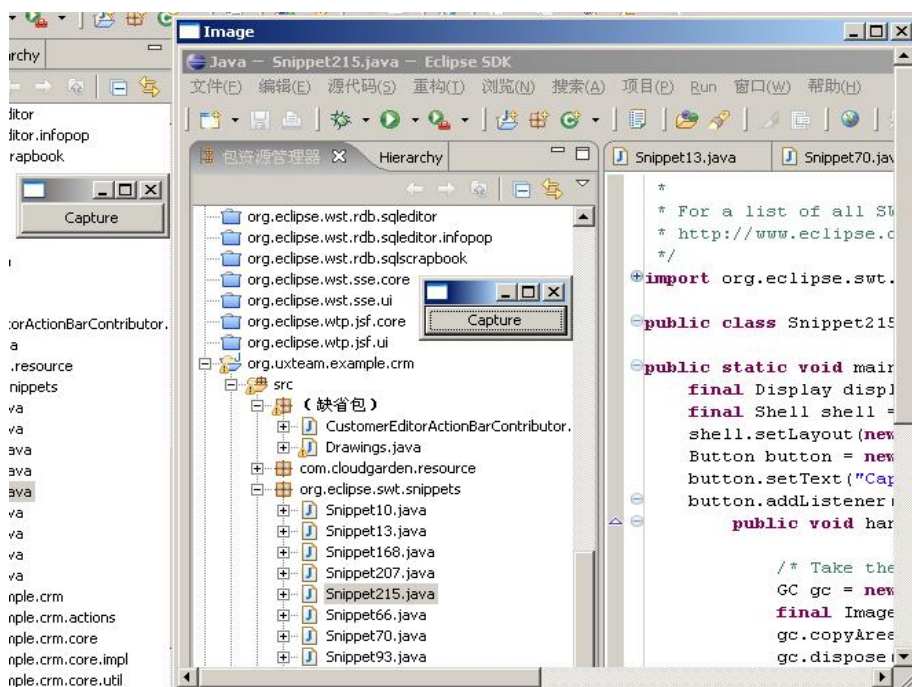
popup.open();

}

});

```

通过一个空的 Image 对象，GC 利用 copyArea 方法将 Display 记录下来，然后填充给了 Image 对象。



3.2.3 Alpha 通道和矩阵变换

GC 支持 Alpha 通道，这让我们绘制的图形具有“透明”功能，而矩阵变化也让我们的图形能够任意旋转、拉伸、倾斜。

我们看看下面的代码片断。

```

.....

GC gc = new GC(image);

gc.setBackground(display.getSystemColor(SWT.COLOR_RED));

```

```

        gc.fillOval(rect.x, rect.y, rect.width, rect.height);
        gc.dispose();

        shell.addListener(SWT.Paint, new Listener() {

            public void handleEvent(Event event) {

                GC gc = event.gc;

                Transform tr = new Transform(display);

                tr.translate(50, 120);

                tr.rotate(-30);

                gc.drawImage(image, 0, 0, rect.width, rect.height, 0,
0, rect.width / 2, rect.height / 2);

                gc.setAlpha(100);

                gc.setTransform(tr);

                Path path = new Path(display);

                path.addString("SWT", 0, 0, font);

                gc.setBackground(display.getSystemColor(SWT.COLOR_GREEN));

                gc.setForeground(display.getSystemColor(SWT.COLOR_BLUE));

                gc.fillPath(path);

                gc.drawPath(path);

                tr.dispose();

                path.dispose();

            }

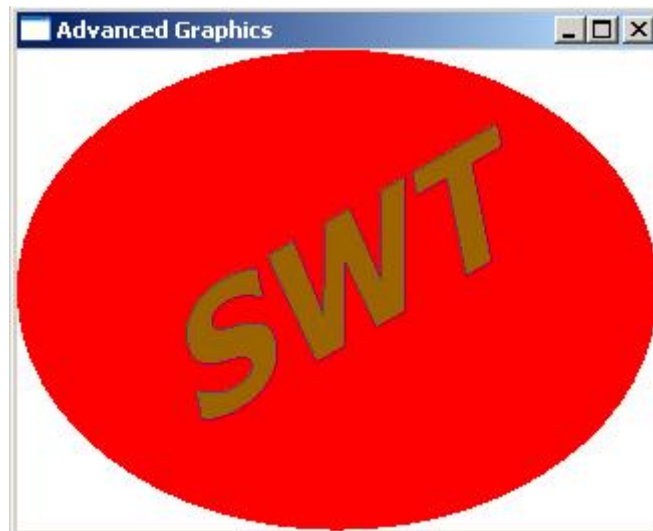
        });

        .....
    }

```

首先我们创建了一个 GC，然后我们创建了一个 Transform 对象，这个对象是用于变换矩阵用的，它具有位移、旋转等方法，上面的代码中，我们将这个 Transform 移动到(50, 120)处，并旋转了 30 度。

而 GC 可以设定一个 Alpha 值，值为 100，这就说明我们绘制的图形将会是一个半透明的图形。



3.3 一个简单的报表柱型图

我们回到 CRM RCP 中，如果我们想要为用户开发一个简单的预览报表的功能的话，就可以采用 GC 来绘制我们的图形。

假设我们将用户的客户数量来进行统计，绘制一个基于 xy 轴的柱型图，x 是用户，而 y 表示的是当前用户所用友的客户数量。

首先我们要绘制出这个 xy 的坐标图：

```
public void paintControl(PaintEvent e) {  
    GC gc = e.gc;  
    int zx = canvas.getBounds().x + 20;  
    int zy =  
canvas.getBounds().y  
+ canvas.getBounds().height - 20;  
    gc.drawLine(zx,zy,zx,canvas.getBounds().y);  
    gc.drawLine(zx,zy,zx+  
canvas.getBounds().width - 40,zy);  
    gc.dispose ();  
}
```

这个程序很简单，下面是效果截图：



现在我们考虑如何绘制一个柱状图在 xy 坐标上。

首先我们衡量一下 y 轴的长度，约定每一个像素代表一个用户的客户数量，而 x 轴的长度，每 20 个像素表示一个 CRM 用户，这样我们就可以预先得知，这个柱状图形的宽度是 20 个像素点。

我们假设现在有一个 CRM 用户，他具有的客户数量为 100，我们可以绘制出一个柱状图：

```
Model model = new Model();

        model.setCustomerNum(100);

        gc.drawRectangle(
zx,zy- model.getCustomerNum(),
20,model.getCustomerNum());

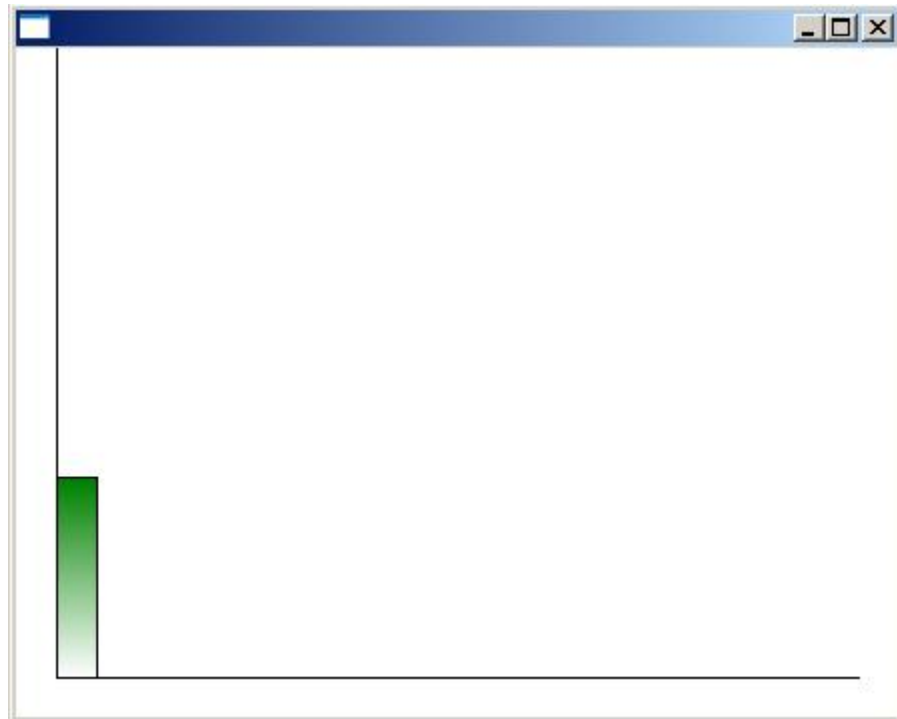
        gc.setForeground(new Color(null,0,128,0));

        gc.fillGradientRectangle(zx + 1
,zy-
model.getCustomerNum() + 1,
20-1,
model.getCustomerNum()-1,true);
```

我们这里是假设了一个用户，然后利用用户的客户数量数据绘制了一个矩形，并且利用

渐变的颜色将它填充——渐变填充也是 GC 的一个特定，但是这只是一些方法的封装，和本身的绘制能力并无太大关系。

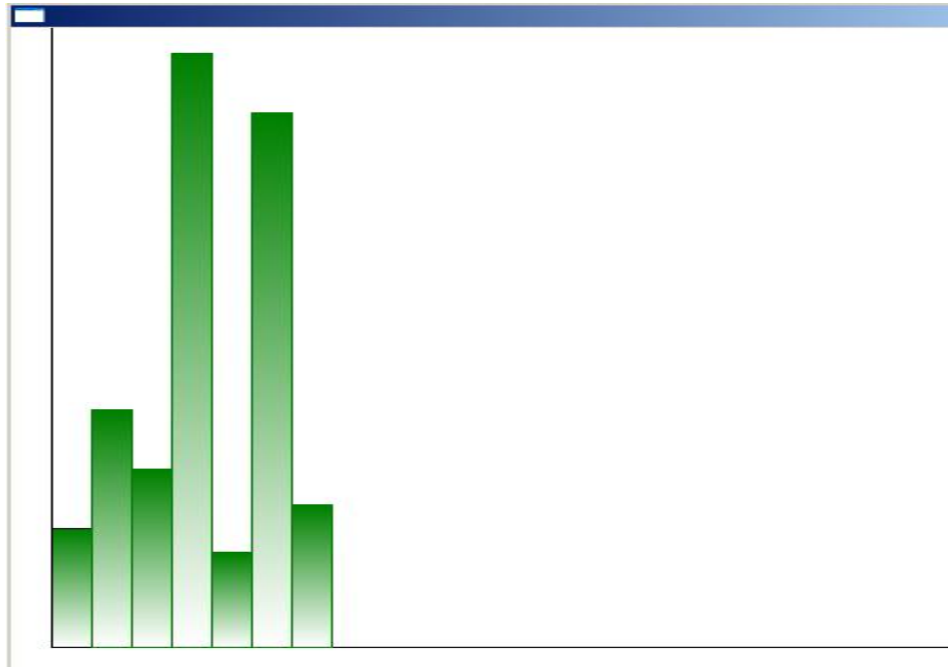
下面是我们的程序示例截图：



我们改进一下上面的绘图程序：

```
for(int i =0;i<modelList.size();i++){  
    Model model = (Model) modelList.get(i);  
    gc.drawRectangle(zx,  
        zy- model.getCustomerNum(),  
        20,model.getCustomerNum());  
    gc.setForeground(new Color(null,0,128,0));  
    gc.fillGradientRectangle(zx + 1,  
        zy- model.getCustomerNum() + 1,  
        20-1,model.getCustomerNum()-1,true);  
    zx = zx + 20;  
}
```

上面代码可以看出，我们是循环地构建这个图形。下面是我们的截图，效果和专业的 Chart 图形很像：



4. PropertySheetPage 的使用

PropertySheetPage 就是我们常见的 Eclipse 所提供的属性页，它提供较强的交互性，特别针对对象属性的编辑更为突出，而且由于 PropertySheetPage 的设计时，将 UI 和模型的解耦做得相当不错，所以在大部分情况下，如果对于模型的属性需要查看或者编辑的时候，常常会使用 PropertySheetPage。

PropertySheetPage 其实是作为 PropertyViewer 的 Control 存在的。PropertyViewer 它利用 Eclipse 平台提供的接口查询（IAdaptable，我们将会在以后的章节中讲述）的方式，获得当前激活页或者编辑器所返回的不同的 PropertySheetPage，通常情况下，开发人员是会自己去个性化设置自己的 PropertySheetPage 的。

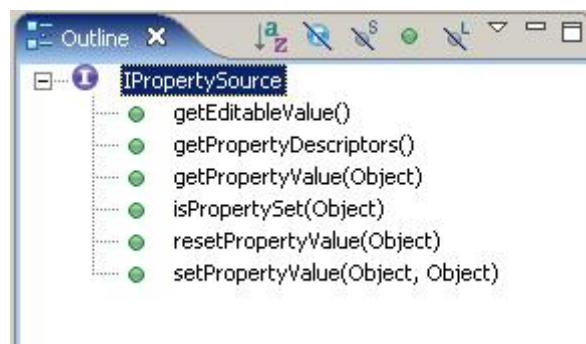
我们接下来将会讲述如何去实现我们 CRM 中 CustomerViewPart 对应的属性页的。

4.1 属性页工作机制

PropertySheetPage 类继承了 ISelectionListener，正如我们在上面章节中所讲的 Selection Service 那样，PropertySheetPage 可以监听当前 Workbench 中激活页和编辑器的 Selection 的改变，这就是为什么，我们在选择了一个 Java 类后，PropertySheetPage 将会显示出该类的一些文件信息属性。

但是这里存在一个问题，`ICollectionListener` 虽然能监听到 `Selection` 的变化，可获得的 `Selection` 很有可能是它自己根本不知道的类型，比如我们现在返回我们的 `Customer` 对象给 `PropertySheetPage`，它怎么去处理呢？

`PropertySheetPage` 利用了 `Entry` 概念，它把这些输入进来的 `Selection`，通过自己内部的一些接口进行了转化，将这些 `Selection` 携带的元素转变成一个一个的 `IPropertySource`：



而这些 `IPropertySource` 的接口方法正是 `PropertySheetPage` 所需要的方法，它接口方法返回的值提交给 UI，让 UI 显示；同时，当我们选中 UI 上的属性控件时，`IPropertySource` 会将可编辑的值提交给 UI。

如何才能让这些 `Selection` 转变成 `IPropertySource` 呢？

`PropertySheetPage` 具有一个名为 `IPropertySourceProvider` 的接口，这个接口有一个接口方法 `getPropertySource`，它通过输入的参数不同，将返回不同的 `IPropertySource` 给 `PropertySheetPage`，这样就使得 `PropertySheetPage` 能够辨识这些不同的 `Selection` 了。但是这个 `IPropertySourceProvider` 需要开发者自己去实现。

4.2 实现一个 `Customer` 的 `IPropertySource`

`IPropertySource` 接口是专门为 `PropertySheetPage` 提供 UI 所需要元素的接口类，所以我们要想让 `Customer` 对象能显示在 `PropertySheetPage` 上，那我们就必须要去实现这个 `IPropertySource`：

```
public class CustomerPropertySource implements IPropertySource {
```

```

        private Customer customer;

        private TextPropertyDescriptor namePD =
new TextPropertyDescriptor("name", "姓名");

        private ComboBoxPropertyDescriptor sexPD =
new ComboBoxPropertyDescriptor("sex",
"性别", new String[]{"男", "女"});

        private TextPropertyDescriptor postPD
= new TextPropertyDescriptor("post", "职位");

        private TextPropertyDescriptor datePD =
new TextPropertyDescriptor("date", "出生日期");

        private IPropertyDescriptor[] pds = new IPropertyDescriptor[]{
            namePD, sexPD, postPD, datePD
        };

        public Object getEditableValue() {
            return customer;
        }

        public IPropertyDescriptor[] getPropertyDescriptors() {
            return pds;
        }

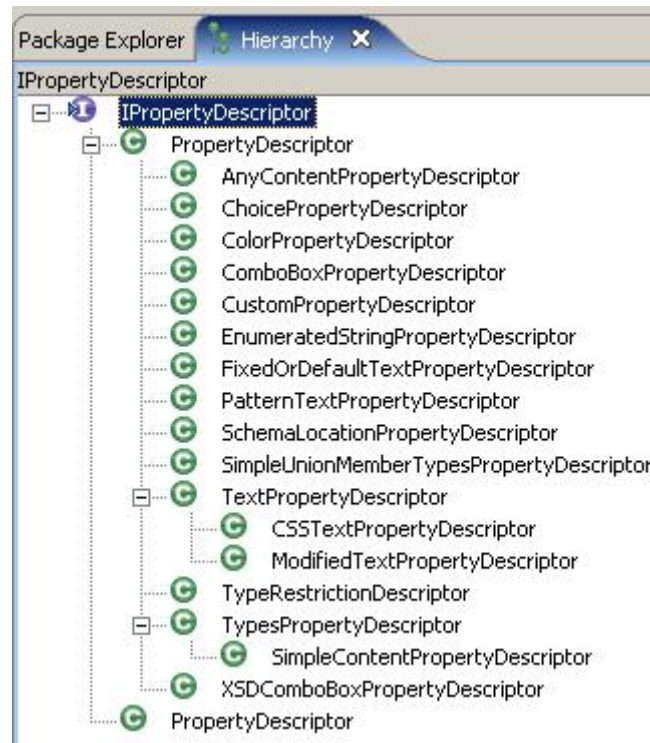
        .....
    }

```

我们注意一下上面的代码，这里我们有一个 IPropertyDescriptor 数组对象，这个

数组对象是什么呢？

`IPropertyDescriptor` 是我们在编辑属性时，显示属性的 UI 显示的接口类，我们可以看看 `IPropertyDescriptor` 接口类的实现类有哪些：



在这些实现类中，常常使用的有下面这几种：

➤ **TextPropertyDescriptor**

显示简单的文本属性的 Descriptor，它有一个文本控件组成的，修改属性的返回值为一个 String 类型。

➤ **ComboBoxPropertyDescriptor**

显示下拉框的一个 Descriptor，多数用在那种有选择约束的属性中，属性修改返回值为 Integer 类型。

➤ **ChoicePropertyDescriptor**

显示一个 CheckButton 控件，经常用在选择 Bool 类型属性，返回值为 Boolean 对象。

我们现在看看 Customer 中的字段属性：

➤ 姓名：姓名可以利用简单的 `TextPropertyDescriptor` 来作为属性的 `PropertyDescriptor`。

➤ 性别：性别可以利用 `ComboBoxPropertyDescriptor`

➤ 出生日期： `TextPropertyDescriptor`

➤ 职位: TextPropertyDescriptor

所以我们在代码中为这四个属性生成了对应的 PropertyDescriptor。

注意一下: 这些 PropertyDescriptor 的构造函数中, 第一个参数是他们所对应的 ID 值, 这个 ID 值是会在修改、查看属性值的时候常常用到的。

我们接着实现 IPropertySource 接口的接口方法:

```
public Object getPropertyValue(Object id) {  
    if(id.equals("name")){  
        return customer.getName();  
    }  
    if(id.equals("sex")){  
        if(customer.getSex().equals("男")) return new  
Integer(0);  
        if(customer.getSex().equals("女")) return new  
Integer(1);  
        return new Integer(0);  
    }  
    if(id.equals("post")){  
        return customer.getPost();  
    }  
    if(id.equals("date")){  
        return customer.getBornDate().toGMTString();  
    }  
    return null;  
}
```

上面这段代码表示的是: 通过不同的 Descriptor 的 ID, 返回这些 Descriptor 对应的值。这里可以看出来, 我们在为 sexDP 返回的一个 Integer 类型的值, 其他的三个都返回的是 String 类型的值。

```
public void setPropertyValue(Object id, Object value) {  
    if(id.equals("name")){
```

```

        customer.setName(value.toString());
    }
    if(id.equals("sex")){
        if(((Integer)value).intValue() == 0){
            customer.setSex("男");
        }
        if(((Integer)value).intValue() == 1){
            customer.setSex("女");
        }
    }
    if(id.equals("post")){
        customer.setPost(value.toString());
    }
    if(id.equals("date")){
        customer.setBornDate(new Date(value.toString()));
    }
}

```

这段代码则是当我们修改属性控件中值的时候，Descriptor 将会把这些修改后的值返回回来，那我们就可以根据 Descriptor 的 ID 来修改对应属性的值了。

4.3 修改 CustomerViewParg 的 getAdapter 方法

我们现在已经有了我们的 IPropertyDescriptor，那现在我们就需要实现自己的一个 IPropertySourceProvider 类了，这个类是为了让 PropertySheetPage 得到对应的 IPropertySource 而创建的，我们先看代码：

```

class MyPropertySourceProvider
implements IPropertySourceProvider{
    public IPropertySource getPropertySource(Object object) {
        if(object instanceof Customer){
            CustomerPropertySource cp =

```

```

new CustomerPropertySource();

        cp.setCustomer((Customer)object);

        return cp;

    }

    return null;

}

}

```

代码很简单，通过对输入的参数进行判断，返回一个我们刚才新建的 CustomerPropertySource 对象即可。

而现在我们就需要修改 getAdapter 方法：

```

public Object getAdapter(Class type){

    if(type == IPropertySheetPage.class){

        PropertySheetPage page = new PropertySheetPage();

        page.setPropertySourceProvider(

new MyPropertySourceProvider());

        return page;

    }

    return super.getAdapter(type);

}

```

这个方法是 Eclipse 中的一个接口查询的方法，PropertyView 在激活的情况下会对当前被激活的页面进行接口查询，如果该页面的 getAdapter 对 IPropertySheetPage 类型有相应的类的话，那 PropertyView 就会更改当前它自身的 PropertySheetPage。

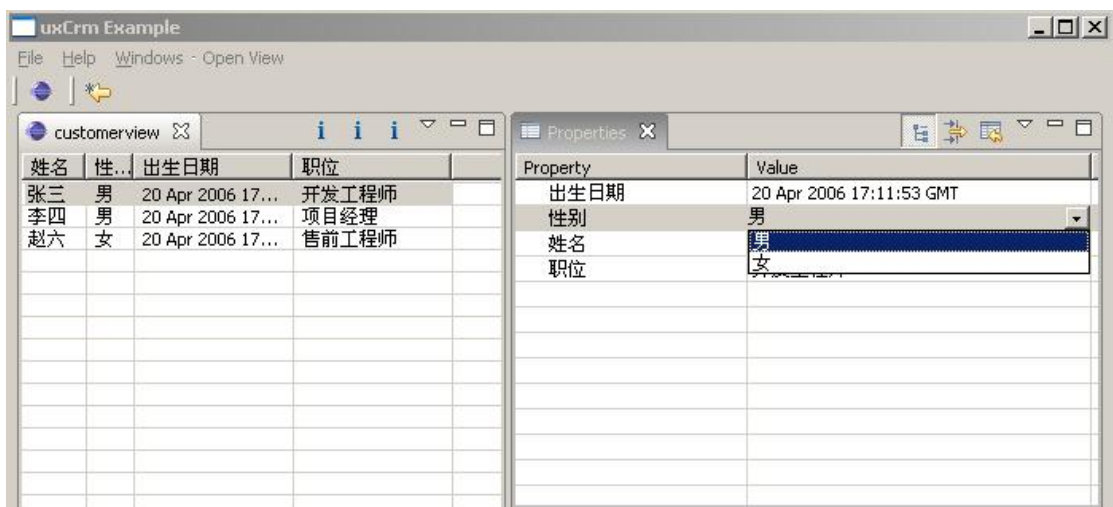
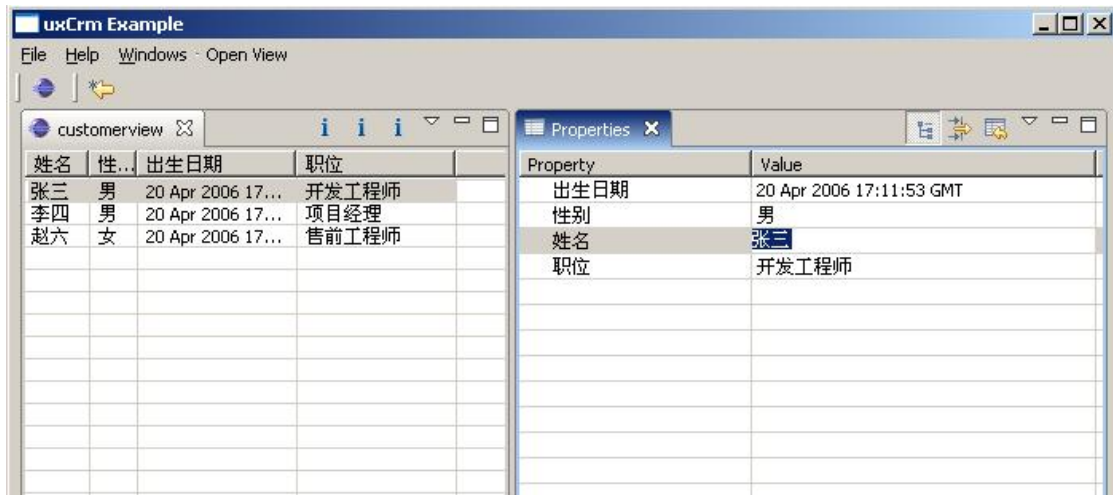
我们在这里新建了一个 PropertySheetPage 对象，并且将它的 PropertySourceProvider 替换成了我们刚才创建的 PropertySourceProvider。

这里需要注意：虽然我们已经把这些准备工作做好了，但是还是要核实一下，我们是不是应该把 SelectionProvider 设置好了呢？也就是我们在前面章节所讲的那样，需要对

Site 进行设置:

```
getSite().setSelectionProvider(viewer);
```

OK, 让我们看看运行的结果:



4.4 另一种实现方法

其实 `PropertySheetPage` 获得 `IPropertySource` 的方法不仅仅在于 `IPropertySourceProvider` 所返回的, 我们可以之间让我们的选择元素去实现 `IPropertySource` 接口, 因为 `PropertySheetPage` 会对 `Selection` 的元素进行判断, 如果它已经是 `IPropertySource` 类型的了, 那就之间使用它。

我们修改一下我们的 Customer 代码，让他实现 IPropertySource 接口，增加以下的这些方法，这些方法和我们刚才所写的差不多：

```
private TextPropertyDescriptor namePD =
new TextPropertyDescriptor("name",
                                "姓名");

private ComboBoxPropertyDescriptor sexPD =
new ComboBoxPropertyDescriptor(
    "sex", "性别", new String[] { "男", "女" });

private TextPropertyDescriptor postPD =
new TextPropertyDescriptor("post",
                                "职位");

private TextPropertyDescriptor datePD =
new TextPropertyDescriptor("date",
                                "出生日期");

private IPropertyDescriptor[] pds =
new IPropertyDescriptor[] { namePD,
                                sexPD, postPD, datePD };

public IPropertyDescriptor[] getPropertyDescriptors() {
    return pds;
}

public Object getPropertyValue(Object id) {
    if(id.equals("name")){
        return this.getName();
    }
}
```

```

        if(id.equals("sex")){

            if(this.getSex().equals("男")) return new Integer(0);
            if(this.getSex().equals("女")) return new Integer(1);
            return new Integer(0);
        }
        if(id.equals("post")){

            return this.getPost();
        }
        if(id.equals("date")){

            return this.getBornDate().toGMTString();
        }
        return null;
    }

    public boolean isPropertySet(Object id) {

        return true;
    }

    public void resetPropertyValue(Object id) {

    }

    public void setPropertyValue(Object id, Object value) {

        if(id.equals("name")){

            this.setName(value.toString());
        }
        if(id.equals("sex")){

            if(((Integer)value).intValue() == 0){

                this.setSex("男");
            }
        }
    }

```

```

        if(((Integer)value).intValue() == 1){
            this.setSex("女");
        }
    }
    if(id.equals("post")){
        this.setPost(value.toString());
    }
    if(id.equals("date")){
        this.setBornDate(new Date(value.toString()));
    }
}

public Object getEditableValue() {
    return this;
}

```

然后我们修改一下 `getAdapter`，不要再使用 `IPropertySourceProvider`：

```

public Object getAdapter(Class type){
    if(type == IPropertySheetPage.class){
        PropertySheetPage page = new PropertySheetPage();
        return page;
    }
    return super.getAdapter(type);
}

```

然后我们运行。可以发现，这种方法也是可以实现属性显示的。

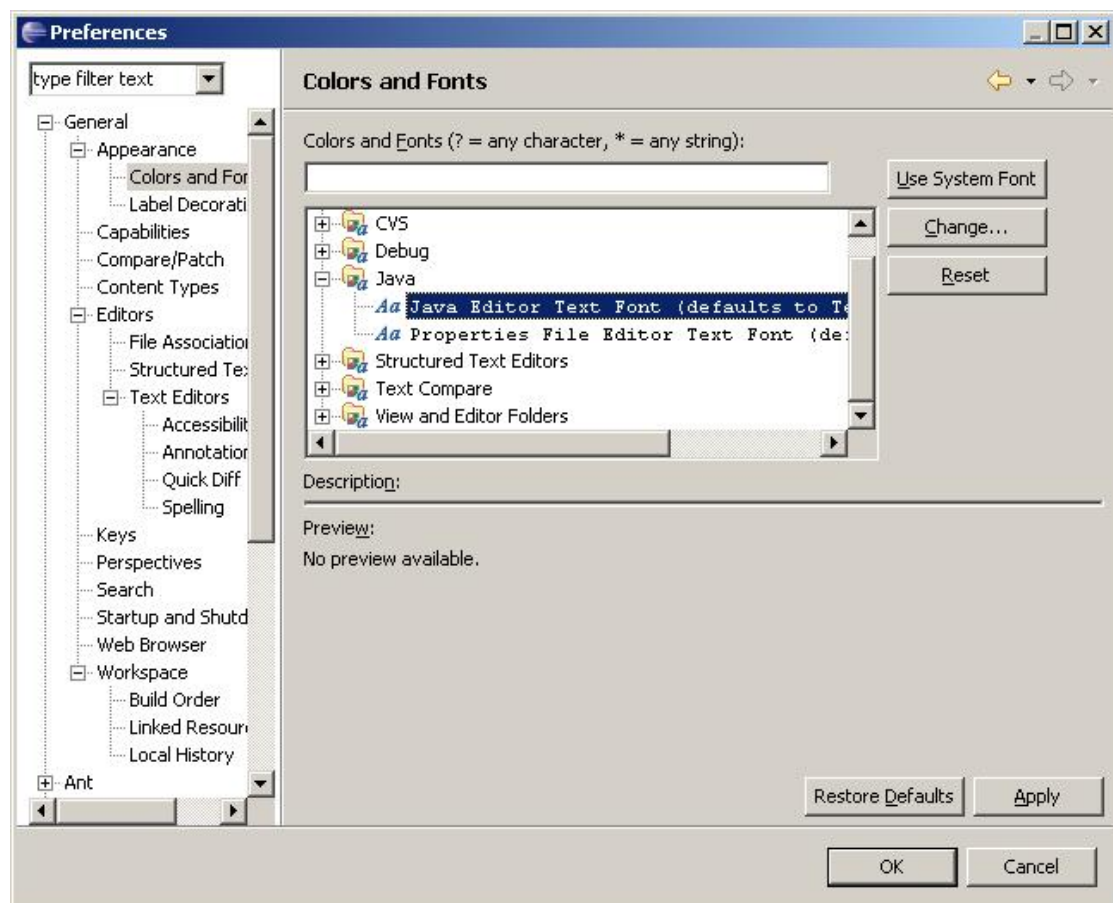
这两种方法可以任意选择一种，不过第二种方法需要模型去实现 `IPropertySource` 接口，这对已有的模型（已经不能修改的模型）来说，是无法实现的，这就需要利用

IPropertySourceProvider 来实现了，就算是模型代码可以任意修改，但是如果代码需要再 Eclipse 以外的地方使用，为了不和 Eclipse 耦合，所以实现 IPropertySource 接口也不太好。

5. PreferencePage 的使用

PreferencePage（首选项页）是在 Eclipse RCP 中比较重要的一个 UI 元素，比如我们经常需要对某一些应用程序的全局变量进行设定时，PreferencePage 的重要性就更突出了。

PreferencePage 能够对输入的一些值进行记录，保存在 Plugin 的 PreferenceStore 当中，这些的存储方式能够让用户自己定义一些需要在物理位置上记录的值，比如我们在设定 Java 代码的字体时候：



当我们改变了当前 Java 编辑器的字体显示后，每当我么启动了 Eclipse 后，Java

编辑器的字体就以 Preference 上的设定为主。

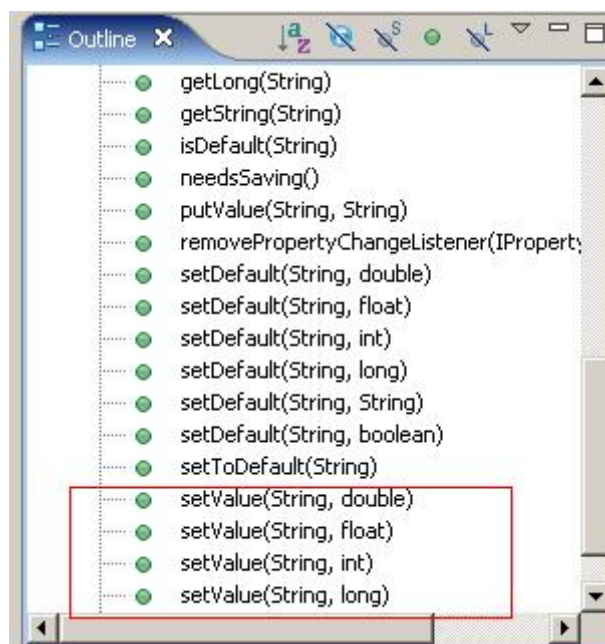
5.1 PreferenceStore

其实在 PreferencePage 上设定的值都需要记录到当前 Plugin 的 PreferenceStore 当中,而且我们要使用的时候也需要从 PreferenceStore 中取出来:

```
IPreferenceStore store =  
XXXPlugin.getDefault().getPreferenceStore();  
  
store.getString(propertyName);  
  
store.getInt(propertyName);  
  
store.getLong(propertyName);  
  
store.getBoolean(propertyName);
```

PreferenceStore 是可以存放多种类型值的,但是不能存放 java 的 Object 对象值,它能对一些基本类型: String、int、long、boolean 等值进行保存。

PreferenceStore 如同一张 HashMap,通过键值对应的方式来存放这些值,上面的代码中,propertyName 就是这个键,我们通过 PreferenceStore 的存储的方法就可以对一些值进行存放:



图中红色框所标出的方法就是 PreferenceStore 进行存储某些值的方法。

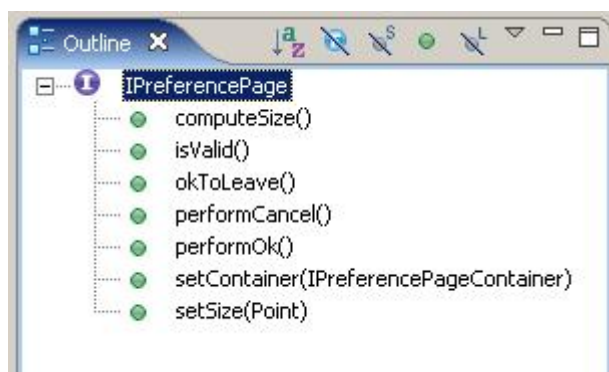
当然，PreferenceStore 也需要对一些属性值进行默认设置，可以从图中看出，setDefault 方法就是对这些默认值的设定方法。

5.2 PreferencePage

PreferencePage 就是和 PreferenceStore 进行挂钩的 UI 页面。

RCP 单独将它拿出来形成了一个扩展点：org.eclipse.ui.preferencePages。

而 PreferencePage 对应的类需要实现 IPreferencePage 接口：



当然 UI 元素方便我们一般都需要继承一些 Eclipse 已经提供好的现成的基类，所以

在制作 PreferencePage 的时候，我们都习惯于继承 FieldEditorPreferencePage，这个 Page 将 Page 上的 UI 元素都看作是一个个的 FieldEditor。

这些 FieldEditor 是经过了层层包装的，开发者可以不去注意它的细节，只需要去把他们安排在 PreferencePage 上即可，我们介绍几种 FieldEditor 的使用：

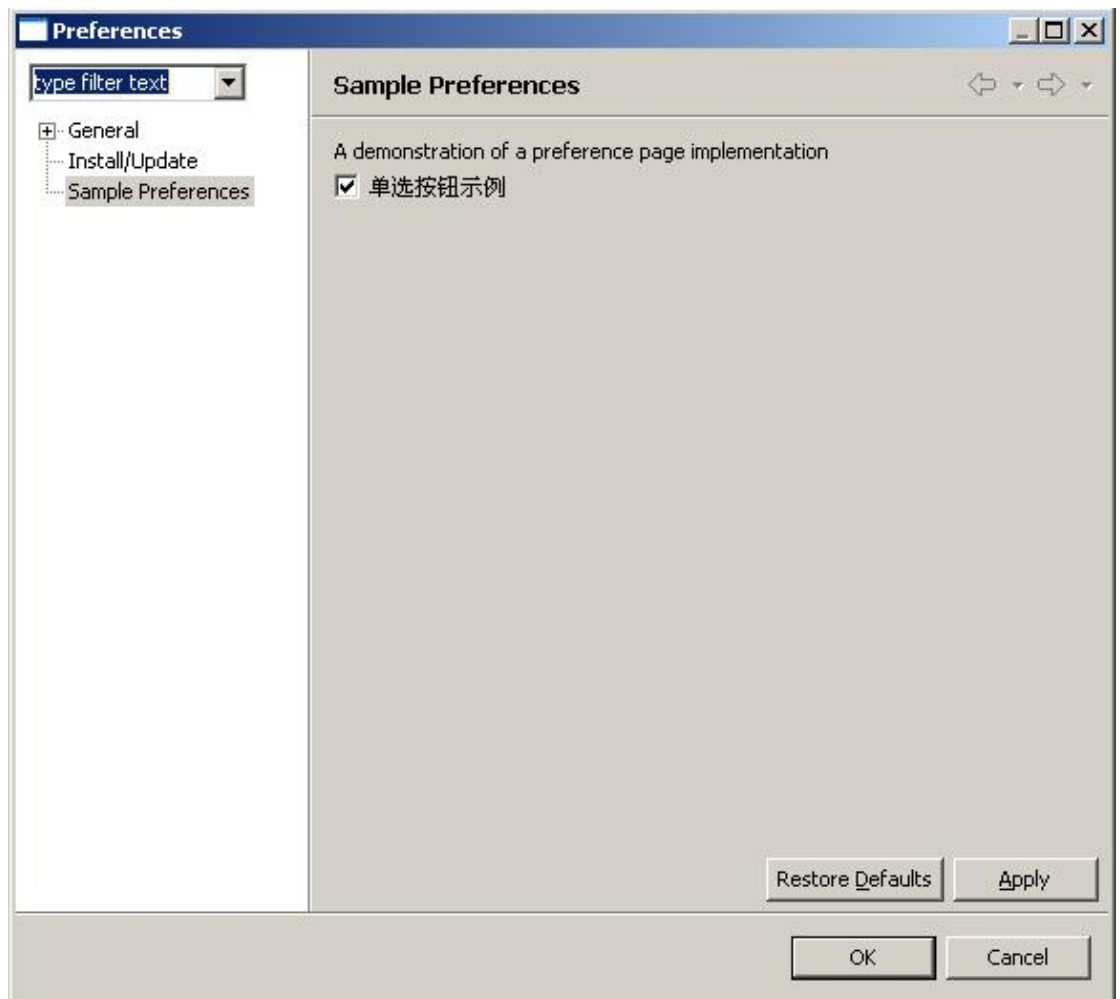
➤ DirectoryFieldEditor

可以对系统文件架进行浏览，并把用户选定的文件夹路径记录下来。



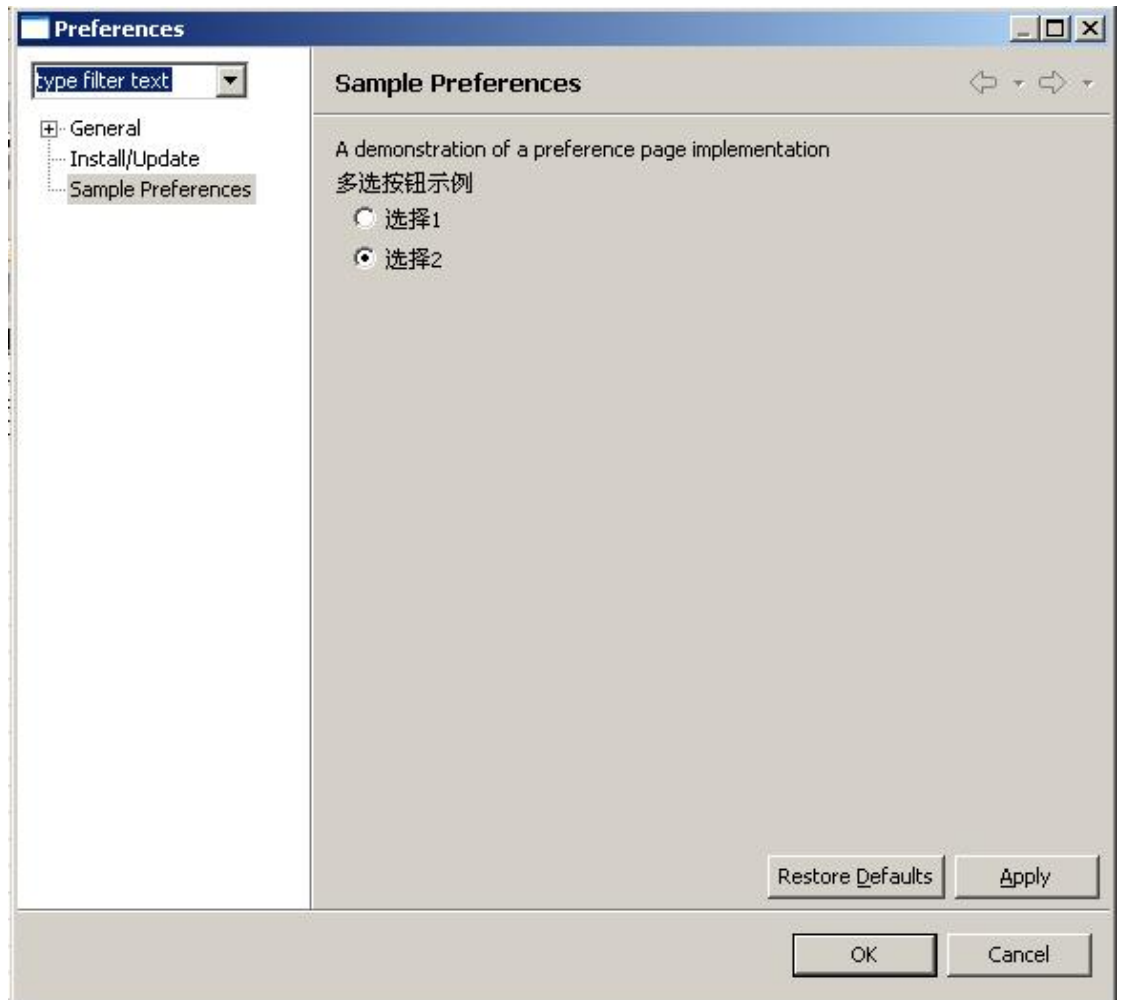
➤ BooleanFieldEditor

它将展示给用户的是一个个的 CheckBoxButton，记录当前按钮就是否被选中。一般是记录 Bool 类型值的首选。



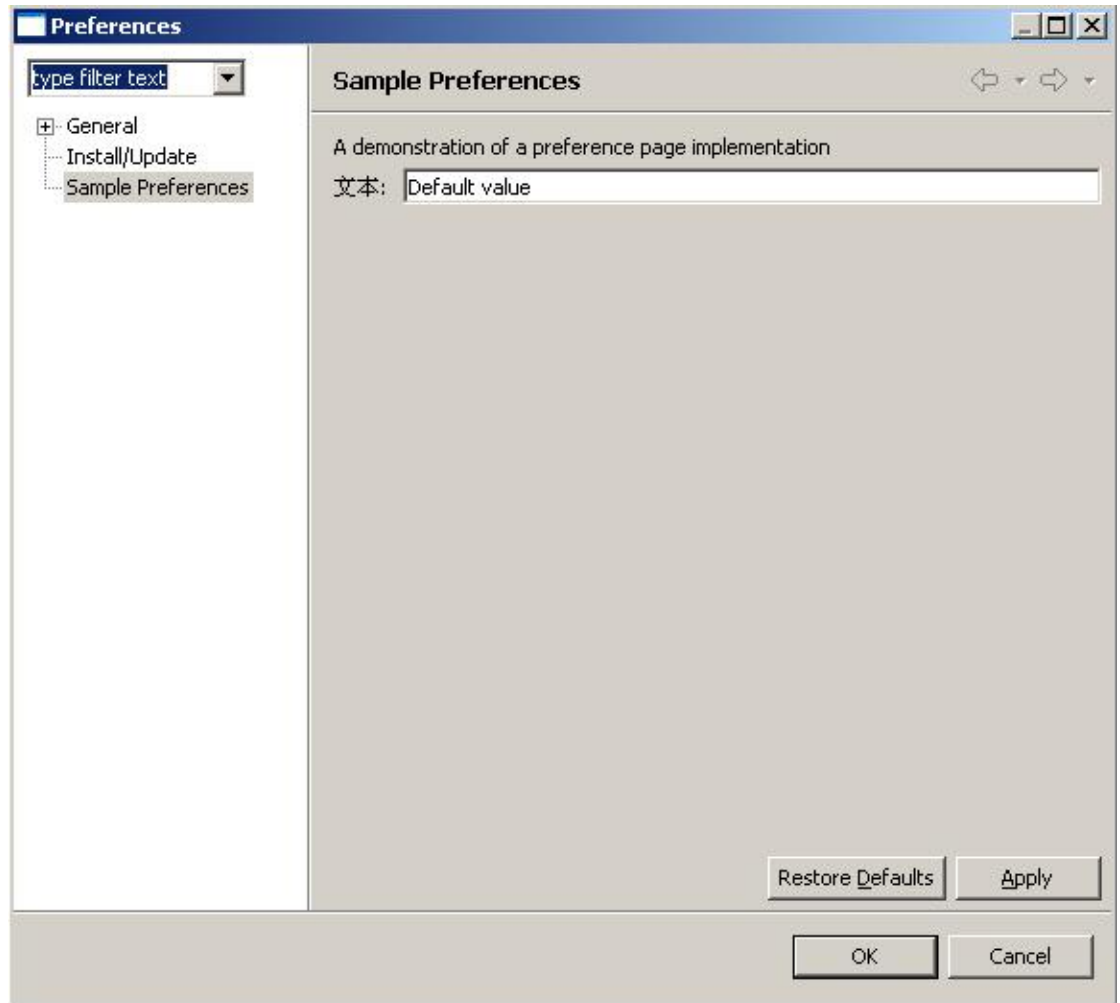
➤ **RadioGroupFieldEditor**

显示出一组 `RadioButton`。用于对可多选选项的管理。



➤ **StringFieldEditor**

显示一个文本框，专门记录 String 类之的值。



5.3 简单的 PreferencePage 例子

我们假定，设置一个简单的 Action，当点击它后会显示在 PreferenceStore 中存放的键值为“MyAction”的值。

我们首先将这个 Action 扩展点做好，然后在点击它后我们弹出一个对话框，上面显示一段文字，而这段文字是存 PreferenceStore 中取出来的。

```
public void run(IAction action) {  
    try {  
        IPreferenceStore store=  
CrmPlugin.getDefault().getPreferenceStore();  
        String displayString = store.getString("MyAction");  
    }  
}
```

```

        MessageDialog.openInformation(window.getShell(),"Info",display
String);

        } catch (Exception e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

    }
}

```

而我们现在要做的就是实现 PreferencePage 的扩展点:

```

<extension
    point="org.eclipse.ui.preferencePages">
    <page
        class="org.uxteam.example.crm.preferences.SamplePreferencePage"
        id="org.uxteam.example.crm.preferences.SamplePreferencePage"
        name="Sample Preferences"/>
    </extension>

```

这个扩展点将 preferencePage 的对应 Page 类指定到了 SamplePreferencePage 上，我们需要实现这个 Page，这个 Page 很简单，仅仅是一个简单的 StringFieldEditor 在其中，记录我们输入的文本值。

```

public class SamplePreferencePage
    extends FieldEditorPreferencePage
    implements IWorkbenchPreferencePage {

```

```

    public SamplePreferencePage() {
        super(GRID);

        setPreferenceStore(CrmPlugin.getDefault().getPreferenceStore()
    );

        setDescription("输入文本值");
    }

    public void createFieldEditors() {
        addField(
            new StringFieldEditor(PreferenceConstants.P_STRING, "
文本:", getFieldEditorParent()));
    }

    public void init(IWorkbench workbench) {
    }
}

```

我们设计完这个 PreferencePage 后好需要实现另外一个扩展点：
org.eclipse.runtime.preferences，这个扩展点是告诉 RCP，在初始化的时候将整个
Store 的值进行设置初始值。

```

<extension
    point="org.eclipse.core.runtime.preferences">
    <initializer
class="org.uxteam.example.crm.preferences.PreferenceInitializer"/>
    </extension>

```

而这个 PreferenceInitializer 类就始初始化 store 的类：

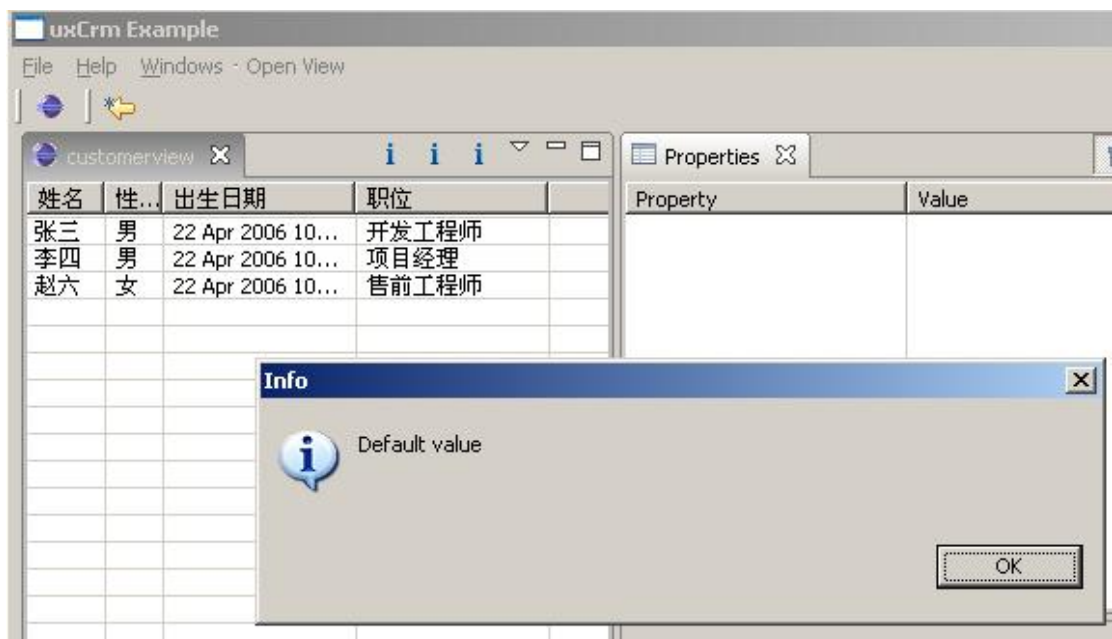
```

public class PreferenceInitializer extends
AbstractPreferenceInitializer {
    public void initializeDefaultPreferences() {
        IPreferenceStore store = CrmPlugin.getDefault()
            .getPreferenceStore();
        store.setDefault("MyAction",
            "Default value");
    }
}

```

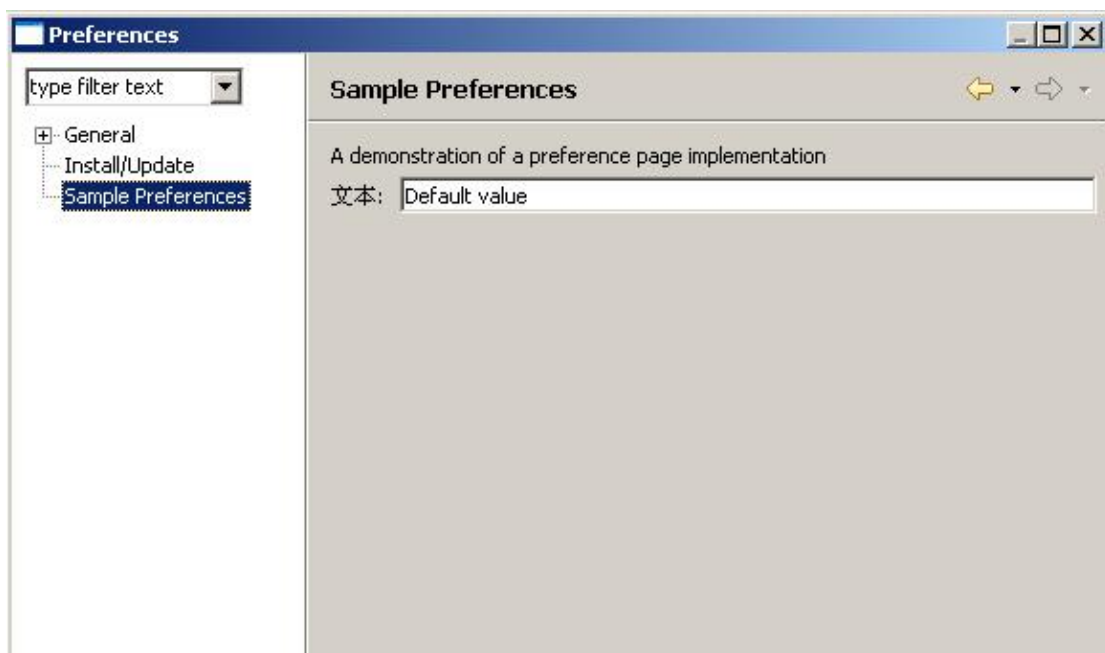
我们做完上面的工作后，运行调试一下。

我们启动后，先点击设定的按钮，它将显示：

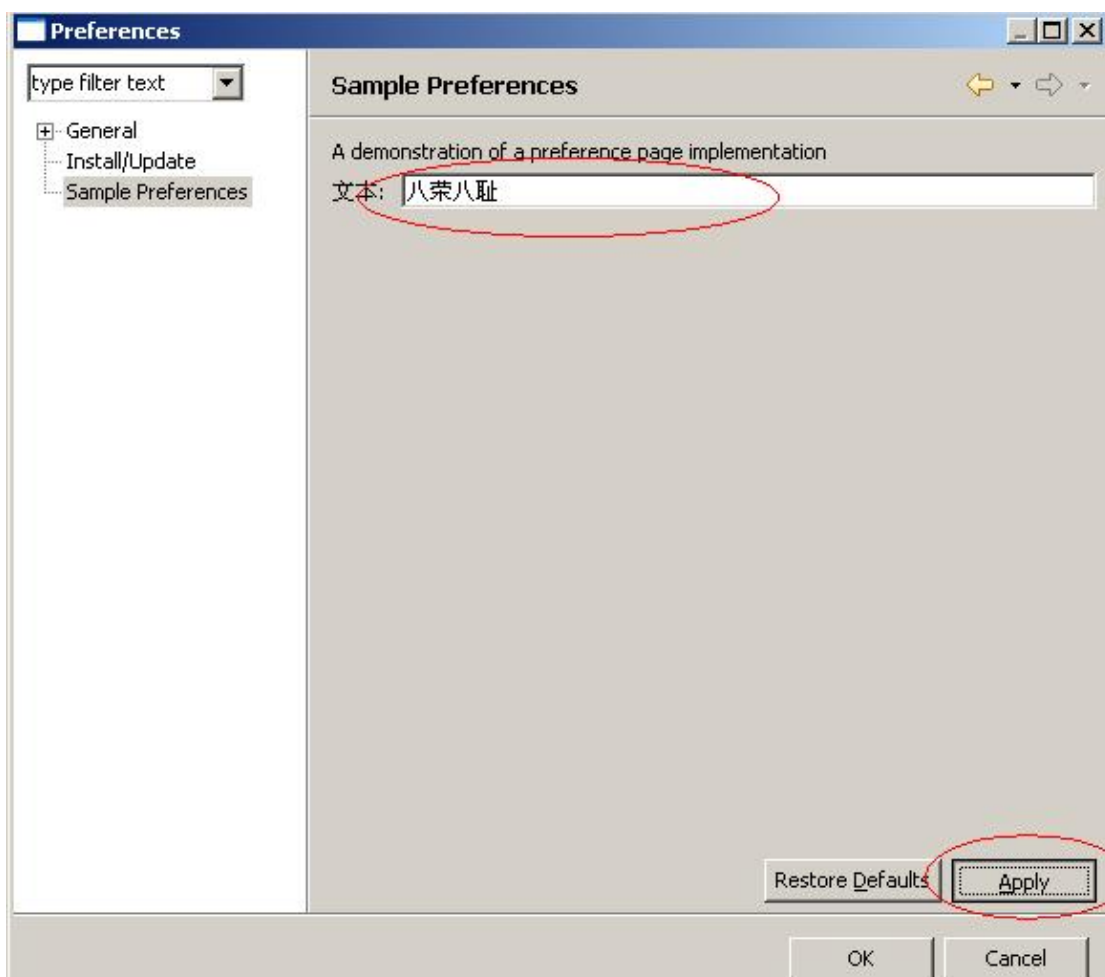


对话框上显示的始“Default value”这正是我们需要的结果，因为在初始化我们的“MyAction”值的时候，我们将默认值设置为了“Default value”，所以在启动的时候 store 里面就存放的是“Default value”。

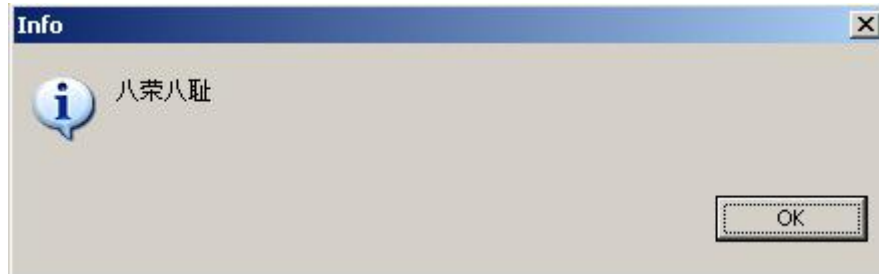
我们打开 PreferencePage:



同样，我们可以看到文本框上显示的也是 Default Value，我们现在更改它的值，并进行保存：



我么输入了文本后，点击 Apply，PreferenceStore 便将我们的这个值进行了记录。
现在再点击按钮，会发现，文本显示发生了改变：



PreferencePage 在 RCP 中专门以存储一些需要在外部文件保存的值时，比较突出，但是它只能是对一些比较简单的值进行存储，正如我们前面提到的那样，所以我们在使用 PreferencePage 进行存储的时候需要留意它所支持的类型，已经使用范围。

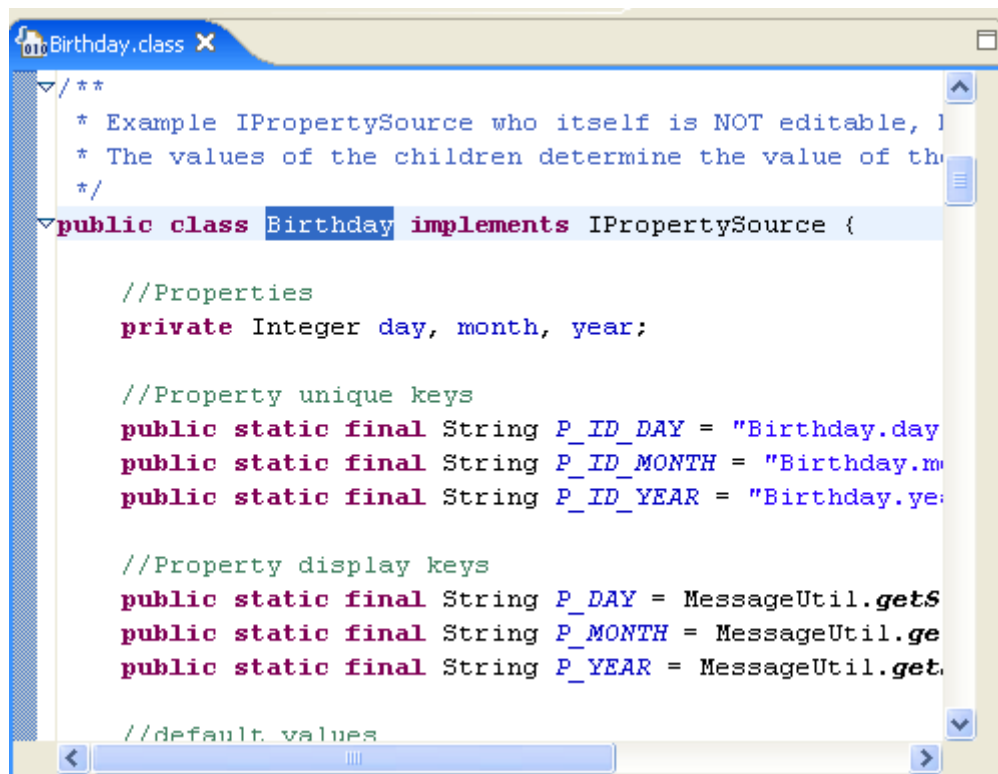
6. 文本编辑器

6.1 文本编辑器简介

我们已经了解了插件可以如何向工作台添加编辑器，但是，我们尚未了解文本编辑器的实现。

编辑器没有“典型”实现模式，原因是编辑器通常提供特定于应用程序的语义。编辑和管理特定内容类型的工具将提供定制行为来处理由资源提供的的数据。

编辑器可以具有各种形状和大小。如果插件的编辑器是基于文本的，则编辑器可以使用现有的缺省文本编辑器，也可以通过使用平台中提供的设施来创建定制的文本编辑器。Java 示例编辑器使用后一种方法。



```
BirthDay.class
/**
 * Example IPropertySource who itself is NOT editable, l
 * The values of the children determine the value of th
 */
public class Birthday implements IPropertySource {

    //Properties
    private Integer day, month, year;

    //Property unique keys
    public static final String P_ID_DAY = "Birthday.day
    public static final String P_ID_MONTH = "Birthday.m
    public static final String P_ID_YEAR = "Birthday.ye

    //Property display keys
    public static final String P_DAY = MessageUtil.getS
    public static final String P_MONTH = MessageUtil.ge
    public static final String P_YEAR = MessageUtil.get

    //default values
```

如果插件的编辑器不是基于文本的，则插件必须实现定制编辑器。可以有几种方法来构建定制编辑器，所有这些方法都取决于编辑器的外观和行为。

- 基于表单的编辑器可以类似于对话框或向导的方式来对控件进行布局。“插件开发环境”（PDE）使用此方法来构建它的清单编辑器。
- 可以使用 SWT 级别代码来编写图形增强编辑器。例如，编辑器可以创建它自己的 SWT 窗口来显示信息，或者它可以使用对应用程序优化的定制 SWT 控件。
- 面向列表的编辑器可以使用 JFace 列表、树和表查看器来处理它们的数据。

一旦已经确定了编辑器的实现模型，实现编辑器就类似于为独立的 JFace 或 SWT 应用程序编程。平台扩展用于添加支持编辑器所需要的操作、首选项和向导。但是编辑器的内部结构很大程度上依赖于应用程序设计原理和内部模型。

6.2 一个简单的 XML 编辑器实现

我们通过扩展点向导一步步为我们的 CRM 生成一个 XML 编辑器。这里我们假设，CRM 客户需要对 XML 文本进行编辑。

打开 plugin.xml，选择“扩展点”页，点击“添加”，然后选择 org.eclipse.ui.editors，在选项选择一个 XML Editor 的模板：



点击下一步后完成。

我们会看到我们的代码中多处了一些代码，这些代码就是 XML 编辑器的代码。

现在我们需要测试一下，这个编辑器是否可用。

由于在 Eclipse 中，这一类的编辑器都需要通过资源文件才能打开，但是我们的 CRM RCP 中是具备这种功能的。所以我们现在需要考虑一下如何打开这个编辑器。一般情况下，

这种编辑器是和本身的 `IEditorInput` 是有很大关系的（我们在前面已经讲过），所以我们需要给它生成一个假设的 `IEditorInput` 实例。

XML 编辑器所能识别的是一个名为 `FileEditorInput` 的 `IEditorInput`，我们在 `SampleAction` 代码中这样写：

```
try {

    IProject p = ResourcesPlugin.getWorkspace()
        .getRoot().getProject("test");

    if(!p.exists()){

        p.create(new NullProgressMonitor());

    }

    if(!p.isOpen()){

        p.open(new NullProgressMonitor());

    }

    IFile file = p.getFile("test.xml");

    if(!file.exists()){

        file.create(

new ByteArrayInputStream("").getBytes()),

false,new NullProgressMonitor());

    }

    FileEditorInput i = new FileEditorInput(file);

    window.getActivePage()

        .openEditor(i,

"org.uxteam.example.crm.editors.XMLEditor");

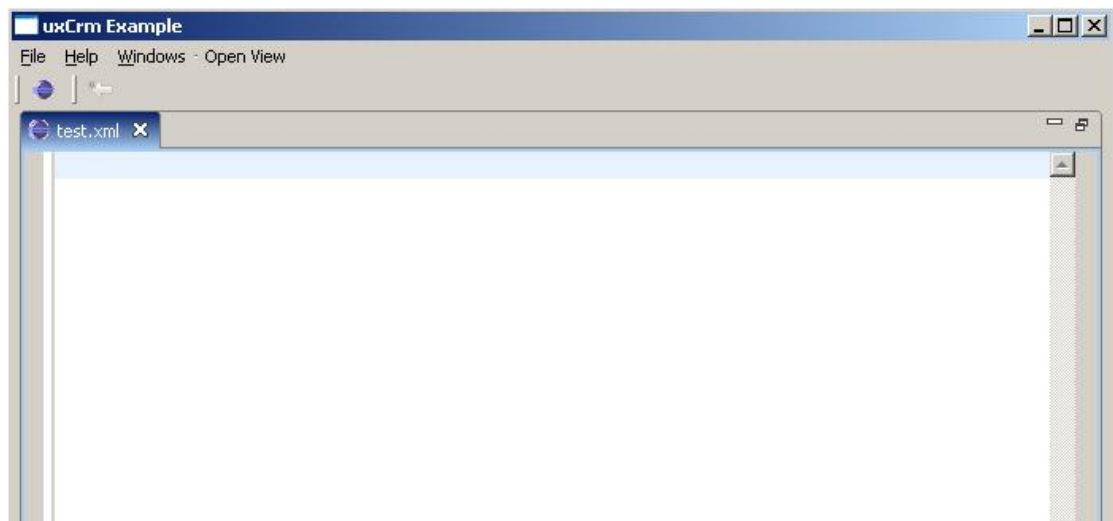
    } catch (Exception e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

    }
```

我们假设有一个名为 test 的工程在 workspace 下, 如果没有我们就创建, 找到 test 工程下的 test.xml 文件, 同样, 如果没有我们就创建它, 最后我们通过这个 IFile 对象生成了一个 FileEditorInput 对象, 利用 window 的 openEditor 方法编辑器, 下面我们代码运行结果:



6.3 文档和分区

文本编辑器中, 文档和分区是一个很重要的概念。

平台文本框架为文本定义文档模型, 并提供一个使用此模型来显示文本的查看器。我们将首先查看 XML 编辑器示例并了解它如何使用此模型。

在工作台中, 当用户选择域元素 (例如, 一个文件或者存储在归档文件中的元素) 并打开它时, 通常都会打开编辑器。当创建编辑器时, 编辑器就会与编辑器输入 (IEditorInput) (它描述正在被编辑的对象) 相关联。

当用户打开XML文件时, 将打开我们之前所生成的XML编辑器示例。在这种情况下, 编辑器的输入是 IFileEditorInput。平台文本框架很少对编辑器输入本身作出假设。它对输入使用称为 IDocument 的表示模型, 因此, 它可以有效地显示和处理文本。

这意味着必须具有一种方法从期望的域模型（编辑器输入）映射至表示模型。在 `IDocumentProvider` 中定义了此映射。只要给定编辑器输入，文档提供程序就会返回适当的 `IDocument`。

Java 编辑器示例继承插件 `org.eclipse.ui.editors` 定义的 `TextFileDocumentProvider`。扩展 `org.eclipse.ui.editors.documentProviders` 用于定义编辑器输入类型（或文件扩展）与文档提供程序之间的映射。XML 编辑器示例未定义它自己的文件提供程序扩展，所以它会继承对属于 `IStorageEditorInput` 的所有输入类型指定的通用文档提供程序。当用户打开文件进行编辑时，平台会管理有关创建适当的文档提供程序实例的详细信息。如果为文件扩展注册了特定文档提供程序，则将使用该提供程序。如果该文件扩展没有特定文档提供程序，则将使用编辑器输入类型来查找相应的提供程序。

通过使用通用平台文档提供程序，Java 编辑器示例可利用该文档提供程序的所有功能，如文件缓冲和其它优化。

既然 XML 编辑器使用平台文本文档提供程序，它会如何提供用于处理 XML 文件的任何专门行为呢？

扩展 `org.eclipse.core.filebuffers.documentSetup` 用于定义文件扩展与 `IDocumentSetupParticipant` 之间的映射。当然，我们也可以通过程序而不是 `plugin.xml` 配置文件来设置我们的 `DocumentSetup`。

一旦向编辑器提供了该文档，设置参与者就会使用所有特殊功能设置该文档。

我们来看一下 `XMLDocumentProvider` 的 `createDocument` 方法的简化版本，我们覆盖了这个方法，就是为了对文档进行设置，而绕过了 `Plugin.xml` 配置文件：

```
protected IDocument createDocument(Object element) throws
CoreException {
    IDocument document = super.createDocument(element);
    if (document != null) {
        IDocumentPartitioner partitioner =
            new FastPartitioner(
```

```

        new XMLPartitionScanner(),

        new String[] {

            XMLPartitionScanner.XML_TAG,

            XMLPartitionScanner.XML_COMMENT }));

    partitioner.connect(document);

    document.setDocumentPartitioner(partitioner);

}

return document;

}

```

设置文本配置称为分区程序 (partitioner) 的对象。

分区程序 (IDocumentPartitioner) 负责将文档划分成不重叠的区域 (称为分区)。分区 (由 ITypedRegion 中表示) 对于根据功能 (例如, 语法突出显示或格式化) 来以不同方式处理文档的不同部分是很有用的。

对于 XML 编辑器示例, 文档被划分成表示 XML 注释、XML 标签和其它内容的分区。对每个区域都指定了内容类型以及它在文档中的位置。当用户编辑文本时, 位置就会相应地更新。

由每个编辑器来确定文档分区程序的适当实现。在 org.eclipse.jface.text.rules 中提供了对基于规则进行文档扫描的支持。使用基于规则的扫描程序允许编辑器使用由框架提供的 FastPartitioner。

```

IDocumentPartitioner partitioner =

    new FastPartitioner(

        new XMLPartitionScanner(),

        new String[] {

            XMLPartitionScanner.XML_TAG,

            XMLPartitionScanner.XML_COMMENT }));

```

RuleBasedPartitionScanner 是基于规则的扫描程序的超类。子类负责枚举和实现一些规则, 当扫描文档时, 应当使用这些规则来区分标记 (例如, 行定界符、空格和标签)。

示例的 XMLPartitionScanner定义用于区分XML内容、XML注释、XML标签。这是在扫描程序的构造函数中完成的：

```
public XMLPartitionScanner() {
    IToken xmlComment = new Token(XML_COMMENT);
    IToken tag = new Token(XML_TAG);
    IPredicateRule[] rules = new IPredicateRule[2];
    rules[0] = new MultiLineRule("<!--", "-->", xmlComment);
    rules[1] = new TagRule(tag);
    setPredicateRules(rules);
}
```

6.4 配置 SourceViewer

SourceViewer 也是使用可插入行为（例如，内容斑竹和语法突出显示）来配置编辑器的中央集线器。对于这些功能部件，编辑器提供了 SourceViewerConfiguration，它用于在创建 SourceViewer 时配置它。我们的XML示例编辑器只需要提供满足其需要的 SourceViewerConfiguration。以下代码段说明 XMLtEditor 如何创建其配置。

```
public XMLtEditor() {
    super();
    colorManager = new ColorManager();
    setSourceViewerConfiguration(
        new XMLConfiguration(colorManager));
    setDocumentProvider(new XMLDocumentProvider());
}
```

XMLConfiguration执行什么操作？它的大部分行为都是从 SourceViewerConfiguration 继承的，后者定义可插入编辑器行为（例如，自动缩进、

撤销行为、双击行为、文本悬浮式帮助、语法突出显示和格式化) 的缺省策略。

SourceViewerConfiguration 中的公共方法提供实现这些行为的 helper 对象。

如果在 SourceViewerConfiguration 中定义的缺省行为不适合您的编辑器, 则应当覆盖以上显示的 initializeEditor(), 并将您自己的源代码查看器配置设置到编辑器中。您的配置可以覆盖 SourceViewerConfiguration 中的方法以提供实现编辑器的行为的定制 helper 对象。以下代码段说明了 XMLConfiguration 为 XML 编辑器示例提供定制 helper 对象的两种方法:

```
public IPresentationReconciler
getPresentationReconciler(ISourceViewer sourceViewer) {
    PresentationReconciler reconciler = new
PresentationReconciler();
    DefaultDamagerRepairer dr =
        new DefaultDamagerRepairer(getXMLTagScanner());
    reconciler.setDamager(dr, XMLPartitionScanner.XML_TAG);
    reconciler.setRepairer(dr, XMLPartitionScanner.XML_TAG);
    dr = new DefaultDamagerRepairer(getXMLScanner());
    reconciler.setDamager(dr,
IDocument.DEFAULT_CONTENT_TYPE);
    reconciler.setRepairer(dr,
IDocument.DEFAULT_CONTENT_TYPE);
    NonRuleBasedDamagerRepairer ndr =
        new NonRuleBasedDamagerRepairer(
            new TextAttribute(
                colorManager.getColor(IXMLColorConstants.XML_COMMENT)));
    reconciler.setDamager(ndr,
XMLPartitionScanner.XML_COMMENT);
    reconciler.setRepairer(ndr,
XMLPartitionScanner.XML_COMMENT);
}
```



```
        return reconciler;
    }
}
```

上面代码是为我们的语法作色所作的工作。

6.5 语法作色

语法着色是在使用损坏、修复和协调模型的平台文本框架中提供的。对于应用于文档的每个更改，表示协调程序确定可视表示的哪个区域应是失效的以及如何修复它。可以将不同的策略用于文档中的不同内容类型。

是否实现语法着色（以及使用表示协调程序来实现）是可选的。缺省情况下，`SourceViewerConfiguration` 不会安装表示协调程序，原因是它不知道用于特定编辑器的文档模型，并且没有语法突出显示的一般行为。

为了使用协调类来实现语法突出显示，必须配置编辑器的源代码查看器配置来定义表示协调程序。我们将再次从 `XMLConfiguration` 开始来查看如何为编辑器定义表示协调程序。

```
public IPresentationReconciler
getPresentationReconciler(ISourceViewer sourceViewer) {
    PresentationReconciler reconciler = new
PresentationReconciler();

    DefaultDamagerRepairer dr =
        new DefaultDamagerRepairer(getXMLTagScanner());
    reconciler.setDamager(dr, XMLPartitionScanner.XML_TAG);
    reconciler.setRepairer(dr, XMLPartitionScanner.XML_TAG);

    dr = new DefaultDamagerRepairer(getXMLScanner());
    .....
    return reconciler;
}
```

```
}
```

要了解表示协调程序执行哪些操作，必须首先了解损坏、修复和协调的概念。

当用户在编辑器中修改文本时，必须重新显示编辑器的各部件以显示更改。计算必须重新显示的文本的过程称为计算损坏。当涉及到语法着色时，编辑操作导致的损坏范围就更广了，原因是单个字符存在或不存在，都可能会更改它周围文本的颜色。

损坏程序（`IPresentationDamager`）确定由于文档更改而必须重建的文档表示的区域。假定表示损坏程序特定于特定文档内容类型（或区域）。它必须能够返回作为表示修复程序（`IPresentationRepairer`）的有效输入的区域。修复程序必须能够从损坏区域中派生它需要的所有信息才能成功描述特定内容类型所需要的修复。

协调描述当在编辑器中进行更改时维护文档的表示的整个过程。表示协调程序（`IPresentationReconciler`）通过它相关联的查看器来监视对文本的更改。它使用文档的区域来确定受更改影响的内容类型，并通知适合于受影响的内容类型的损坏程序。一旦计算了损坏，就会将它传递至适当的修复程序，该修复程序将构造适用于查看器的修复描述以使它与基本内容再次同步。

`org.eclipse.jface.text.reconciler` 中的类定义附加支持类，以使文档模型与文档的外部处理同步。

对于在文档中找到的每种内容类型，表示协调程序应当与修复程序和损坏程序对一起提供。由每个编辑器来确定表示协调程序的适当实现。但是，平台在 `org.eclipse.jface.text.rules` 中提供了使用基于规则的文档扫描程序来计算和修复损坏的支持。缺省损坏程序和修复程序是在此包中定义的。可以将它们与 `org.eclipse.jface.text.presentation` 中的标准协调程序配合使用来通过定义文档的扫描规则来实现语法着色。

现在，我们具有足够的背景知识来详细了解示例表示协调程序的创建过程。回想一下，XML 编辑器示例实现 `XMLPartitionScanner`。

对于每种这些内容类型，都必须指定损坏程序 / 修复程序对。这是在下面使用 `NonRuleBasedDamagerRepairer` 完成的。

```

        if (!documentPartitioningChanged) {
            try {

                IRegion info =

fDocument.getLineInformationOfOffset(event.getOffset());

                int start = Math.max(partition.getOffset(),
info.getOffset());

                int end =

                    event.getOffset()

                        + (event.getText() == null

                            ? event.getLength()

                            : event.getText().length());

                if (info.getOffset() <= end

                    && end <= info.getOffset() + info.getLength())
{

                    // optimize the case of the same line

                    end = info.getOffset() + info.getLength();

                } else

                    end = endOfLineOf(end);

                .....

            } catch (BadLocationException x) {

            }

        }

        return partition;

```

注意，该示例为每种内容类型都提供了扫描程序。

缺省内容类型是使用 `XMLScanner` 设置的，因此可以对关键字进行检测和着色。`XMLTagScanner` 将构建用于检测不同种类的标记（例如，单行注释、空格和词语）的规则。它描述应当用于不同标记类型的词语的颜色。

其它内容类型是使用 `XMLPartitionScanner` 设置的，并且给定了要用于这些内容类型中的标记的颜色。



6.6 内容帮助

在我们生成的类中，有一个名为 `XMLConfiguration` 的类，该类对 `XML Editor` 进行了一些设置，包括如何去为不同的文本区域显示不同的颜色等，`TextEditor` 所维护的 `SourceViewer` 就是通过它来进行设置的，但这不是我们所要讨论的范围，这里简单地介绍一下即可。

接下来我们需要复写 `XMLConfiguration` 的一个方法：`getContentAssistant`。这个方法便是告诉我们的编辑器，我们所具有的内容帮助是什么，在创建 `XML Editor` 的时候，默认是不为我们生成这方面代码的，所以我们需要自己复写：

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {  
    // 生成一个 ContentAssistant  
    ContentAssistant assistant = new ContentAssistant();  
  
    // 设置帮组内容弹出响应时间
```

```
        assistant.setAutoActivationDelay(200);

        assistant.enableAutoActivation(true);

        return assistant;
    }
}
```

ContentAssistant 并不是内容帮助的提供者，它只是维护我们的内容帮助，帮我们弹出菜单以及帮助内容信息等作用。

真正告诉 ContentAssistant 要显示那些帮助内容的，是 IContentAssistProcessor 接口类。让我们创建一个名为 StrutsContentAssisProcessor 的类，并让它实现 IContentAssistProcessor 接口：

```
public class StrutsContentAssisProcessor implements IContentAssistProcessor {

    public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer,
        int offset) {
        return null;
    }

    public IContextInformation[] computeContextInformation(ITextViewer viewer,
        int offset) {
        return null;
    }

    public char[] getCompletionProposalAutoActivationCharacters()
    {
        return null;
    }
}
```

```
    }

    public char[] getContextInformationAutoActivationCharacters()
    {
        return null;
    }

    public String getErrorMessage() {
        return null;
    }

    public IContextInformationValidator getContextInformationValidator() {
        return null;
    }
}
```

大家注意下 `computeCompletionProposals` 方法，这个方法便是返回我们的具体内容帮助。所以我们需要为我们的编辑器创建所需要的内容帮助：`CompletionProposal`

先看一下这个类的构造函数各个参数的含义：

- **replacementString** 选择帮助信息后所要替代的文本内容
- **replacementOffset** : 替代内容输入的位置
- **replacementLength** : 替代文本覆盖原来文本的长度
- **cursorPosition** : 完成内容帮助的文本替代后，光标所在位置
- **image** : 帮助内容显示的图标
- **displayString** : 帮助内容的显示字符串

➤ **contextInformation** : 帮助内容的信息描述

➤ **additionalProposalInfo** : 附加信息

在这几个参数中 `image`、`contextInformation`、`additionalProposalInfo` 我们可以设置为空。现在让我们在 `computeCompletionProposals` 生成我们的帮助内容:

```
public ICompletionProposal[] computeCompletionProposals(
    ITextViewer viewer, int offset) {
    ICompletionProposal[] proposals =
    new ICompletionProposal[2];
    proposals[0] = new CompletionProposal(
        "替换文本 1",
        offset,
        0,
        new String("替换文本 1").length(),
        null, "帮组内容 1",
        null,null) ;
    proposals[1] = new CompletionProposal(
        "替换文本 2",
        offset,
        0,
        new String("替换文本 2").length(),
        null,
        "帮组内容 2", null,null) ;

    return proposals;
}
```

`computeCompletionProposals` 输入的参数中 `offset` 是指当内容帮助弹出的时候, 文本编辑器光标所在位置。

大家都知道，帮助内容弹出的时候是需要一定条件的，也就是当我们输入了激活内容帮助的字符的时候，它便会弹出来。IContentAssistProcessor 的 getCompletionProposalAutoActivationCharacters 方法便是让我们返回激活帮助内容字符的，假设当我们输入了'<'时，弹出帮助内容：

```
public char[] getCompletionProposalAutoActivationCharacters()
{
    return new String("<").toCharArray();
}
```

好了，我们的第一步已经完成了，接下来就是在 ContentAssis 对象中设置我们所要返回的内容帮助。

返回到 XMLConfiguration 的 getContentAssistant 方法：

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {
    // 生成一个 ContentAssistant
    ContentAssistant assistant = new ContentAssistant();

    // 让帮助内容在 XML 的 Tag 标签范围内激活
    assistant.setContentAssistProcessor(
        new StrutsContentAssisProcessor(),
        XMLPartitionScanner.XML_TAG);

    // 让帮助内容在 XML 文本的默认文本区域内激活
    assistant.setContentAssistProcessor(
        new StrutsContentAssisProcessor(),
        IDocument.DEFAULT_CONTENT_TYPE);

    // 设置帮组内容弹出响应时间
    assistant.setAutoActivationDelay(200);

    assistant.enableAutoActivation(true);

    return assistant;
}
```



```
}
```

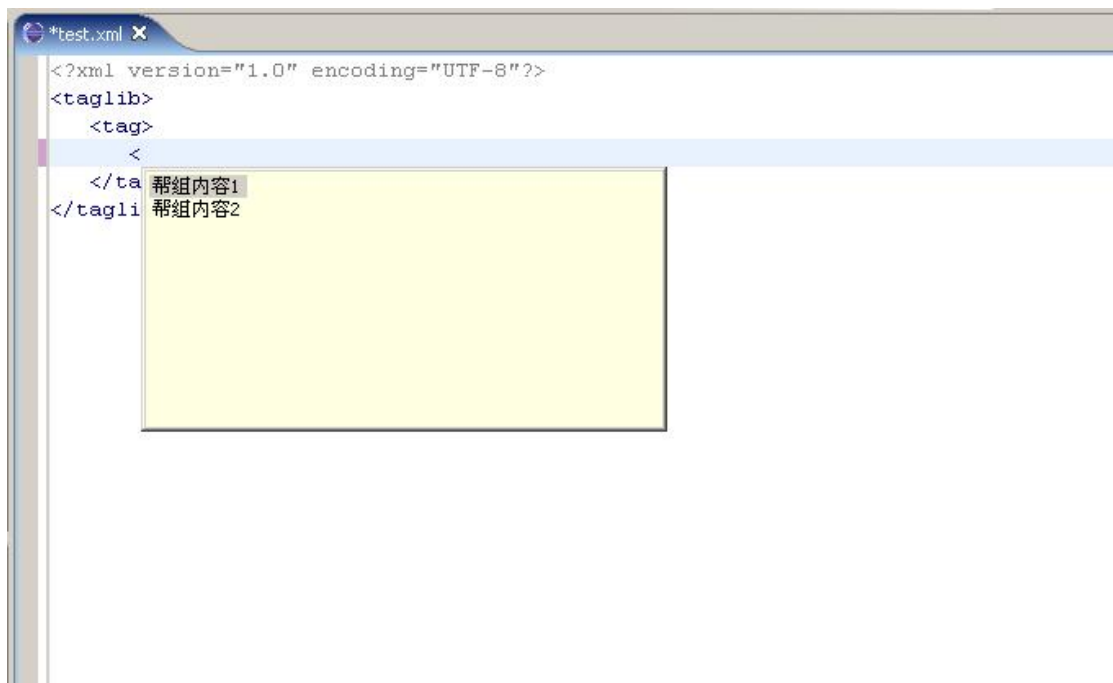
我们加入了两句代码，给 ContentAssistant 设置了我们生成的 StrutsContentAssisProcessor 对象：

XMLPartitionScanner.XML_TAG: XML 的标签区域，既在以“<”开始到“>”结束的文本范围内

IDocument.DEFAULT_CONTENT_TYPE : 文本编辑器的默认文本区域

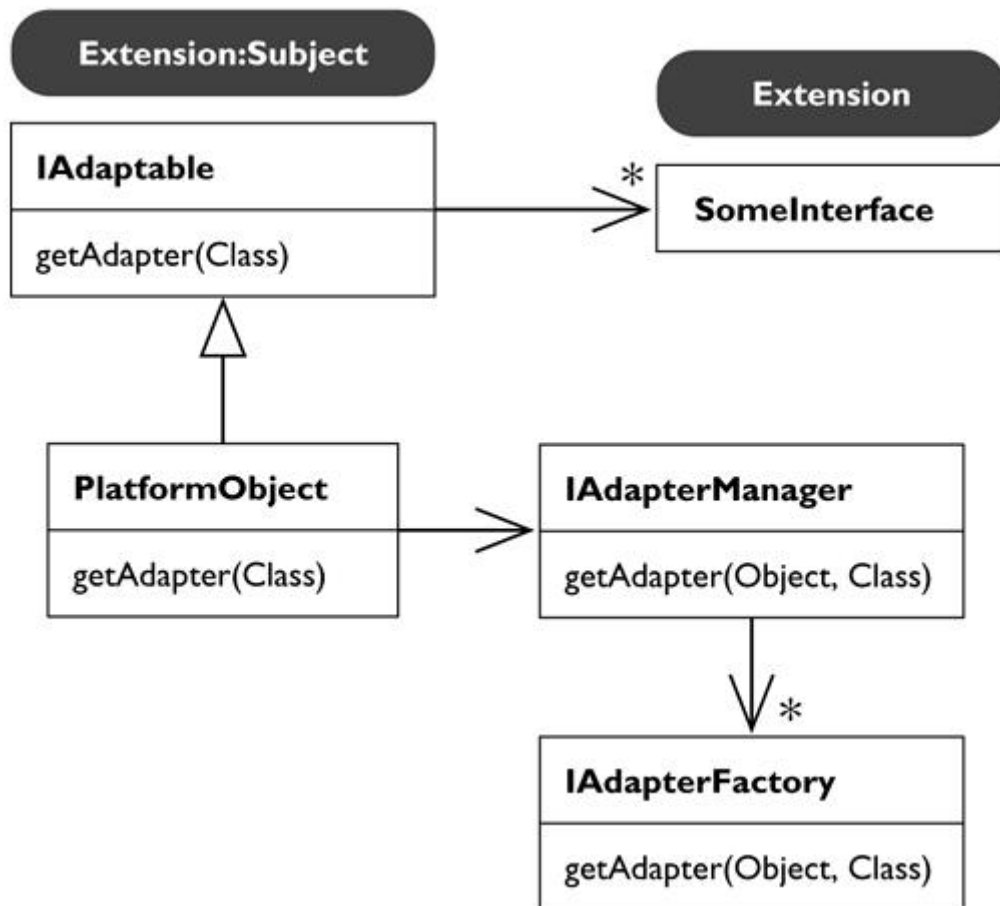
让我们运行一下，并以一个简单的 XML 文本作为参考：

在 XML 的标签中，当我们输入“<”的时候，便会弹出我们的帮助内容。



7. IAdaptable 和 IAdaptableFactory

IAdaptable在Eclipse里是一个非常重要的接口。对于Eclipse开发老手来说，它就像异常处理和抽象类一样寻常；但是对新手而言，它却令人感到困惑和畏惧。这篇文章将向你解释IAdaptable到底是什么，以及它在Eclipse里起到的作用。



Java 是所谓的强类型语言，也就是说，每个实例都对应一个类型。其实类型分为两种：声明类型和运行时类型（也分别被称为静态类型和动态类型）。像 Python 这样的弱类型语言常被称为无类型的语言，其实严格说来不是这样，因为每个实例都对应一个运行时类型，只是你并不需要声明这一点而已。

现在回到 Java，为了能够执行一个类的某个方法，这个方法必须在声明类型中可见，换句话说，即使在运行时实例是某个子类型，你也只能执行那些父类型里定义的方法。

```

List list = new ArrayList();

list.add("data");           // 正确，add 是 List 里定义的方法

list.ensureCapacity(4);     // 不正确，ensureCapacity() 只在 ArrayList 被
                             定义
  
```

如果一定要执行特定类型的方法，我们必须先强制转换这个实例到正确的类型。对于上面的例子，我们可以将 `list` 转换为 `ArrayList`（译注：原文 In this case, we can

cast ArrayList to List, 怀疑是笔误), 因为 ArrayList 实现了 List 接口, 你甚至可以在运行时通过 instanceof 关键字检验 list 是否为 ArrayList 的一个实例。

不幸的是, 一个类可能并没有实现你需要的接口, 这样就无法进行强制类型转换了。原因有很多, 比如只在少数情况下才需要这个接口, 或者你需要的接口是在另一个不相关的库里, 又或者接口是有了类以后才开发出来的, 等等。

这时你就需要 IAdaptable 了。可以把 IAdaptable 想象为一个能够动态进行类型转换的途径。对比下面的直接类型转换:

```
Object o = new ArrayList();  
List list = (List)o;
```

换一种方式, 我们可以这样做:

```
IAdaptable adaptable = new ArrayList();  
//译注: 这里的 ArrayList 应该不是指 java.util.ArrayList  
List list = (List)adaptable.getAdapter(java.util.List.class);
```

这就是上面所说的动态类型转换, 我们所做的事情是试图把 adaptable 转换为一个 List 实例。

那么, 当可以直接转换的时候为什么要费这个力气通过 getAdapter() 来转换呢? 其实这种机制可以让我们将目标类转换为它并没有实现的接口。举个例子, 我们可能想把一个 HashMap 当作 List 来用, 尽管这两个类的性质并不相同, 可以这么做:

```
IAdaptable adaptable = new HashMap(); //译注: 这里的 HashMap 应该不是指 java.util.HashMap  
List list = (List)adaptable.getAdapter(java.util.List.class);
```

大部分 IAdaptable 的实现是一些 if 语句的叠加, 比如我们现在要实现 HashMap 的 getAdapter() 方法, 它看起来可能是这样:

```
public class HashMap implements IAdaptable {  
    public Object getAdapter(Class clazz) {  
        if (clazz == java.util.List.class) {
```

```
List list = new ArrayList(this.size());

    list.addAll(this.values());

    return list;
}

return null;
}
```

所做的就是返回一个适配器（adapter，更确切的说是一个副本），而不是进行直接的类型转换。如果参数类型没有被支持，惯例是返回 null 值（而非抛出异常），代表这个方法失败了。因此，在调用这个方法时，不应该假定它总是返回非 null 值。

当然，如果你希望增加一个新的被支持的 adapter 类型时必须编辑这个类才行（译注：在 getAdapter() 里增加更多的 if 语句），这会比较辛苦。而且，既然你已经知道了这个类型，何不直接修改接口声明呢？其实有很多原因使得你并不希望直接编辑这个类（例如更容易保持向下兼容性），也不想改变它的类型（HashMap 虽然不是一个 List，但可以转换过去）。

Eclipse 通过 PlatformObject 抽象类来解决以上问题，它为你实现了 IAdaptable 接口，Eclipse 平台（Platform）提供了 IAdapterManager 的一个实现，并且可以通过 Platform.getAdapterManager() 访问到，它把所有对 getAdapter() 的请求（调用）委托给一个名为 IAdapterManager 的东西。你可以将它想象为一个巨大的保存着类和 adapter 信息的 Map，而 PlatformObject 的 getAdapter() 方法会查找这个 Map。

这样，PlatformObject 不需要重新编译就能够支持新的 adapter 类型，这一点在 Eclipse 里被大量使用以支持 workspace 的扩展点。

现在假设我们想要将一个只包含 String 类型元素的 List 转换为一个 XML 节点，这个节点的格式如下：

```
<List>

    <Entry>First String</Entry>
```

```
<Entry>Second String</Entry>

<Entry>Third String</Entry>

</List>
```

因为 `toString()` 方法可能有其他用途，我们不能通过覆盖 `toString()` 方法来实现这个功能。所以，我们要给 `List` 关联一个工厂类以处理 XML 节点类型的适配请求。要管理工厂类需要以下三个步骤：

1、由 `List` 生成一个 `Node`，我们把这个转换过程用 `IAdapterFactory` 包装起来：

```
import nu.xom.*;

public class NodeListFactory implements IAdapterFactory {

    /* 可以转换到的类型 */
    private static final Class[] types = {
        Node.class,
    };

    public Class[] getAdapterList() {
        return types;
    }

    /* 转换到 Node 的功能代码 */
    public Object getAdapter(Object list, Class clazz) {
        if (clazz == Node.class && list instanceof List) {
            Element root = new Element("List");
            Iterator it = list.iterator();
            while(it.hasNext()) {
                Element item = new Element("Entry");
                item.appendChild(it.next().toString());
                root.appendChild(item);
            }
            return root;
        } else {
```

```
        return null;
    }
}
}
```

2、把这个工厂类注册到 Platform 的 AdapterManager，这样当我们希望从 List 的实例中获得一个 Node 实例时，就会找到我们的工厂类。注册一个工厂类的方式也很简单：

```
Platform.getAdapterManager().registerAdapters(
    new NodeListFactory(), List.class
);
```

这条语句将 NodeListFactory 关联到 List 类型。当从 List 里请求 adapter 时，Platform 的 AdapterManager 会找到 NodeListFactory，因为在后者的 getAdapterList() 方法的返回结果里包含了 Node 类，所以它知道从 List 实例得到一个 Node 实例是可行的。在 Eclipse 里，这个注册步骤一般是在 plugin 启动时完成的，但也可以通过 org.eclipse.core.runtime.adapters 扩展点来完成。

3、从 List 获得 Node，下面是例子代码：

```
Node getNodeFrom(IAdaptable list) {
    Object adaptable = list.getAdapter(Node.class);
    if (adaptable != null) {
        Node node = (Node)adaptable;
        return node;
    }
    return null;
}
```

综上所述，要在运行时为一个已有的类增加功能，所要做的只是定义一个用来转换的工厂类，然后把它注册到 Platform 的 AdapterManager 即可。这种方式在保持 UI 组件和非 UI 组件的分离方面特别有用。例如在 org.rcpapps.rcpnews.ui 和 org.rcpapps.rcpnews 这两个 plugin 里，前者的 IPropertySource 需要与后者的

数据对象（data object）相关联，当前者初始化时，它将 `IPropertySource` 注册到 `Platform`，当数据对象在导航器（navigator）里被选中的时候，属性视图里就会显示正确的属性。

显然，`java.util.List` 并不是 `PlatformObject` 的子类，所以如果你希望能够编译这里所说的例子，必须建立一个 `List` 的子类型。注意，可以直接实现 `IAdaptable` 接口，而非必须继承 `PlatformObject` 抽象类。

```
public class AdaptableList implements IAdaptable, List {
    public Object getAdapter(Class adapter) {
        return Platform.getAdapterManager().getAdapter(this, adapter);
    }
    private List delegate = new ArrayList();
    public int size() {
        return delegate.size();
    }
}
```

8. OSGI 和 Eclipse Plugin

8.1 Plugin Architecture 简述

Plugin Architecture，无疑是如今软件界最为热门的名词，在各种各样的解决方案、白皮书中经常都能看到即插即用这么几个字，但真的又有多少软件做到了呢，当然，不可否认的是也有部分的软件确实做到了，而且做的很好，象 `Eclipse`, `maven` 这些都是，其实 `portlet container` 那些也都是的，列举出来还真的有不少。插件式系统带来的好处很明显，最大程度的重用，为快速的搭建系统提供帮助，潜在的好处在于要求系统以插件式的方式进行设计，帮助你更好的做到模块化的划分以及帮助系统达到良好的封装性。

要做 Plugin Architecture，首先需要做的是如何考虑 Plugin，其实也就是需求，要看你是怎么看待一个 Plugin 的，认为一个 Plugin 应该是怎么样的，认为 Plugin 是

怎么样被组装起来构成系统的，这个时候需要的是大家从需求的角度来提出要求，不要从技术角度来提。一个系统既然是按照 Plugin 的方式搭建出来，那么首先需要知道的就是 Plugin 能提供什么样的功能，这些功能需要什么参数，这是最基本的，其次是如何去管理这个 Plugin，包括修改它的配置参数，对它的生命周期进行管理(启动/暂停/停止等)，这是以单独的 Plugin 角度来看，如果这个 Plugin 又得调用其他 Plugin 提供的功能，那么应该怎么做，考虑了这些后又想到 Plugin 应该怎么去部署、怎么去自动升级。从扩展方面我们又会考虑到那么我要如何去扩展一个 Plugin 呢，这也是很关键的。这都只是简简单单的提了一些 Plugin 的需求，归纳上面的需求可以得出主要的几点就是 Plugin 的功能的暴露、Plugin 的管理、Plugin 的调用、Plugin 之间的协作、Plugin 的部署、Plugin 的扩展这几个大的方面。

8.2 OSGI 介绍

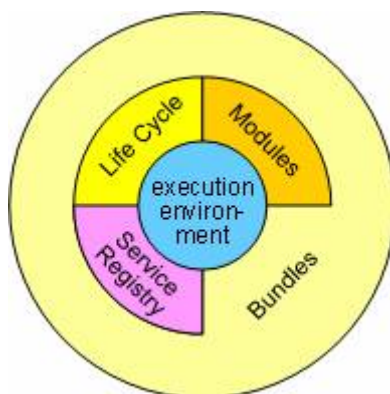
OSGI 规范为网络服务定义了一个标准的、面向组件的计算环境。将 OSGI 服务平台添加到一个网络设备中，可以为其增加在网络的任何地方管理组件的生命周期的能力。软件组件可以从运行中被安装、升级或者移除而不需要中断设备的操作。软件组件可以动态的发现和使用的其他库或者应用程序。通过这个平台，软件组件可以作为商品在柜台中出售以及在家里开发。OSGI 联盟已经开发出很多标准组件接口，从普通的功能如：HTTP server、configuration、logging、security、user administration、XML 等等很多。一致的插件机制可以使这些组件满足不同买主的不同需求。

软件组件架构致力于一个软件开发中越来越大的问题：大量的基础配置需要开发和维护。标准化的 OSGI 组件架构显然可以简化这个配置过程。

OSGI 规范的核心组件使 OSGI 框架。该框架为应用程序(被成为 bundles)提供一个标准化的环境。这个框架被分为以下几个层次：

- L0：执行环境
- L1：组件模块
- L2：组件生命周期管理
- L3：服务注册

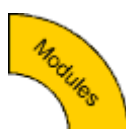
另外，还有一个安全机制深深的缠绕在所有的层中。



L0: 执行环境就是 java 环境的规范。Java2 配置和 profiles, 如 j2se、CDC、MIDP 等等都是可用的执行环境。OSGi 还标准化了一个基于基本 profile 的执行环境和一个可用于 OSGi bundles 的最小执行环境的规范。



L1: 模块层定义了类加载策略。OSGi 框架是一个健壮而严格定义的类加载模型。它基于 java 但是更加模块化。在 java 中, 通常只有一个单独的 classpath 包含所有的 class 和 resource。OSGi 模块层为一个模块添加私有的类并控制模块之间的关联。



L2: 生命周期层添加能够动态的安装、启动、停止、升级和卸载的 bundles。Bundles 加载 class 时依赖于模块层, 但也一个 API 在运行期管理模块。生命周期层引入了通常不属于应用一部分的动态性。广泛的依赖机制过去常用于确认环境的当前操作。



L3 层添加了一个服务注册器。服务注册器为 bundles 提供了一个协作模块用于动态注册。Bundles 可以通过传统的 class 共享来协作, 但是 class 共享与动态安装和卸载的代码不太协调。服务注册器提供了一个全面的模块似的 bundles 可以共享对象。一些事件被定义来处理服务的加载和卸载。服务只是一些能够代表任何东西的 java 对象。很多服

务的活动象一个 HTTP 服务器，其他服务代表了真实世界中的一个对象，比如：附近的一个蓝牙电话。



安全是基于 java 和 java2 的安全模块。语言的设计限制了许多可能的结构。比如在病毒中常用的 buffer 溢出是不可能出现的。语言中的访问控制限制了其他开发者对代码的可见度。OSGI 通过允许在标准的 java 中不可见的私有类来扩展这个模型。Java2 的安全模块提供了一个全面的模型来检查代码对资源的访问权限。OSGI 也添加了全面的动态权限管理。

在框架的上层，OSGI 联盟定义了很多 services。Services 通过 java 接口定义。Bundles 可以实现这些接口并注册到 service 注册器上。这些 service 的客户端可以通过注册器找到它们，或者当它出现/消失时对它作出反应。

OSGI 框架提供一个权限管理服务，一个包管理服务和一个启动级别服务。这些服务是框架的一部分（可选）并指向操作的。

框架服务如下：

- **Permission Admin:** 通过本服务现在或将来能够操作的 bundles。权限在它们被设置时即时生效。
- **Package Admin :** Bundles 同 Classes 和 resources 共享包。Bundles 的升级可能需要系统重新计算依赖关系。这个 Package Admin 服务提供实际包在系统中的共享情况的信息，并且能够刷新已共享的包。I.e.打破依赖和重新计算依赖。
- **Start Level :** 启动级别是一组应该一起运行或在其他服务之前初始化的 bundles。启动级别服务设置当前启动级别，指定一个 bundle 的启动级别并且查看当前设置。

系统服务提供每个实际系统所需要的底层功能。例如：Log Service, Configuration Admin Service, Device Access Service, User Admin Service, IO Connector Service 和 Preferences Service。

➤ **Log Service:** 记录通过 log service 处理的 information, warnings, debug 信息或者 error。它获得 log 实体然后分派这些消息实体到订阅该消息的 bundles。

➤ **Configuration Admin Service:** 该 service 提供一个灵活和动态的模型来设置和访问配置信息。

➤ **Device Access Service:** 设备访问是一个 OSGI 机制, 当有新的设备添加时, 它为新设备匹配驱动, 并下载一个 bundle 来实现该驱动。这对即插即用的情形非常有用。

➤ **User Admin Service:** 这个服务使用一个用户信息的数据库 (私有或共有) 来达到授权和认证的目的。

➤ **IO Connector Service:** IO Connector Service 实现 CDC/ CLDC javax.microedition.io 包作为一个 service. 这个 service 允许 bundles 实现新的、可选的协议。

➤ **Preferences Service:** 这个 service 提供分层的属性数据库的访问。类似于 windows 的注册表或者 java 的属性类。

OSGI 联盟还定义了一组服务, 每个 OSGI 服务对应一个外部协议:

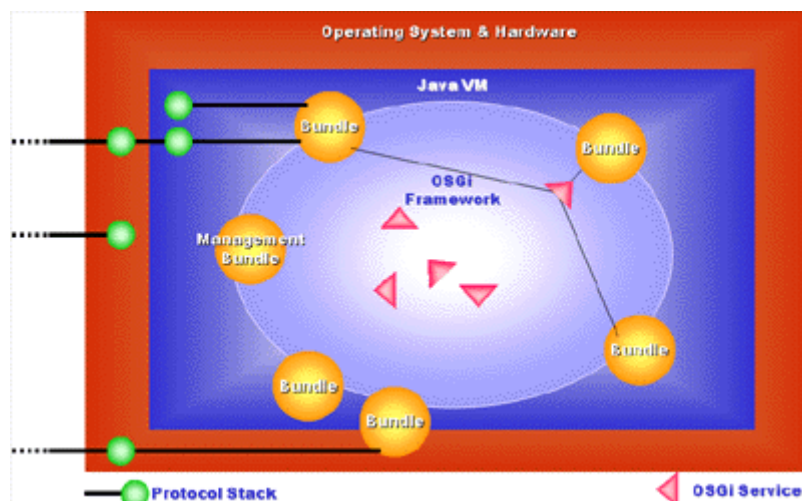
➤ **Http Service:** Http service 是一个 servlet 运行环境。Bundles 能够提供可通过 http 协议访问的 servlets。OSGI 平台的动态更新工具使 http service 变成了一个非常吸引人的 web server, 因为它可以远程动态更新 servlet 而不需要重启

➤ **UPnP Service:** 统一即插即用 (UPnP) 是一个对用户电器暴露的统一接口。OSGI UPnP Service 通过 service 注册器对应一个设备到一个 UPnP 网络。可选的, 它也能够对应一个 OSGI service 到一个 UPnP 网络。这在 release3 规范中是被推荐的做法。

➤ **Jini Service:** Jini 是一个网络协议用来发现网上的 Jini 服务并从该服务器上下载这些服务的 java 代码并执行它们。这在 release3 规范中是被推荐的做法。

➤ Wire Admin Service : 通常 bundles 建立寻找它们想要与之合作的 service 的规则。然而, 通常这应该在部署时决定。Wire Admin service 将一个配置文件中定义的不同 service 连接到一起。Wire Admin service 使用生产者消费者模式在 service 之间交换对象。

➤ XML Parser Service: XML Parser service 允许一个 bundle 通过指定的属性定位一个解析器并同 JAXP 兼容。



OSGI 规范有广泛的适用性, 是因为它是在一个单独的 JVM 中的一个很小的层, 并允许多样的、基于 java 的、组件的高效合作。它提供一个可扩展的安全机制以便组件可以运行在一个受保护的环境。而且, 通过适当的权限, 组件可以重用和合作, 而不像其他 java 应用环境。OSGI 框架提供一个可扩展排队机制使得这种合作变得可能和安全。

8.3 Eclipse Plugin 和 OSGI

Eclipse 中的 Plugin 的概念为包含一系列服务的模块即为一个 Plugin。既然是遵循 OSGI 的, 也就意味着 Plugin 通常是由 Bundle 和 N 多 Service 共同构成的, 在此基础上 Eclipse 认为 Plugin 之间通常存在两种关系, 一种为依赖, 一种为扩展, 对于依赖可通过 OSGI 中元描述信息里添加需要引用的 Plugin 即可实现, 但扩展在 OSGI 中是没有定义的, Eclipse 采用了一个 Extension Point 的方式来实现 Plugin 的扩展功能。Eclipse 遵循 OSGI 对于 Plugin 的 ID、版本、提供商、classpath、所依赖的 plugin 以及可暴露对外的包均在 manifest.mf 文件中定义。对于扩展, Eclipse 采用 Extension Point 的方式来实现, 每个 Plugin 可定义自己的 Extension Point, 同时也可实现其他 Plugin 的 Extension Point, 由于这个在 OSGI 中是未定义的, 在 Eclipse 中仍然通过在 plugin.xml 中进行描述, 描述的方法为通过

`<extension-point id="" name="" schema="">`的形式来定义 Plugin 的扩展点, 通过`<extension point="">`的形式来定义实现的其他 Plugin 的扩展点, 所提供的扩展点通过 schema 的方式进行描述, 详细见 eclipse extension-point schema 规范, 为了更好的说明扩展点这个概念, 举例如下, 如工具栏就是工具栏 Plugin 提供的一个扩展点, 其他的 Plugin 可通过此扩展点添加按钮至工具栏中, 并可相应的添加按钮所对应的事件(当然, 此事件必须实现工具栏 Plugin 此扩展点所要求的接口), 工具栏的 Plugin 将通过 callback 的方式来相应的响应按钮的动作。可见通过 Extension Point 的方式可以很好的提供 Plugin 的扩展方式以及实现扩展的方式。

那么 Eclipse 是如何做到 Plugin 机制的实现的呢? ? 还是先讲讲 Eclipse 的设计风格, Eclipse 在设计时有个重要的分层法则, 即语言层相关和语言层无关的代码分开(如 `jdt.core` 和 `core`), 核心与 UI 分开(如 `workbench.ui` 和 `workbench.core`)这两个分层法则, 这个在 Eclipse 代码中处处可见, 在 Plugin Framework 部分也充分得体现了这个, 遵循 OSGI, Eclipse 首先是实现了一个 OSGI Impl, 这个主要通过它的 `FrameWork`、`BundleHost`、`ServiceRegistry`、`BundleContextImpl` 等对象来实现, 如果关心的话大家可以看看这部分的代码, 实现了 Bundle 的安装、触发、卸载以及 Service 的注册、卸载、调用, 在 Plugin 机制上 Eclipse 采用的为 lazy load 的方式, 即在调用时才进行实际的启动, 采用的为句柄/实体的方式来实现, 外部则通过 OSGI 进行启动、停止等动作, 各 Plugin 则通过 `BundleContext` 来进行服务的注册、卸载和调用, 这是 OSGI 的部分实现的简单介绍。

那么 Extension Point 方面 Eclipse 是如何实现的呢, 在加载 Plugin 时, Eclipse 通过对 `plugin.xml` 的解析获取其中的`<extension-point>`节点和`<extension>`节点, 并相应的注册到 `ExtensionRegistry` 中, 而各个提供扩展点的 Plugin 在提供扩展点的地方进行处理, 如工具栏 Plugin 提供了工具栏的扩展点, 那么在构成工具栏时 Plugin 将通过 `Platform.getPluginRegistry().getExtensionPoint(扩展点 ID)` 的方法获取所有实现此扩展点的集合 `IExtensionPoint[]`, 通过此集合可获取 `IConfigurationElement[]`, 而通过这个就可以获取`<extension point="">`其中的配置, 同时还可通过 `IConfigurationElement` 创建回调对象的实例, 通过这样的方法 Eclipse 也就实现了对于 Plugin 的扩展以及扩展的功能的回调。在 Plugin Framework 中还涉及很多事件机制的使用, 比如 Framework 的事件机制, 以便在 Bundle 注册、Service 注册的时候进行通知。

我们把 Eclipse 插件的功能点和 OSGI 结合起来总结出下面几点：

➤ 插件的定义。

OSGI 规范中将插件称为 Bundle，Bundle 作为整个插件的生命周期管理对象，负责插件的启动和停止动作，通过 `Meta-inf/mainfest.mf` 来描述 Bundle，主要描述 Bundle 的名称、厂商、版本、对外暴露的包、对外暴露的服务、依赖的插件、引用的包、动态引用的包等，具体可参考 OSGI R4 中 Framework Specification Chapter，插件可通过 Bundle 对象获取插件的定义信息。

➤ 插件的加载。（文件、URL 等形式）

OSGI 规范中定义通过 BundleContext 来完成 Bundle 的加载工作，每个 Bundle 拥有独立的 classloader 以及 BundleContext。

➤ 插件的生命周期管理。

OSGI 规范中定义通过 BundleContext 对 Bundle 进行生命周期的管理，或通过 Bundle 本身对象来进行。

➤ 插件间的交互机制。

OSGI 规范中定义通过插件的定义中定义所需依赖的插件以及所需引用的包来实现插件的交互机制。

➤ 插件的开发。

OSGI 规范中定义一个新的插件的开发需要的是构成其 BundleActivator 对象以及完成插件定义的描述。

➤ 插件所暴露的功能。

OSGI 规范中定义通过在插件定义文件中描述插件所暴露对外的服务来说明插件对外所暴露的功能以及允许外部对此插件引用的包。

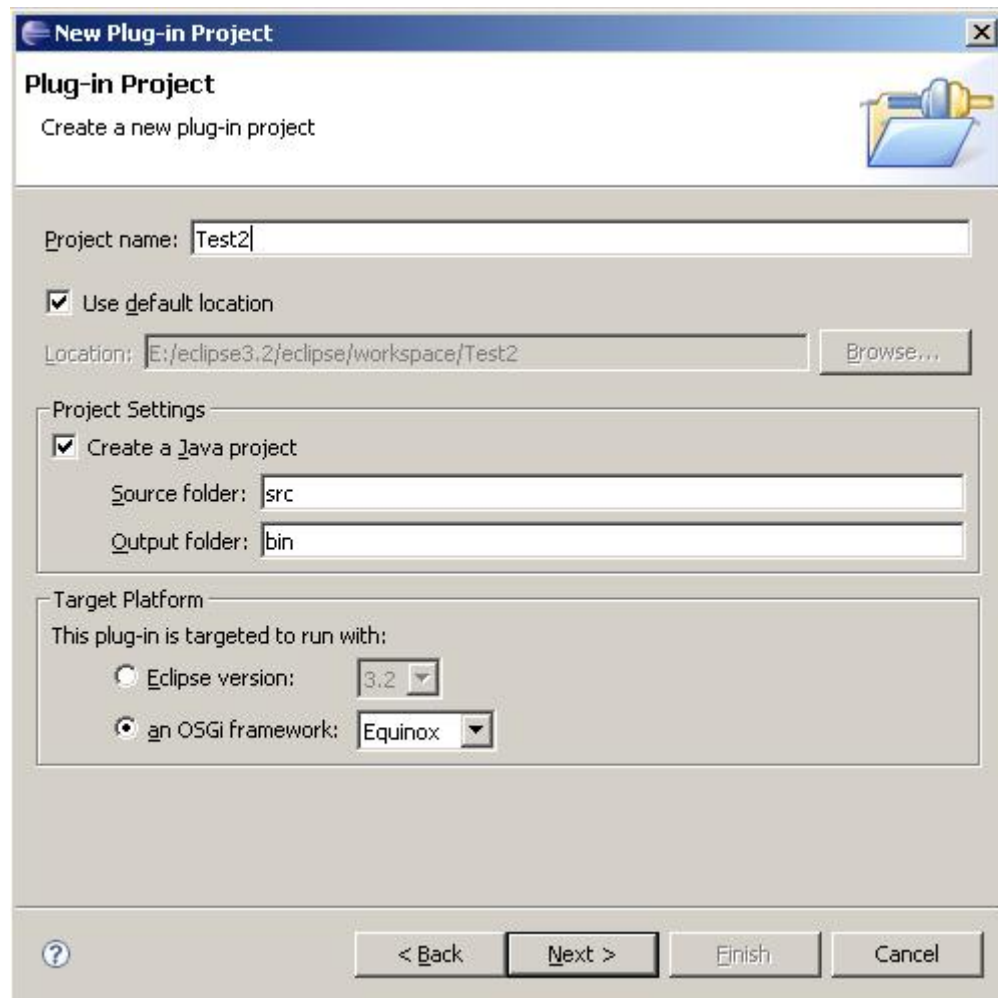
对于插件如何扩展 OSGI 规范中提及的是在修改插件后插件的自动更新以及热加载来实现，而不是象 Eclipse 的扩展点机制。

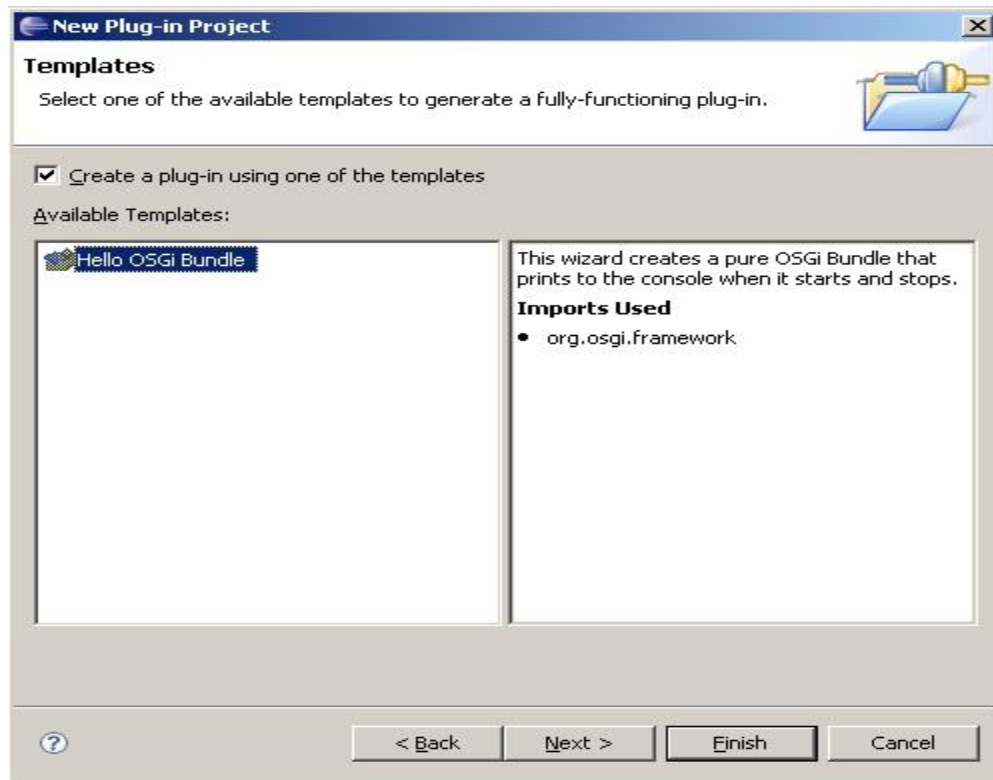
根据上面我们可以看出 OSGI 规范确实非常适用于 Plugin Architecture。

8.4 一个简单的 Eclipse OSGI Framerwork 程序

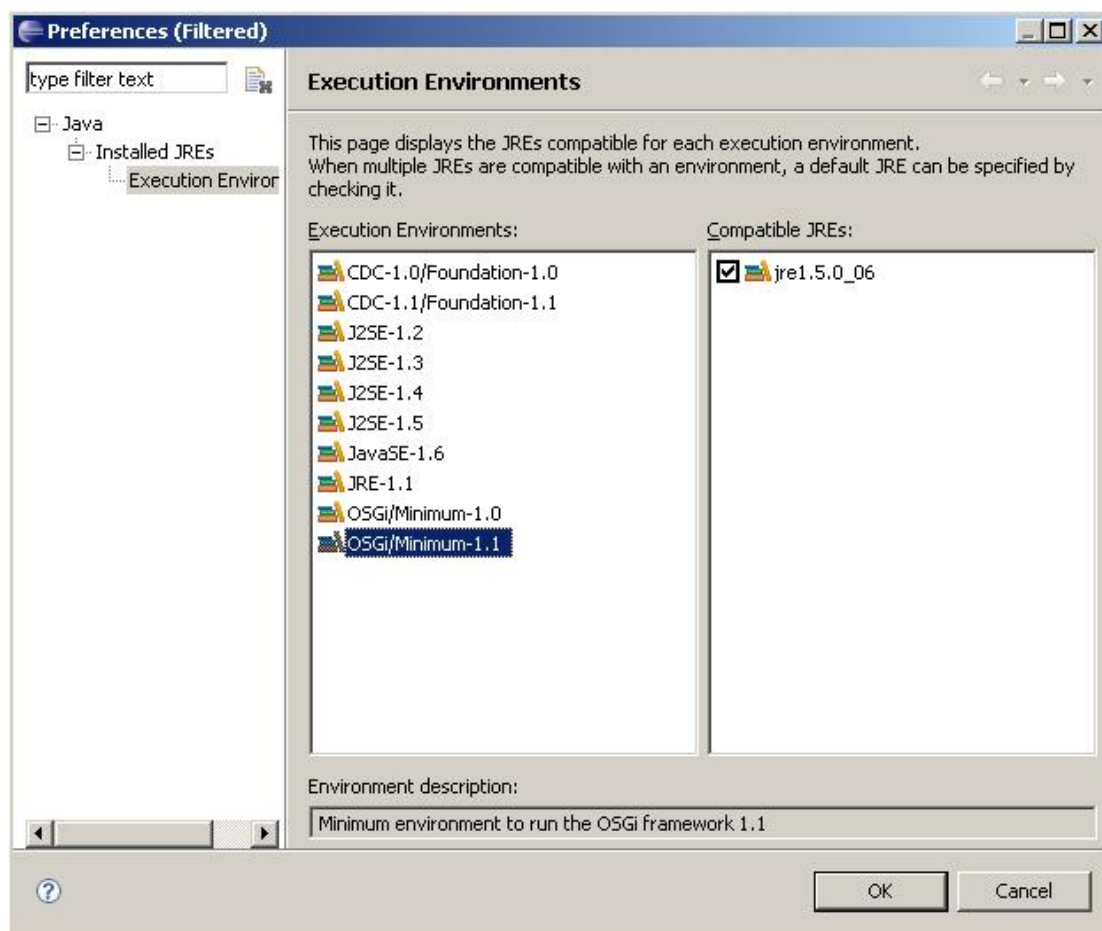
首先我们需要安装 Eclipse3.2。

通过 Eclipse3.2 的 plugin project 向导，我们创建一个 OSGI 的项目：





点击完成后我们会得到我们的项目，然后我们需要配置他的 JDK。打开 plugin.xml，在第一页中我们点击 Configure JRE，然后选择 OSGI—Minimum1.1:



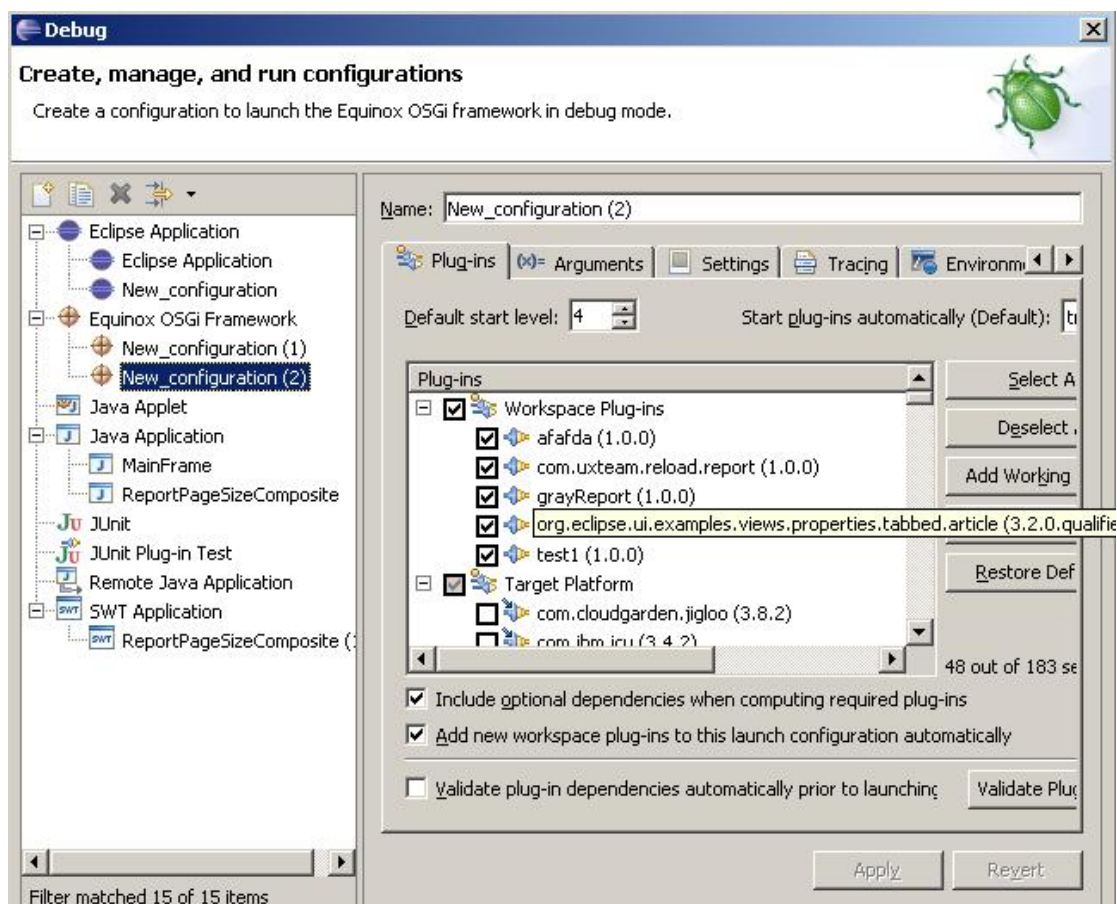
看看我们生成的代码，我们会更清楚。

```
public class Activator implements BundleActivator {  
    public void start(BundleContext context) throws Exception {  
        System.out.println("Hello World!!");  
    }  
  
    public void stop(BundleContext context) throws Exception {  
        System.out.println("Goodbye World!!");  
    }  
}
```

我们生成的这个 Bundle 很简单，仅仅是在启动和停止的时候打印一句话。

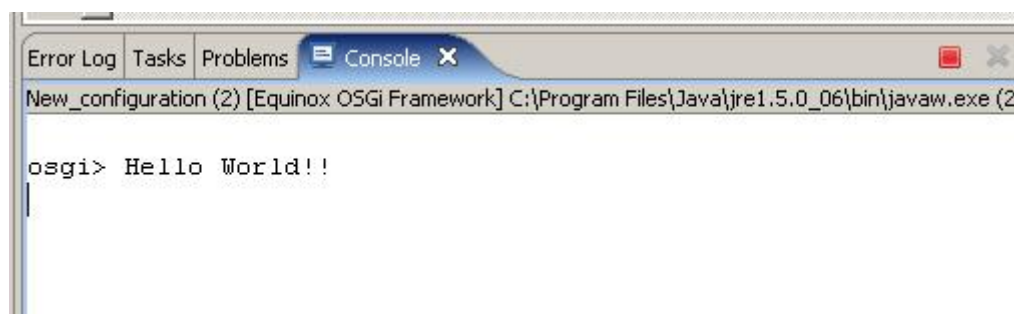
现在让我们来调试一下：

选择 Eclipse 的 Debug 选项，双击 Equinox OSGi Framework，然后配置需要的 Plugin 后点击 debug



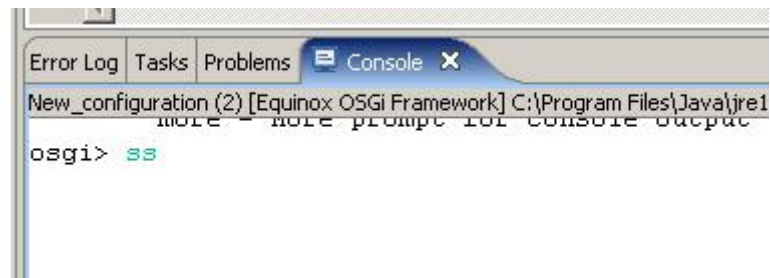
Debug 开始后我们会发现，和往常的 Eclipse 插件程序不同，这次没有起来我们的 Eclipse 平台。

打开 Console 视图我们发现，OSGi 开始运行了：

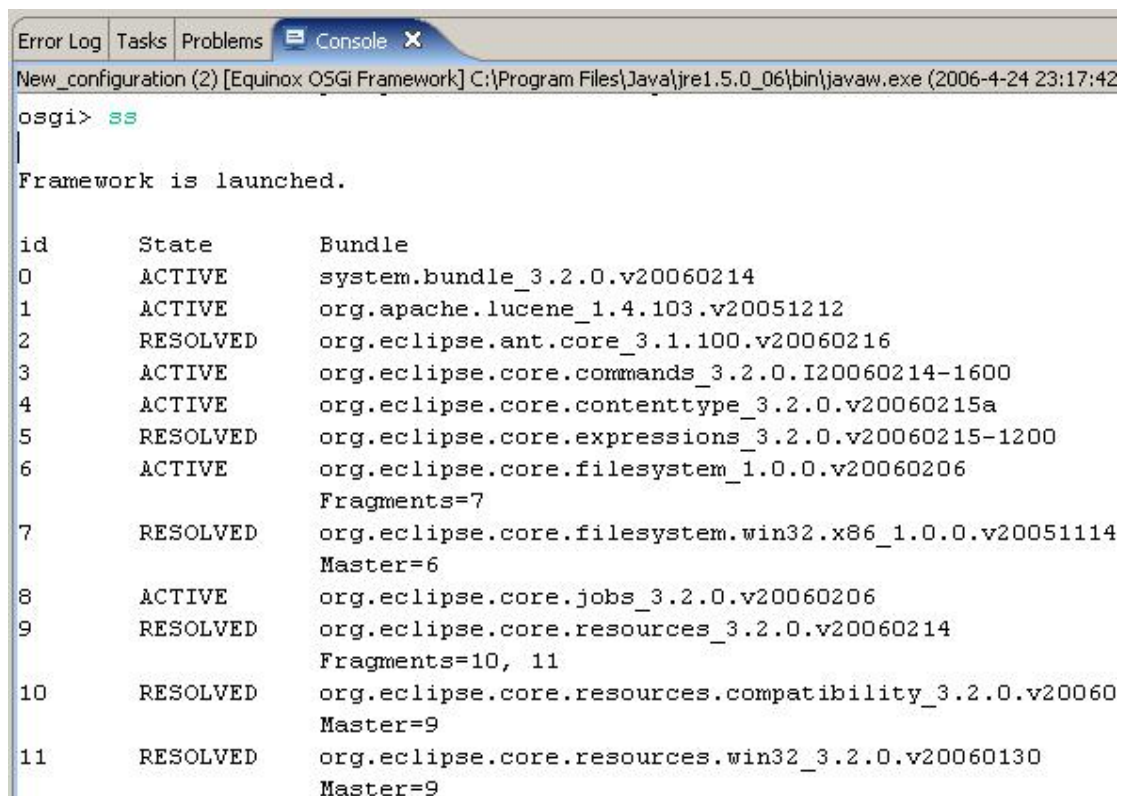


上面所打印的正是我们刚才在代码中所看到的 Hello World。

现在我们在控制台中输入“ss”命令：



我们会发现，控制台将所有在 osgi 下的 Bundle 都打印了出来，包括他们的 ID 以及目前的状态情况：



我们可以发现刚才我们创建的 test1 工程所处 ID 号是 43，状态是 ACTIVE。

现在我们将它卸载掉，输入命令 `uninstall 43`，我们会看到 Goodbye world 的文字显示出来，这就是我们在刚才创建的 Activator 中的 `stop` 方法所打印的文字：

```
41      RESOLVED      org.eclipse.update.c
                        Master=40
42      RESOLVED      org.eclipse.update.i
43      ACTIVE         test1_1.0.0

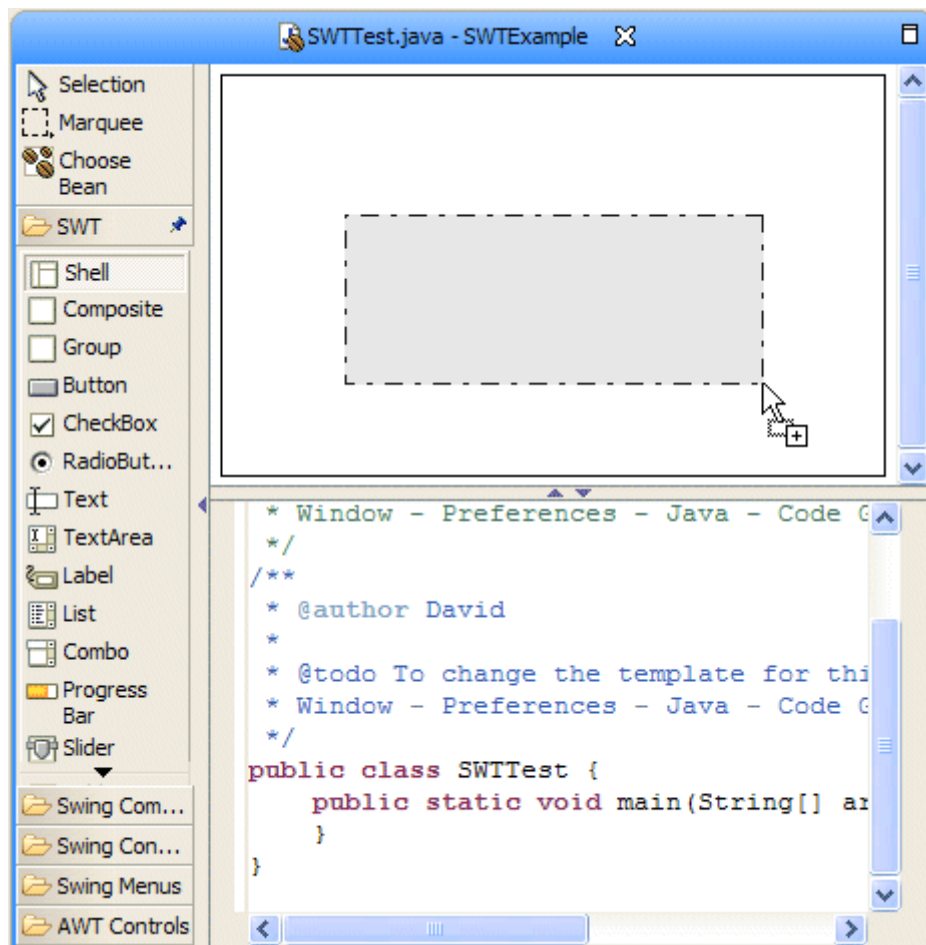
osgi> uninstall 43
Goodbye World!!

osgi>
```

四. 其他 Eclipse RCP 相关技术介绍

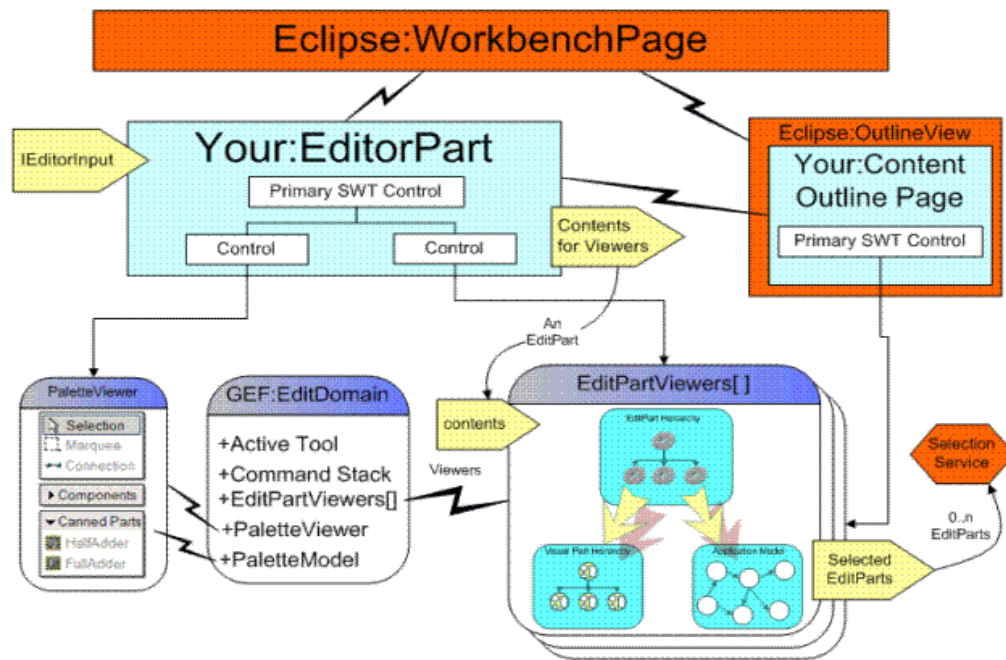
1.GEF/Draw2d

GEF(Graphical Editing Framework)是 Eclipse旗下的强有力的Tool Project，利用 GEF可以轻松实现类似于Visual Editor的可视化图形编辑应用程序。



1.1 什么是 GEF, EditPartViewer

上面已经说过了, GEF 是 Eclipse 旗下的 Tool Project, 为开发者提供了一个图形编辑开发平台。GEF 是一个很出色的 MVC 平台, 见下图:



EditPartViewer 是用于承载图形的, 就像一张画布一样, 但是并不是直接讲图像绘制在其之上, 而是通过 GEF 的 **EditPart** 来获得的图形, 这里只是介绍一下, 下面会进行讲解, 但是涉及到 **EditPartViewer** 本身的东西比较少。

1.2 Model , EditPart , View

模型 (model) 是开发者自己根据自身程序的要求而定制的, 它包括了开发者所想在程序中展现的一些信息, 比如我们创建数据库编辑工具, 那么我们的模型中就会有数据表 (DataTable) 这么一个模型, 一个完整的数据表就会拥有自己的表名 (name)、列 (Column)、元组 (Row)、主键 (Key) 等属性。这些属性对于一个数据表来说是必须的, 只有这些属性才能完整地去描述一个数据表。

此外, 在 GEF 中, 模型有时候还需要一些附属属性, 我们姑且这些属性为“图形属性”, 这些属性和模型本身是没有什么关系的, 只是为了反应模型对应的视图 (View) 的一些变化而

设置的，比如坐标、尺寸等。

模型的变化需要反应在视图上，典型的 MVC 是通过控制器来改变视图的，在 GEF 中称控制器为“编辑单元”（EditPart），编辑单元和模型是一对一的关系。

编辑单元是架起模型是试图的一道桥梁，模型的改变会触发事件，让 EditPart 更新目前的视图，视图会根据模型的属性改变而做出响应的调整，当然这些调整都是开发人员自己规定的了。简单的做法是，在模型中设置一个属性更改发起源：`java.bean.PropertySource`，然后让编辑单元(EditPart)实现监听器：`java.bean.PropertyChangeListener`，这样一来，每当模型变化的时候，就会发出变化通知，编辑单元得到通知后，刷新视图。但是这种做法不是很好，一般情况下，开发人员都应该利用 EMF（Eclipse 旗下的另一个 Tool project）进行建模，然后利用 EMF 模型的属性通知功能来实现事件通知以及响应。

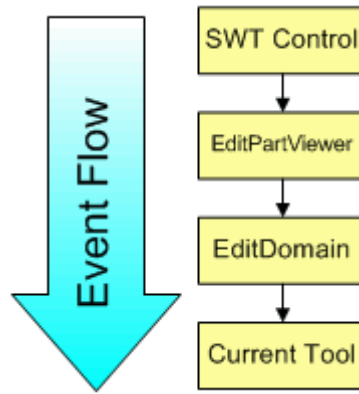
1.3 Request 和 Policy

视图变化发出的事件，其实是开发人员所不能控制的，我这么说并不是指无法控制，而是想引出下一个问题：Request 和 Policy

其实 GEF 的 MVC 模式中，在利用控制器去改变模型的时候，利用了策略模式（Policy）。这里需要说一下 GEF 中如何进行对图形变化的事件处理。

在 GEF 构架中，图形的变化并不是像通常的控件那样，利用添加监听器来回调函数，它是把这些事件都封装成了一个个的 Request 向外发送，然后让 EditPart 来截获处理。维护这些事件，并将他们封装成 Request 的是一个叫 EditDomain 的类，它专门负责去获得获得画布上得事件，然后利用 Tool 封装这些事件。

EditPart 处理 Request 是通过 EditPolicy 以及自身方法来处理得。举个例子：当我们在图形上做了一个点击操作，点了一下某一个图形，这时候，EditPartViewer 就会截获事件，传给 EditDomain，然后将信息封装生成一个 Reuquest,传递到 EditPart 那里，让它处理。下面是事件方向：

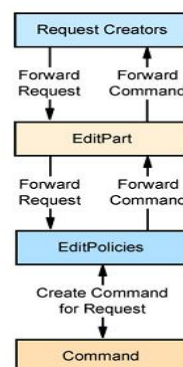


如果我们点击后要高亮显示这个图片，或者是在图片四周形成一个黑色的边框，这就是 Policy 在预处理这个事件。

Policy 能够对这些 Request 进行一些基本的处理，比如上述的那样，一些 Policy 能够处理当我们选中一个图形后所产生的高亮效果等，不仅仅如此，还比如拖动图形、拉伸图形以及绘制图形等等操作，都是由 Policy 来处理的。

Policy 是被 EditPart“安装”到自己身上的，利用 EditPart 的 `installEditPartPolicy` 方法，将一些相关的 Policy“安装”好，好让他们处理 Request。EditPart 所需要安装的 Policy 是有不同的，这是由于 Request 有多种“角色”，不同的角色是不同事件封装的结果，只有对应的 Policy 才能对他们处理，Policy 和 Request 是多一多的关系，不同的 Policy 能够处理同一个 Request。

但是 Policy 却又什么都不能做，这是由于很多代码还是需要程序员来写的，Policy 只处理一些基本东西（上面提到的现实黑色边框什么的），然后它又会通过 Request“索取”Command，然后再执行 Command。见下图：



1.4 Command 和 CommandStack

从上图可以看到有一个名为 Command 的类，下面介绍一下。

Command，顾名思义，方法，它是一个简单的类，需要开发人员去实现它的一些方法，最常用的方法有：execute, redo, undo。这三个方法分别是执行、重新执行、取消执行，Policy 索取到 Command 后会去执行它的 execute 方法，所以说开发人员必须清楚地去理解 Policy 所要 Command 的含义，以及触发的时机。

两个方法 redo, undo 是为了实现取消、回滚操作而设定的方法。说到 Command，就不能提到 CommandStack（方法栈），它维护了一系列的 Command，从而实现了回滚、前进等操作，而这个方法栈又是由 Domain 进行维护的。

1.5 Draw2D

Draw2D 是在 SWT 之上的，它是一套方便开发人员使用的绘制工具包。

我们上面章节已经说过了，SWT 绘制是通过 GC 来做的，Draw2D 是将绘制图形的过程重新封装成了一套容易维护的类，让开发人员能方便地进行图形开发。

GEF 的图形实现就是利用了 Draw2D。这里不做太多介绍。

2.GMF

Eclipse图形建模构架(Graphical Modeling Framework)是在 Eclipse Technology Project下的开源项目。

Eclipse Modeling Framework (EMF) 以及 Graphical Editing Framework (GEF) 是 Eclipse Tools Project中重要的两个项目，Eclipse Tools Project在许多Eclipse项目以及许多基于Eclipse应用软件有成功的应用。这些被普遍的应用在为EMF表述的领域模型的可视化设计上。尽管存在如何连接两种技术的例子（例如：Eclipse Development using the Graphical Editing Framework and the Eclipse M

odeling Framework, eDiagram GEF Sample),所需要的是一个已经生成的基础结构去简化他们在Eclipse中的建模应用。

例如:UML2 项目是一个基于EMF并且为Eclipse提供了OMG's UML 2.0 元数据模型的实现,但是普遍缺乏图形化功能。这和其他基于EMF的模型能够从一个普遍的基础模型中受益去简化基于GEF的需要可视化编辑组件的图形化开发。

GMF 能够提供为在 Eclipse 中图形化开发以及建模的基础构架和组件。本质上,GMF 能在当工程可能会移出对 EMF 以及/或者 GEF 的依赖时,在 EMF 和 GEF 中形成一座桥梁。

这个项目能为在 EMF 中的需要图形化编辑的领域模型添加图形化编辑功能提供已生成的方法。在许多情况下,GMF 是用于扩展 EMF 功能的。GMF 有以下三个组件组成:

- 图形化基础构架
- 图形生成构架
- 模拟建模工具

图形定义由提供的工具包和基于GMF提供的元数据模型生成。尽管在工程确认阶段alternatives会被检查,元数据的入口点仍是OMG's UML 2.0 Diagram Interchange Specification。图形定义及相应的模型用于为需要可视化设计的应用中生成图形。图形定义本身将被运用GMF生成的外表建模。

图形基础构架会为生成器组件提供一套基础组件和服务。该基础构件能提供图形元素(点、边框、连线),工具箱,视图,编辑器,导航,强制约束等。GMF 需要 EMF 插件等为运行基础。而生成什么及与之相对的运行构架所提供的之间的平衡将在项目的确认阶段也会被检查。