

ChinaPub 在线购买: <http://www.china-pub.com/computers/common/info.asp?id=36806>

CSDN 在线阅读: <http://book.csdn.net/bookfiles/550/>

CowNew 开源团队: <http://www.cownew.com>

内容介绍:

本书系统地介绍了 **SWT**、**Draw2D**、**GEF**、**JET** 等与 **Eclipse** 插件开发相关的基础知识，并且以实际的开发案例来演示这些知识的实战性应用，通过对这些实际开发案例的学习，读者可以非常轻松地掌握 **Eclipse** 插件开发的技能，从而开发出满足个性化需求的插件。

本书以一个简单而实用的枚举生成器作为入门案例，通过该案例读者能学习到扩展点、**SWT**、**JET** 等 **Eclipse** 插件开发的基本技能；接着对 **Eclipse** 插件开发中的基础知识进行了介绍，并且对属性视图的使用做了重点介绍；最后以两个具有一定复杂程度的插件(**Hibernate** 建模工具和界面设计器)为案例介绍了 **SWT**、**Draw2D**、**GEF**、**JET** 等技术的综合运用。

第 1 章 Eclipse 插件.... 1

1.1 插件的安装... 1

 1.1.1 直接复制安装... 1

 1.1.2 links 安装方式... 2

 1.1.3 Eclipse 在线安装方式... 3

1.2 内置 JUnit 插件的使用... 5

1.3 可视化 GUI 设计插件

 ——Visual Editor 9

 1.3.1 Visual Editor 的安装... 9

 1.3.2 一个登录界面的开发... 10

1.4 Eclipse 的反编译插件... 21

- 1.4.1 为什么要反编译... 21
- 1.4.2 常用 Java 反编译器... 22
- 1.4.3 反编译不完全的代码的
查看... 23

1.5 WTP 插件使用... 26

第 2 章 Eclipse 插件开发.... 30

- 2.1 Eclipse 插件开发介绍... 30
 - 2.1.1 开发插件的步骤... 30
 - 2.1.2 Eclipse 插件开发学习资源的
取得... 31
- 2.2 简单的案例插件功能描述... 31
- 2.3 插件项目的建立.... 33
 - 2.3.1 建立项目... 33
 - 2.3.2 以调试方式运行插件项目... 38
- 2.4 改造 `EnumGeneratorNewWizardPage`
类... 39
 - 2.4.1 修改构造函数... 39
 - 2.4.2 修改 `createControl` 方法... 40
 - 2.4.3 修改 `initialize` 方法... 41
 - 2.4.4 修改 `handleBrowse` 方法... 46
 - 2.4.5 修改 `dialogChanged` 方法... 49
 - 2.4.6 分析 `updateStatus` 方法... 50
 - 2.4.7 取得界面控件值的方法... 51
- 2.5 开发枚举项编辑向导页... 51
 - 2.5.1 初始化... 53
 - 2.5.2 相关环境数据的处理... 54
 - 2.5.3 代码生成... 54
- 2.6 编写代码生成器... 57
- 2.7 功能演示、打包安装... 64

第 3 章 插件开发导航.... 68

- 3.1 程序界面的基础——SWT/JFace. 68
 - 3.1.1 SWT 的类库结构... 68
 - 3.1.2 SWT 中的资源管理... 70
 - 3.1.3 在非用户线程中访问
用户线程的 GUI 资源... 70
 - 3.1.4 访问对话框中的值... 72
 - 3.1.5 如何知道部件支持
哪些 style. 73
- 3.2 SWT 疑难点... 74
 - 3.2.1 Button 部件... 74
 - 3.2.2 Text 部件... 74
 - 3.2.3 Tray. 74
 - 3.2.4 Table. 74

3.2.5 在 SWT 中显示 AWT/Swing 对象... 75
3.3 异步作业调度... 76
3.4 对话框... 79
3.4.1 信息提示框... 79
3.4.2 值输入对话框... 80
3.4.3 错误对话框... 81
3.4.4 颜色选择对话框... 82
3.4.5 字体对话框... 83
3.4.6 目录选择对话框... 83
3.4.7 文件选择对话框... 84
3.4.8 自定义对话框及配置保存与加载... 85
3.5 首选项... 86
3.6 Eclipse 资源 API 和文件系统... 88
3.6.1 资源相关接口的常见方法... 89
3.6.2 方法中 <code>force</code> 参数的意义... 91
3.6.3 资源相关接口的方法使用示例... 91
3.6.4 在 Eclipse 中没有当前项目... 92
3.7 Java 项目模型... 92
3.7.1 类结构... 92
3.7.2 常用工具类... 94
3.7.3 常用技巧... 95
3.7.4 设定构建路径实战... 100
3.7.5 如何研读 JDT 代码... 105
3.8 插件开发常见的问题... 106
3.8.1 <code>InvocationTargetException</code> 异常的处理... 106
3.8.2 <code>Adaptable</code> 与 <code>Extension</code> <code>Object/Interface</code> 模式... 107
3.8.3 千万不要使用 <code>internal</code> 包... 111
3.8.4 打开视图... 111
3.8.5 查找扩展点的实现插件... 111
3.8.6 项目 <code>nature</code> . 111
3.8.7 透视图开发... 112
3.8.8 关于工具条路径... 113
3.8.9 Eclipse 的日志... 116
第 4 章 属性视图.... 117
4.1 基本使用... 117
4.1.1 <code>IPropertySource</code> 接口说明... 118
4.1.2 对象实现 <code>IPropertySource</code> 接口... 120
4.1.3 对象适配成 <code>IPropertySource</code> 对象... 125
4.2 属性视图高级话题... 128

4.2.1 属性分类...	128
4.2.2 复合属性...	133
4.2.3 常用属性编辑器...	140
4.2.4 自定义属性描述器...	146

第 5 章 开发 Hibernate 插件.... 154

5.1 功能描述...	154
5.2 XML 文件的处理...	158
5.2.1 XML 处理技术比较...	158
5.2.2 Dom4j 的使用...	159
5.2.3 XStream 的使用...	165
5.3 实体模型文件创建向导...	169
5.4 模型的定义和模型文件处理...	176
5.5 实体属性描述器...	187
5.6 实体编辑器...	193
5.6.1 字段的编辑...	193
5.6.2 编辑器基类...	200
5.6.3 实体编辑器核心配置界面...	203
5.6.4 多页实体编辑器...	224
5.7 代码生成...	228
5.7.1 代码生成器接口...	228
5.7.2 代码生成器配置文件...	232
5.7.3 代码生成向导...	235
5.7.4 公共工具类 CommonUtils...	243
5.8 Hibernate 代码生成器...	245
5.8.1 命名策略...	246
5.8.2 工具类...	247
5.8.3 代码生成的 JET 代码...	251
5.9 CowNewStudio 使用实例...	259

第 6 章 基于 GEF 的界面设计工具.... 263

6.1 GEF 简介...	263
6.1.1 Draw2D...	263
6.1.2 请求与编辑策略...	264

6.1.3 视图与编辑器...	264
6.1.4 GEF 的工作过程...	265
6.2 系统需求...	265
6.2.1 界面设计工具的分类...	265
6.2.2 功能描述...	266
6.3 构建模型...	267
6.4 实现控制器...	275
6.4.1 窗体和组件的控制器...	275
6.4.2 编辑策略...	279
6.4.3 命令对象...	283
6.5 窗体文件创建向导...	287
6.6 组件加载器...	289
6.7 编辑器...	295
6.8 代码生成和构建器...	310
6.8.1 代码生成...	310
6.8.2 构建器...	313
6.8.3 为项目增加构建器...	320
6.9 实现常用组件...	323
6.9.1 标签组件...	323
6.9.2 按钮组件...	327
6.9.3 复选框...	331
6.9.4 编辑框...	336
6.9.5 列表框...	338
6.10 使用演示...	346

前　　言

Eclipse 是一款非常优秀的开源 IDE，非常适合 Java 开发，由于支持插件技术，受到了越来越多的开发者的欢迎。

作为一款优秀的平台，如果我们只是使用 Eclipse 的现有功能进行开发，无疑不能发挥出 Eclipse 的全部威力，如果能根据需要开发基于 Eclipse 的插件，那么将会大大提高开发效率。现在市场上已经有了几本 Eclipse 的相关书籍，但基本上都是偏重于 Eclipse 的使用，很少有涉及到基于 Eclipse 的插件开发的书籍，即使有讲述到 Eclipse 插件开发的，其内容也是浅尝辄止，根本没有对有一定复杂程度和实用性的插件开发进行讲解。

Eclipse 的插件体系是非常复杂的，学习门槛也非常高，为了帮助国内开发人员掌握 Eclipse 的插件开发技术，从而开发出自己的插件，本教程系统地从初学者到高手的每一个环节，并且通过实际的工程

案例来演示这些知识的实战性应用。

书中的对应的 Eclipse 版本为 Eclipse 3.2, 可以从 <http://www.eclipse.org> 网站免费下载。

本书内容安排:

第 1 章介绍常用的 Eclipse 插件的安装和使用。第 2 章以一个枚举生成器插件的开发为案例讲解一个简单、实用的插件的开发步骤。第 3 章介绍 Eclipse 插件开发中常用的基础知识。第 4 章介绍插件对属性视图的支持。第 5 章以 Hibernate 建模插件为案例讲解有一定复杂程度和实用性的插件的开发。第 6 章以界面设计器插件为案例讲解基于 GEF 技术的图形插件的开发。

如果您对本书有任何意见和建议, 可以发送邮件到 about521@163.com, 本书相关的后续资料将会发布到 CowNew 开源团队网站(<http://www.cownew.com>)中。

杨中科

序言

“自己动手写开发工具”是很多开发人员的梦想, 虽然市场上已经有了各种开发工具, 但是在一些情况下还是有编写自己开发工具的需求的:

- | 使用的编程语言没有合适的开发工具。比如在 Eclipse 出现之前, Python、Ruby、JavaScript 等语言都没有很好的全面支持代码编写、调试以及重构的开发工具, 使用这些语言进行开发非常麻烦。
- | 为自己开发的语言配备开发工具。有时我们会开发一款新的开发语言, 为了方便使用, 我们也需要为其提供相应的开发工具。
- | 为控件库、框架等提供开发工具。Echo2、Tapestry、Spring 等都是非常优秀的产品, 但是通过手工编码的方式使用这些产品仍然是非常麻烦的, 如果能配备图形化的开发工具, 那么通过简单地鼠标拖拽就可以快速完成工作。
- | 为产品提供二次开发工具。很多公司都有自己的产品, 而这些产品一般都提供了二次开发的能力, 开发人员只要进行少量的编码或者配置就可以很轻松的实现针对特定客户的个性化功能。由于二次开发人员的技术水平相对较差, 如果能提供一个图形化的二次开发工具也必将提高二次开发的效率及质量。

对于上面的几种情况, 已经有很多开发人员探索着实现了, 比如 Boa Constructor 就是一款用 Python 语言编写的 Python 开发工具, 润乾报表提供了用 Swing 技术实现的报表设计器。这种所有功能全盘自己实现的方式有如下的缺点:

- | 必须自己处理所有的细节问题。比如实现一个语言的开发工具就必须自己处理语法高亮、语法分析、代码提示、调试、重构、可视化的界面编辑器以及代码生成等, 这些问题的处理对开发人员的要求非常高, 而且开发工作量也非常大。
- | 各个工具的差异性非常大, 增加了用户的学习成本。
- | 不同的工具之间的集成非常困难。由于不同的工具是由各个厂商独立开发出来的, 互相之间的集成非常麻烦, 不仅使用的时候需要运行多个工具, 而且经常需要在多个实现相似功能的工具之间做出取舍。

Delphi、VS.Net Studio、JBuilder、NetBeans 等都提供了一定的扩展机制, 我们只要按照要求编写插件就能在这些工具中开发扩展功能, 但是这些工具提供的扩展功能是非常简单和有限的, 我们几乎无法完成编写开发工具这样复杂的功能。

做为 IDE 界的一匹黑马, Eclipse 在几年内异军突起, 很多开源项目或者商业化的产品都提供了相应的 Eclipse 插件, 比如 Echo2、GWT、Struts 等开源产品以及 IBM Websphere、Crystal、金蝶、普元等商业公司的开发工具都基于 Eclipse 进行开发, 甚至 Borland 也将新版本的 JBuilder 移植到 Eclipse 上。Eclipse 能够得到这么多厂商的支持, 究其原因有如下几点: 免费且开源; 开放性; 可扩展性强; 对开发工具的开发提供了强大的支持; 基于 Eclipse 的产品更专业; 各种插件可以组合使用。

免费且开源

大多数开发工具都是按用户数收费的, 对于开发人员比较多的公司来说开发工具的支出是一笔不小的费用, 而且基于这些开发工具开发出来的扩展插件在发布的时候也会涉及到授权的问题。Eclipse 是免费使用的, 这样就为公司节省了不小的一笔开支, 而且只要遵守 EPL 协议, 那么基于 Eclipse 开发的扩展插件可以任意发布。Eclipse 是开源的, 通过研读 Eclipse 的代码, 我们能更快的开发出高质量的插件。

开放性

Eclipse 并没有局限于 Java 语言，我们可以开发非 Java 语言的开发插件，比如 Ruby、Python、C/C++、C#以及 PHP 等语言都有了 Eclipse 上的开发插件。而且 Eclipse 也没有限定插件的应用领域，所以 Eclipse 成为了很多领域开发工具的基础，不仅 IBM、金蝶、普元等企业级系统开发商选择 Eclipse 做为其开发工具的基础，而且像风河系统公司、Accelerated 科技、Altera、TI 和 Xilinx 等嵌入式系统公司也将 Eclipse 平台作为自身开发工具的基础。

可扩展性强

Eclipse 采用微内核架构，其核心只有几兆大小，我们平时使用的代码编辑、调试、查找以及重构等功能都是以插件的形式提供的。我们不仅可以扩展现有插件，而且还可以提供扩展点，这样其他用户同样可以基于我们的插件开发扩展插件从而满足用户的个性化需求，这样我们只需要实现我们个性化功能即可，通用功能由基础插件来完成。比如我曾经开发过一个 Python 的远程调试插件，由于 PyDev 已经提供了本地调试的功能，所以我对 PyDev 进行了少量扩展开发就完成了这个插件。

对开发工具的开发提供了强大的支持

Eclipse 提供了新建向导、代码编辑、调试、运行、图形化界面以及代码生成等开发工具常见功能的支持，这大大简化了一个复杂开发工具的开发。只需数十行代码就可以实现语法高亮、代码提示等代码编辑功能、只需数百行代码就可以实现调试功能、只需数百行代码就可以实现一个所见即所得的图形化编辑器，这一切让开发一个专业的开发工具变得如此简单。这样厂商只要按照自己领域相关的逻辑进行定制，其他的基础功能则由 Eclipse 提供，这使得厂商能够把更多的精力投入到自己熟悉的业务领域。比如我们要开发一个 Python 的所见即所得的界面绘制工具，那么我们只需要基于 GEF 进行少量开发即可实现一个所见即所得的图形化编辑器，而生成的 Python 代码的编辑、调试以及重构等功能则由现有的 PyDev 插件来完成。可以想像如果没有 Eclipse 的话，我们从头开发一个 Python 的图形化编辑器需要我们处理多少的技术难题！

基于 Eclipse 的产品更专业

一个专业的开发工具通常需要考虑很多问题，比如需要考虑被选择对象的属性编辑方式、长时间操作的进度条展示、编辑窗口的布局方式以及工具选项的配置等问题，这些问题 Eclipse 的开发人员已经替我们考虑好了，我们开发的插件将自动拥有这些功能，这使得我们的插件显得更加专业。

各种插件可以组合使用

以前每开发一个开发工具，都需要实现代码版本控制等功能，而在 Eclipse 中则已经有了支持 VSS、CVS 和 SVN 等版本控制协议的插件，我们只要实现我们的开发工具即可，这些版本控制插件可以正交的和我们的插件组合使用，并且用户可以选择任何他们喜欢的版本控制插件，使得我们的工具使用起来更加灵活。

现在市场上已经有了 XML 编辑器、版本控制、UML 绘制工具及 EJB 开发工具等插件，并且这些插件也有不同的厂商实现的多个版本，这样用户可以随意挑选他们喜欢的插件，在同一个 Eclipse 环境中任意组合这些插件来完成复杂的功能。

Eclipse 的出现使得 IDE 市场出现了一个新的格局，主流的开发工具都开始向 Eclipse 靠拢，这不仅使得开发工具的开发变得更容易了，中小型企业甚至个人也能开发一个实用的开发工具出来。这些基于 Eclipse 的开发工具不仅能提高开发效率，而且将用户统一到 Eclipse 平台中，减少了用户的学习成本。相信基于 Eclipse 的插件开发将成为未来开发工具的主流，那么就让我们开始激动人心的 Eclipse 插件开发学习之旅吧！

第 2 章 Eclipse 插件开发

Eclipse 已经不仅仅是一个开发工具了，它更是一个平台。在 Eclipse 下开发的插件可以不仅限用于 Java 语言，现在已经出现了 C++、Python、C# 等语言的开发插件，而且我们还可以把它当成一个应用框架，用它

开发与编程无直接关系的系统，比如信息管理软件、制图软件、聊天软件等。不过目前国内大部分开发人员还仅仅是把 Eclipse 当成了一个开发工具来使用，没有发挥它的最大潜力。开发人员一直是在网上寻找相应功能的插件，一旦没有相应的插件或者插件安装失败就抱怨 Eclipse 没有 JBuilder 之类的工具强大。“工欲善其事，必先利其器”，Eclipse 的插件开发其实并不复杂，我们只要稍加学习，就能开发出满足我们个性化要求的插件，从而大大提高开发效率。学会 Eclipse 插件开发可在以下几个方面给人们带来方便：

- | 开发满足用户要求的插件。
- | 如果现有的开源第三方插件有一些 bug，我们也可以自己进行修改，而不必依赖于插件的开发者修补 bug。
- | 可以在第三方开源插件的基础上做二次开发，从而使其更能满足个性化要求。
- | 在使用一些 Eclipse 插件的时候如果出现问题我们也能更快地发现问题的所在，并快速排除问题。

本章我们将首先介绍插件开发的一些基础知识，然后就以一个具有实用价值的插件为例介绍插件开发的整个过程，学习完整个例子之后，我们就可以开发一些实用的插件了^①。

2.1 Eclipse 插件开发介绍

2.1.1 开发插件的步骤

开发一个插件需要如下几步。

- (1) 标识需要进行添加的扩展点以便与插件进行集成。
- (2) 根据扩展点的规范来编写扩展代码。
- (3) 编写 plugin.xml 文件描述正在提供的扩展以及对应的扩展代码。

如果要完全手动开发插件的话，我们需要自己去查询相应的扩展点、编写扩展点实现代码、编写 plugin.xml 等配置文件，这项工作非常麻烦、非常容易出错，而且插件的测试、部署也非常麻烦，为了简化插件开发，Eclipse 提供了一个用来开发插件的插件 PDE(Plug-in Development Environment)。

在 PDE 中，每个正在开发的插件都被单个的特殊的 Java 项目所代表，这个项目被称为插件项目。PDE 提供了一个插件创建向导，可以选择各种需要的部分创建插件项目。

在 PDE 中，插件配置工作大部分是在项目多页编辑器中完成的，这个编辑器简化了插件的配置和开发。插件 Manifest 编辑器用到 3 个文件： META-INF/MANIFEST.MF、plugin.xml 和 build.properties。它允许我们通过编辑所有必要的属性来描述一个插件，包括它的基本运行时要求、依赖项、扩展和扩展点等。PDE 提供了一个特别的启动配置类型，允许我们使用它的配置中包含的工作区插件来启动另外一个工作台实例(被称为运行时工作台)，我们可以像调试普通 Java 程序一样调试插件项目。

2.1.2 Eclipse 插件开发学习资源的取得

如果想快速地学习 Eclipse 插件开发的话，我们可以首先学习插件开发入门文档，通过入门项目的开发对 Eclipse 插件开发有一个感性的认识，然后再阅读 Eclipse 帮助文档中有关插件开发的部分(主要在 Platform plug-in Developer Guide、JDT Plug-in Developer Guide、Plug-in Development Environment Guide 这 3 个项目下)，以便对插件开发的知识有进一步了解，然后再阅读一些开源的 Eclipse 插件的源码，学习他人的代码，最后就可以根据自己的需要尝试进行插件的开发，这个过程经常是迭代的，比如在学习开源插件源码的时候还需要借助 Eclipse 的帮助文档来查询某个 API 的说明。

Eclipse 插件开发涉及到的知识是比较的，比如 SWT/JFace、JDTAPI、GEF、EMF 等，插件涉及到的类也非常多，容易使初学者望而生畏。其实 Eclipse 插件开发的学习是循序渐进的，没必要对各个方面都了解得非常透彻，比如如果不开发图形界面插件，那么就没必要学习 GEF，如果不开发模型相关的插件就没必要学习 EMF，即便是任何插件开发人员都躲不掉的 SWT/JFace 也没必要全部掌握，只要能用 SWT 编写一个简单的界面就可以，其他的东西可以一边做一边学。

2.2 简单的案例插件功能描述

在 Internet 搜索引擎中以“Eclipse 插件开发”为关键字搜索就可以找到数篇讲述 Eclipse 插件开发的经典入门文章，接着文章中的例子一步一步地做，就可以做出一个显示 Hello world 对话框的例子。这个例子虽然简单，但是却是一个真正的插件开发项目，通过这个项目我们就可以理解插件的工作原理以及配置文件各个配置项的作用。

在本节中我们来讲解一个有一定实用价值的简单插件，通过这个插件就可以对 Eclipse 的插件开发有更多的认识，并且可以立即应用到实际的开发中去。

JDK 1.5 提供的枚举大大简化了我们的开发工作，但是在有的情况下我们暂时还不能使用 JDK 1.5，那么此时如果需要枚举的话就只能自己写代码，比如：

```
public class CustomerTypeEnum
{
    private String type;

    public CustomerTypeEnum VIP = new CustomerTypeEnum( "VIP" );
    public CustomerTypeEnum MEMBER = new CustomerTypeEnum( "MEMBER" );
    public CustomerTypeEnum NORMAL = new CustomerTypeEnum( "NORMAL" );

    private CustomerTypeEnum( String type )
    {
        super();
        this.type = type;
    }

    public int hashCode()
    {
        final int PRIME = 31;
        int result = 1;
        result = PRIME * result + ((type == null) ? 0 : type.hashCode());
        return result;
    }

    public boolean equals( Object obj )
    {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        final CustomerTypeEnum other = (CustomerTypeEnum) obj;
        if (type == null)
        {
            if (other.type != null)
                return false;
        } else if ( !type.equals( other.type ) )
            return false;
        return true;
    }
}
```

如果需要很多枚举定义的话就会非常繁琐，因此我们下面就来开发这样一个代码生成器插件来简化这个操作。

2.3 插件项目的建立

2.3.1 建立项目

插件项目的建立与普通 Java 项目的建立类似。

(1) 在【包资源管理器】中右击，在弹出的快捷菜单中选择【新建】|【项目】命令，在对话框中选择【插件项目】选项，然后单击【下一步】按钮，在接下来的向导页中输入插件项目的名称、保存位置等信息，然后单击【下一步】按钮。在接下来的“插件内容”界面中进行如图 2.1 所示的配置。

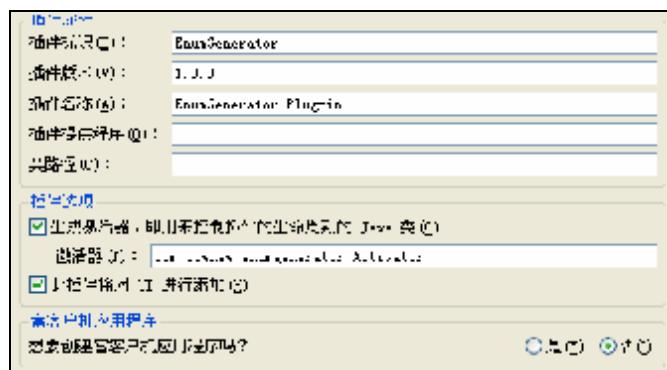


图 2.1 插件向导

(2) 设置完成后，单击【下一步】按钮，进入如图 2.2 所示的界面。

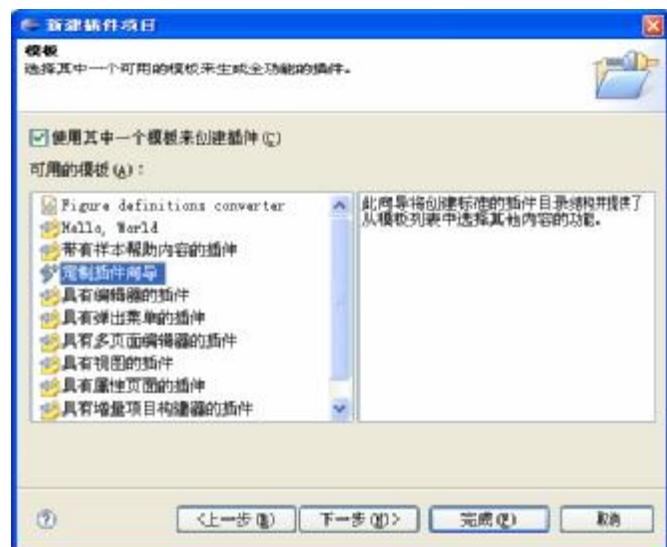


图 2.2 插件模板选择

(3) 选择【定制插件向导】选项，单击【下一步】按钮，在接下来的“选择模板”界面中，选中【新建文件向导】复选框，应取消选中其他复选框。单击【下一步】按钮，在“新向导选项”界面中进行如图 2.3 所示的配置后，单击【完成】按钮。

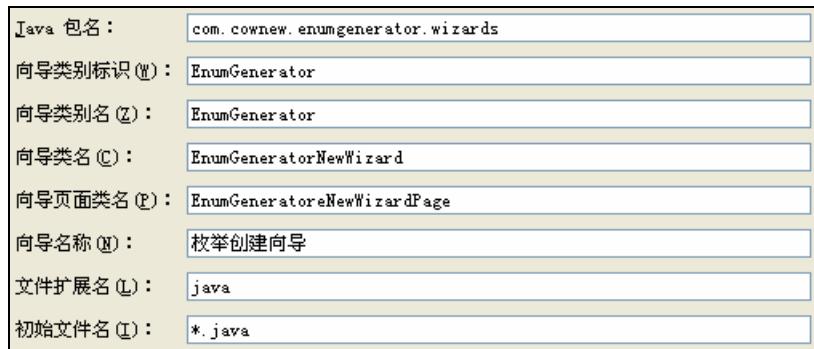


图 2.3 新建向导

(4) 完成以后的项目结构如图 2.4 所示。

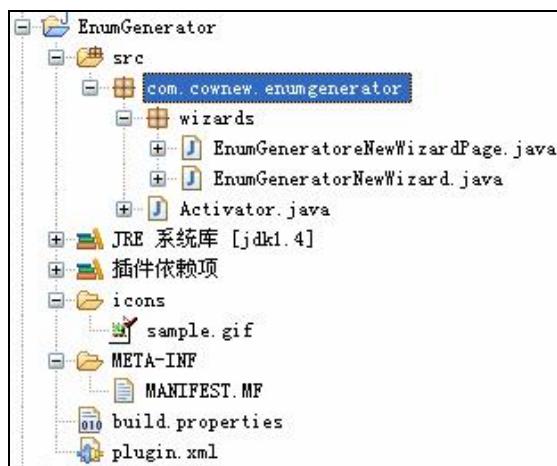


图 2.4 插件项目结构

下面看一下各个主要文件的作用。

(1) plugin.xml

这个文件是插件清单文件，它定义了此插件项目中所有的插件：

```

<?xml version="1.0" encoding="UTF-8"?>
<?Eclipse version="3.2"?>
<plugin>
    <extension
        point="org.eclipse.ui.newWizards">
        <category
            name="EnumGenerator"
            id="EnumGenerator">
        </category>
        <wizard
            name="枚举创建向导"
            icon="icons/sample.gif"
            category="EnumGenerator"
            class="com.cownew.enumgenerator.wizards.EnumGeneratorNewWizard"
            id="com.cownew.enumgenerator.wizards.EnumGeneratorNewWizard">
        </wizard>
    </extension>
</plugin>

```

标记 plugin 内可以定义多个 extension 标记，每个标记表示一个对扩展点的扩充，比如我们这里扩展的是 org.eclipse.ui.newWizards 扩展点，也就是“新建向导”扩展点；category 定义的是对这个扩展点的归类；wizard

导的图标, category 属性代表此向导的分类, 对向导分类可以使得向导看起来更清晰, 比如图 2.5 中的 CSS、HTML 等就属于 Web 分类下。

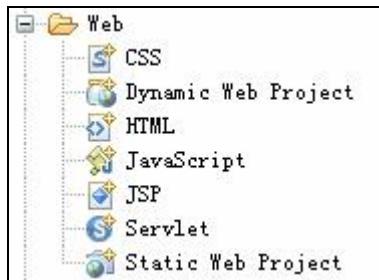


图 2.5 向导的分类

class 属性表示此扩展点对应的实现类, 大部分扩展点的实现都需要编写实现代码, 因此需要这个属性来指定此扩展点使用的是哪个类; id 属性定义的是此 wizard 的唯一标识, 作者的习惯是定义成和 class 一样, 这样一般就不会与其他插件的唯一标识冲突了。

需要注意 extension 标记内的 category、wizard 等标记是 “org.eclipse.ui.newWizards” 扩展点特有的标记, 也就是其他扩展点很可能没有这些标记。这些特有的标记是 Eclipse 供不同的扩展点用来进行属性定义的, 这样灵活性就更加好, 每种不同的插件的扩展点的标记定义格式都可以在 Eclipse 帮助文档中找到。

plugin.xml、build.properties 和 MANIFEST.MF 是插件项目中重要的配置文件, 共同配置了插件的不同方面的信息, 双击其中任何一个文件都会打开此项目的配置编辑器, 3 个文件的配置都在这同一个编辑器中完成, 如图 2.6 所示。

这个编辑器一共有 9 个选项卡, 分别是【概述】、【依赖项】、【运行时】、【扩展】、【扩展点】、【构建】、【MANIFEST.MF】、【plugin.xml】、【build.properties】。其中【概述】、【依赖项】、【运行时】中配置的是 “MANIFEST.MF” 文件的内容, 【扩展】、【扩展点】中配置的是 plugin.xml 文件中的内容, 【构建】配置的则是 build.properties 中的内容, 我们既可以在前面这些可视化编辑界面中进行配置, 也可以在【MANIFEST.MF】、【plugin.xml】、【build.properties】这几个选项卡中直接修改配置文件。建议尽量使用可视化编辑界面来进行配置, 这样可以减少很多错误。

在后面的章节中我们会介绍这些配置项的意义。



图 2.6 插件配置编辑器

(2) Activator.java

Activator.java 起着此插件的生命周期控制器的作用。

【代码 2-1】 插件生命周期控制器:

```
public class Activator extends AbstractUIPlugin {  
  
    // The plug-in ID  
    public static final String PLUGIN_ID = "EnumGenerator";
```

```

// The shared instance
private static Activator plugin;

public Activator() {
    plugin = this;
}

public void start(BundleContext context) throws Exception {
    super.start(context);
}

public void stop(BundleContext context) throws Exception {
    plugin = null;
    super.stop(context);
}

/**
 * Returns the shared instance
 * @return the shared instance
 */
public static Activator getDefault() {
    return plugin;
}

/**
 * Returns an image descriptor for the image file at the given
 * plug-in relative path
 *
 * @param path the path
 * @return the image descriptor
 */
public static ImageDescriptor getImageDescriptor(String path) {
    return imageDescriptorFromPlugin(PLUGIN_ID, path);
}
}

```

在 Eclipse 3.2 以前的版本中，此类习惯于被命名为**Plugin.java，它使用了单例模式，需要通过 getDefault 方法得到此类的实例。当插件被激活的时候 start 方法会被调用，插件可以在这个方法中编写初始化的代码，当插件被关闭的时候 stop 方法则会被调用。

类中还定义了一个静态方法 getImageDescriptor，用于得到插件目录下的图片资源。

在 Activator 的父类中还定义了很多有用的方法，在这里我们简要地列出常用的一些方法，在后面的章节中会展示这些方法的用途。

- | `getDialogSettings`: 得到对话框配置类实例。
- | `getPreferenceStore`: 得到首选项配置的储存类实例。
- | `getWorkbench`: 得到工作台。
- | `getLog`: 得到日志记录类实例。

(3) `EnumGeneratorNewWizard.java`、`EnumGeneratoreNewWizardPage.java`

这两个文件是向导界面的实现代码。为什么是两个文件呢？一个向导对话框通常有不止一个界面，因此整个向导和每个向导界面要分别由不同的类来维护。

`EnumGeneratorNewWizard` 是维护所有界面的向导类，在这个例子中只有一个界面，即 `EnumGeneratoreNewWizardPage`。

插件模板向导生成的 `EnumGeneratorNewWizard.java`、`EnumGeneratoreNewWizardPage.java` 这两个类文件并不能完全满足我们的要求，需要进行修改，因此这里暂时不讲解类中的代码。

2.3.2 以调试方式运行插件项目

在插件项目上右击，在弹出的快捷菜单中选择【调试方式】|【Eclipse 应用程序】命令(或运行时工作台)，Eclipse 就会启动另一个 Eclipse 实例。

在这个新启动的 Eclipse 中新建的一个 Java 项目，名称为 EnumGenTest，并创建一个源文件夹，在源文件夹下创建包 com.cownew.enumtest，在包 com.cownew.enumtest 上右击，选择【新建】|【其他】命令，这样就可以在 EnumGenerator 分组下看到刚才创建的“枚举创建向导”了，如图 2.7 所示。

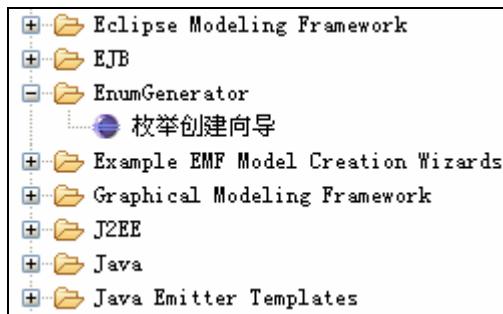


图 2.7 新建向导页

选择【枚举创建向导】选项，单击【下一步】按钮，进入如图 2.8 所示的界面。

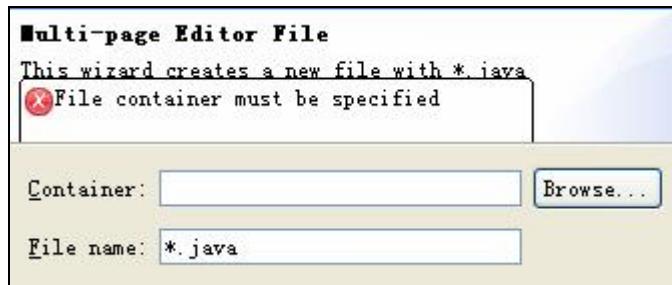


图 2.8 枚举创建向导

这个界面在类 `EnumGeneratoreNewWizardPage` 中定义。Container 用来指定生成的文件存放在哪个容器下边(容器是进行 Eclipse 插件开发时要弄懂的概念，这里不详细说明，可以暂时认为源文件夹、普通文件夹、项目根目录都是容器)，单击 `Browse` 按钮，在弹出的对话框中可以选择源文件夹、项目根目录，甚至还可以选择输出路径，显然这不满足我们的要求，因为我们生成的枚举类只能放在文件夹下，因此需要对其进行改造。枚举项目结构如图 2.9 所示。

选择 `src/com/cownew/enumtest`，然后单击【确定】按钮，在 `File name` 中输入 `TestEnum.java`，单击【完成】按钮，向导为 `TestEnum.java` 填充了如下内容“`This is the initial file contents for *.Java file that should be word-sorted in the Preview page of the multi-page editor`”。

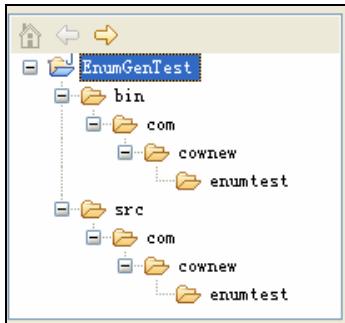


图 2.9 枚举项目结构

2.4 改造 **EnumGeneratoreNewWizardPage** 类

这个类有很多地方不能满足我们的要求，下面一一地进行修改。

2.4.1 修改构造函数

代码 2-2 是修改以后的 **EnumGeneratoreNewWizardPage** 类的构造函数。

【代码 2-2】修改以后的构造函数：

```
public EnumGeneratoreNewWizardPage(ISelection selection)
{
    super("wizardPage");
    setTitle("Multi-page Editor File");
    setDescription("This wizard creates a new file with *.Java
extension that can be opened by a multi-page editor.");
    this.selection = selection;
}
```

在构造函数中首先调用父类的构造函数，然后调用 `setTitle` 为向导页设定标题，调用 `setDescription` 方法为页设置描述信息，最后把代表用户选择项的 `ISelection` 赋值给 `selection` 私有变量。

向导的标题、向导页的标题、向导页的描述是不同的，如图 2.10 所示。



图 2.10 向导不同的提示显示

“向导的标题”是一个向导窗口的标题，这个标题在这个向导的生命周期内是不变的；“向导页的标题”是向导中每一页的标题，它标明了此页的作用，“向导页的描述”是对此页功能的进一步解释。

2.4.2 修改 createControl 方法

createControl 方法是 Override 父类的一个方法，框架会调用此方法绘制向导页界面。

【代码 2-3】修改后的 createControl 方法：

```
public void createControl(Composite parent)
{
    Composite container = new Composite(parent, SWT.NULL);
    GridLayout layout = new GridLayout();
    container.setLayout(layout);
    layout.numColumns = 3;
    layout.verticalSpacing = 9;
    Label label = new Label(container, SWT.NULL);
    label.setText("&Container:");

    containerText = new Text(container, SWT.BORDER | SWT.SINGLE);
    GridData gd = new GridData(GridData.FILL_HORIZONTAL);
    containerText.setLayoutData(gd);
    containerText.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e)
        {
            dialogChanged();
        }
    });
    Button button = new Button(container, SWT.PUSH);
    button.setText("Browse...");
    button.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e)
        {
            handleBrowse();
        }
    });
    label = new Label(container, SWT.NULL);
    label.setText("&File name:");

    fileText = new Text(container, SWT.BORDER | SWT.SINGLE);
    gd = new GridData(GridData.FILL_HORIZONTAL);
    fileText.setLayoutData(gd);
    fileText.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e)
        {
            dialogChanged();
        }
    });
    initialize();
    dialogChanged();
    setControl(container);
}
```

我们不能直接在 createControl 传递过来的 parent 上绘制界面，因为这个 parent 代表所有向导页面(包括向导选择页面)的父 Composite，如果直接在这个 parent 上绘制界面会导致界面显示混乱，所以要调用“Composite container = new Composite(parent, SWT.NULL);”创建一个新的 Composite。

接下来为这个 container 设定布局管理器，在插件项目中采用 GridLayout 是比较方便的。GridLayout 把界面分成网格，控件都放在这些网格中，与 Swing 的 GridLayout 不同的地方就是它不仅可以为每一列设定不同的宽度，而且还可以让一个控件占据多个网格。由于插件项目中的界面大部分都是规矩的表单式排列，因此采用 GridLayout 布局管理器是比较好的。因此这里采用 GridLayout 作为布局管理器，并设定其为 3 列。

创建标记控件、按钮控件、文本框控件等，并为它们设定合适的 GridData。需要注意的是每个控件的 GridData 不能与其他控件共享，也就是说即使两个控件使用的 GridData 全部一样，也要为它们各创建一个 GridData。

在创建控件的时候，给 containerText 控件和 fileText 都增加了 ModifyListener，当控件内容改变的时候，就会去调用 dialogChanged 方法。我们给【浏览】按钮增加了监听器，这样当按钮被按下的时候 handleBrowse 方法就会被调用以弹出容器选择对话框。

为了减少复杂性，这里指定 fileText 为不可手工编辑。接着调用 initialize 方法进行一些初始化操作，然后调用 dialogChanged 方法，最后调用 setControl 方法将 container 设定为本向导页的页面。

2.4.3 修改 initialize 方法

initialize 方法被 createControl 方法调用来初始化控件，为了适应需求，我们进行如下的修改。

【代码 2-4】 修改后的 initialize 方法：

```
private void initialize()
{
    if (selection != null && selection.isEmpty() == false
        && selection instanceof IStructuredSelection)
    {
        IStructuredSelection ssel = (IStructuredSelection) selection;
        if (ssel.size() > 1)
            return;
        Object obj = ssel.getFirstElement();
        if (obj instanceof IResource)
        {
            IContainer container;
            if (obj instanceof IContainer)
                container = (IContainer) obj;
            else
                container = ((IResource) obj).getParent();
            containerText.setText(container.getFullPath().toString());
        }
    }
    fileText.setText("*.Java");
}
```

这段代码用来把打开向导界面之前在【包资源管理器】视图中选中的对象的容器作为【容器】控件的初始值，并为 fileText 控件赋初值。

基础知识：

① IResource、IContainer、IFolder、IProject、IWorkspaceRoot、IFile

图 2.11 是这些接口的类型层次图(在 IResource 上右击，在弹出的快捷菜单中选择【打开类型层次结构】命令)。

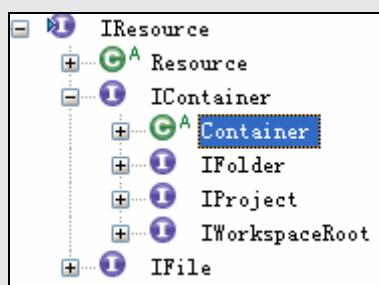


图 2.11 Eclipse 的资源类继承图

上面这个结构图就代表了 Eclipse 中的所有资源。其中 `IFolder` 代表文件夹, `IProject` 代表项目, `IWorkspaceRoot` 代表工作空间根目录。因为这 3 个资源都可以包含下级资源, 所以为它们抽象一个公共接口 `IContainer` 出来。与 `IContainer` 同级的 `IFile` 代表文件。为 `IFile` 和 `IContainer` 再抽取一个公共接口 `IResource` 出来。

② IStructuredSelection

被选择对象用 `ISelection` 来表示, 但是 `ISelection` 表达的内容太少, 因此继承一个子接口 `IStructuredSelection` 出来, 通过这个接口可以得到被选择的对象, 而且支持多选。

在 `initialize` 方法中首先判断 `selection` 是不是不为空、是不是实现了 `IStructuredSelection` 接口。请注意, 如果在资源视图的某个节点上能通过右键菜单弹出这个向导的话那么 `selection` 一定实现了 `IStructuredSelection` 接口。如果选择了一个对象并且被选择的对象实现了 `IResource` 接口的话, 则继续判断。

如果被选择的对象实现了 `IContainer` 接口(即被选择的对象是文件夹、项目或者工作空间根目录)的话, 就调用这个 `IContainer` 的 `getFullPath` 方法把容器的路径填充到“容器”控件中。如果被选择对象没有实现 `IContainer` 接口, 也就是文件的话, 那么其父容器一定实现了 `IContainer` 接口, 所以这里把其父容器的 `FullPath` 填充到“容器”控件中。

这段代码需要被改造, 因为它的实现是把选择对象的容器路径赋值给控件, 而我们的要求是把被选择包的包名赋值给控件。

我们需要判断的是被选择的对象是不是 Java 包。在 JDT 中用 `IPackageFragment` 类代表的包, 它定义在 `org.eclipse.jdt.core` 下。在文件的 `import` 部分加入 “`import org.eclipse.jdt.core. IPackageFragment;`”, 我们会发现系统报错, 说找不到 `IPackageFragment` 接口。原来 `IPackageFragment` 所在的包没有被放到项目的类路径中。那么如何把包放到类路径中呢? 您也许会说, 找到相应的 jar 包, 然后加入到项目的构建路径中不就可以了吗?

这是初学插件开发的人员常犯的错, 他们经常会问这样的问题: 我做的插件在 Eclipse 中开发的时候没有编辑错误, 可是我为什么就运行不了呢?

这是因为 Eclipse 插件的类引用机制比较特别。要弄明白这个问题, 首先要弄清楚“插件依赖”的概念。

一个插件不依赖于其他插件是几乎不可能的事情, 只有“站在巨人的肩膀上”才能更快更好地做出一个有实用价值的插件。我们写程序时依赖的一些组件通常都是以 jar 包的形式提供的, 比如 XML 解析用的 dom4j、日志工具 log4j 等, 使用这些包的时候只要将 jar 包加入构建路径就可以了。Eclipse 插件则不同, Eclipse 插件是有生命周期的, 也就是说 Eclipse 插件的 jar 包(Eclipse 插件并不一定以 jar 包形式发布, 这里这样说是为了解释)是“活的”, 而普通 jar 包是“死的”。我们在开发插件的时候, 只要指定此插件依赖的插件的标识 `id` 即可, 无需知道此插件对应的 jar 包, Eclipse 会自动加载此插件对应的 jar 包。在插件工程中, 有个名字为“插件依赖项”的库列表, 如图 2.12 所示。

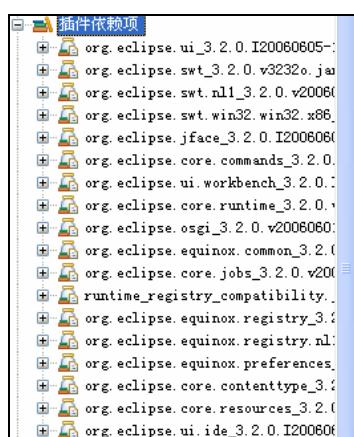


图 2.12 插件依赖项

这个列表是只读的，无法向其中加入内容，此库列表中的 jar 包是由 PDE 读取插件项目的依赖项以后根据依赖项的标识自动加载的，只是起到了方便开发的作用，在实际运行的时候并不一定会引用这些列表中所列的这些包，而且此工程拿到其他版本的 Eclipse 中打开的时候，这个列表中的内容也会随着变化。

那么如何在开发环境中添加依赖项呢？

双击 `plugin.xml` 打开插件配置编辑器，切换到【依赖性】页，单击【添加】按钮，如图 2.13 所示。



图 2.13 添加依赖项

在弹出的【选择插件】窗口中可以直接输入插件的标识 id，也可以在下方的插件列表中选择。那么如何知道我们依赖的插件的标识 id 呢？第一种方式就是老老实实地去查 Eclipse 帮助文档，另一种方式就是猜测。Eclipse 中标准插件的命名是很有规律的，每个不同的插件都放在不同的包中，此插件也以此包作为标识 id，这样就可以避免冲突。所以当我们要引用一个类的时候，只要尝试着将类的包路径输入【选择插件】文本框中，然后一级一级地排除包，直到有和下方的插件列表符合的为止。比如接口 `IPackageFragment` 在包 `org.eclipse.jdt.core` 下，我们在【依赖性】选项卡中单击【添加】按钮，在【选择插件】文本框中输入“`org.eclipse.jdt.core`”就立即可以看到有两个相符项了，如图 2.14 所示。



图 2.14 选择依赖插件

选择第一项，单击【确定】按钮，将此插件引入。再来查看“插件依赖项”，可以看到“`org.eclipse.jdt.core_***.jar`”已经被引入构建路径了，如图 2.15 所示。

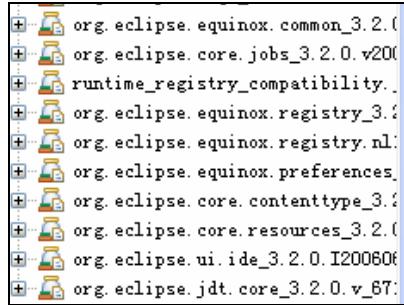


图 2.15 动态生成的依赖 Jar 包

编写如下代码：

```
if (obj instanceof IPackageFragment)
{
    IPackageFragment pckFragment = (IPackageFragment) obj;
    containerText.setText(pckFragment.getElementName());
}
```

运行插件项目，在一个包上右击，运行此向导，可以发现选中的包名被自动填到了【包】文本框中，如图 2.16 所示。

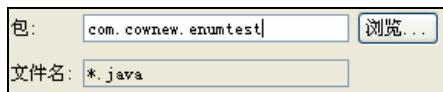


图 2.16 选择被创建文件所在的包

但是如果我们在一个 Java 文件夹上右击，运行向导的时候，Java 文件所在包的包名就不会自动填充到文本框中了。可以断定 Java 文件不是 IPackageFragment 类型了，可是到底是什么类型呢？查帮助文档？到网上搜索？到 BBS 中发帖提问？当然不用。这些都来得太慢了，程序员最不缺乏的就是探索精神。

以调试方式启动 Eclipse 插件工程，在“if (obj instanceof IPackageFragment)”一句处设置断点，在被调试的 Eclipse 中的工程下选择一个 Java 文件，右击，选择“枚举创建向导”，单击【下一步】按钮，此时程序就会在刚才添加的断点处暂停。选择变量“obj”，右击，在弹出的快捷菜单中选择【检查】命令，此时就会显示出此变量的类型等信息，如图 2.17 所示。

从图中可以看出 obj 的类型是 CompilationUnit。

根据实验结果来完善代码，添加如下代码：

```
else if (obj instanceof CompilationUnit)
{
    CompilationUnit cu = (CompilationUnit) obj;
    containerText.setText(cu.getParent().getElementName());
}
```

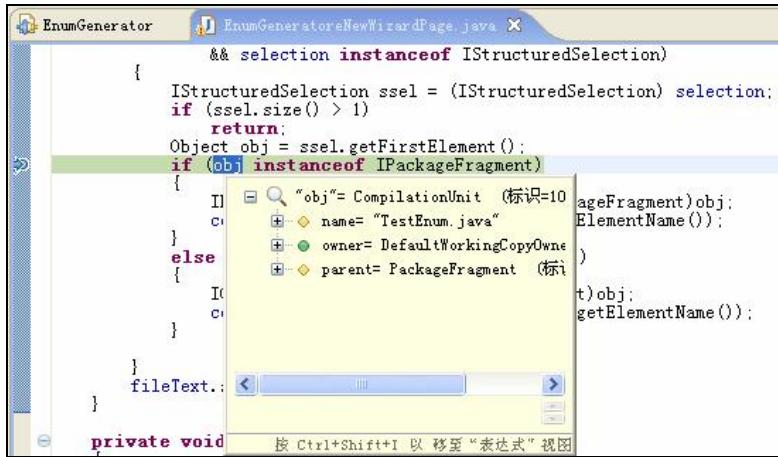


图 2.17 调试视图中查看变量类型

这里这样写是没有错误的，但是在 Eclipse 的插件开发中要尽量基于接口编程，`ICompilationUnit` 是实现了 `ICompilationUnit` 接口的实现类，因此我们要修改代码如下：

```
else if(obj instanceof ICompilationUnit)
{
    ICompilationUnit cu =(ICompilationUnit)obj;
    //cu 的父容器一定是一个包，所以直接通过 cu.getParent() 得到 Java
    //文件所在的包
    containerText.setText(cu.getParent().getElementName());
}
```

`ICompilationUnit` 在两个地方都有定义，分别是“`org.eclipse.jdt.core`”包下和“`org.eclipse.jdt.internal.compiler.env`”包下的，要记住这里使用的是第一个。

2.4.4 修改 handleBrowse 方法

当用户单击【浏览】按钮的时候，`handleBrowse` 方法会被调用以弹出一个容器选择对话框，并把用户选择的值填充到文本框中。这是不符合我们的需求的，我们的需求是弹出一个包选择对话框，并把用户选择的值填充到文本框中。

问题的焦点就集中到了如何弹出一个包选择对话框了。如果要自己实现的话需要处理的问题就太多了，好在 JDT 为我们提供了一个工具类 `org.eclipse.jdt.ui.JavaUI`，这个工具类中有一个静态方法 `createPackageDialog`，调用这个方法就可以创建一个包选择对话框。通过查询帮助文档得知 `JavaUI` 定义在插件“`org.eclipse.jdt.ui`”中，因此在使用这个类之前，首先要将“`org.eclipse.jdt.ui`”加入插件的依赖项。

`createPackageDialog` 方法有 4 个重载方法，因为我们要创建一个可以选择工程中所有包的包选择对话框，我们最终敲定使用：

```
public static SelectionDialog createPackageDialog(Shell parent, IJavaProject project, int
style)
```

我们可以把当前向导页的 `Shell` 传递给第 1 个参数，第 3 个参数传递一个风格就可以。难点在第 2 个参数，因为它要我们传递要对哪个 Java 项目创建对话框，这个 `IJavaProject` 接口就代表了一个 Java 项目。

我们现在运行的类是在一个界面中，因此取得外界信息的唯一方式就是构造函数传递过来的 `ISelection`。我们在一个 Java 项目中启动“枚举创建向导”之前，一般都会选中项目中的某些元素，比如包、Java 文件、项目等，我们只能尝试通过它们去突破了。打开熟悉的 `ICompilationUnit` 接口，来看看它有哪些方法。

在 `ICompilationUnit` 接口源码中，按 `Ctrl+O` 快捷键打开此类的类型成员，如图 2.18 所示。

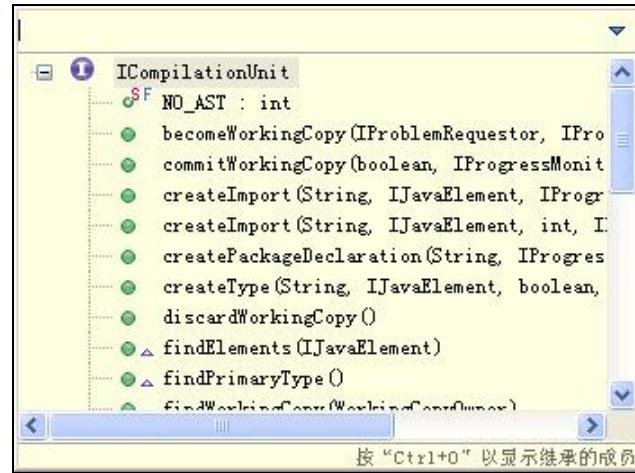


图 2.18 ICompilationUnit 的成员

保持这个提示框不关闭，再次按 Ctrl+O 快捷键打开此类的所有继承的成员，如图 2.19 所示。

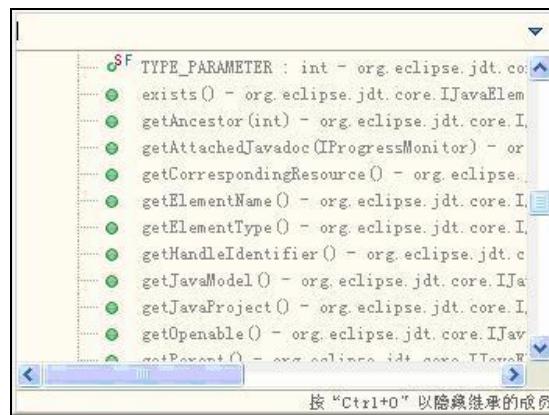


图 2.19 ICompilationUnit 以及父接口的成员

在这个列表中发现了我们要寻找的目标：`getJavaProject`，单击此方法就会转到此方法的声明处，原来这个方法定义在 `IJavaElement` 接口中。`IJavaElement` 是 Java 工程中所有 Java 特有元素(包、源文件、Java 工程等)的基础接口。选择 `IJavaElement`，右击，在弹出的快捷菜单中选择【打开类型层次结构】命令，如图 2.20 所示显示出了 `IJavaElement` 的类体系。

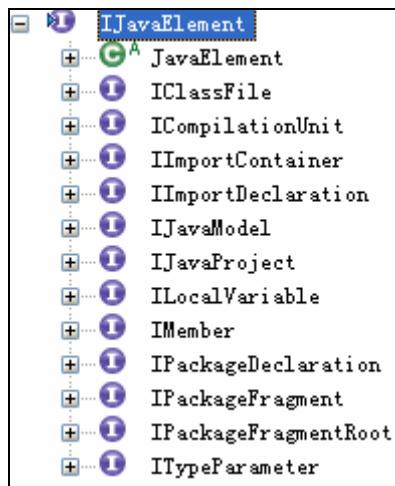


图 2.20 `IJavaElement` 的类继承图

根据名字可以看出：`IClassFile` 表示.class 文件，`IJavaProject` 代表 Java 工程等，这些就代表了所有 Java 项目中能选择的元素。

经过一番简单分析，得到当前 Java 项目的方法完成。

【代码 2-5】 得到当前 Java 项目：

```
private IJavaProject getCurrentJavaProject()
{
    if (selection != null && selection.isEmpty() == false
        && selection instanceof IStructuredSelection)
    {
        IStructuredSelection ssel = (IStructuredSelection) selection;
        Object obj = ssel.getFirstElement();
        if (obj instanceof IJavaElement)
        {
            return ((IJavaElement) obj).getJavaProject();
        }
    }
    return null;
}
```

其他的功能都好实现，我们可以直接参考已完成的 `handleBrowse` 方法。

【代码 2-6】 `handleBrowse` 方法：

```
private void handleBrowse()
{
    IJavaProject JavaProject = getCurrentJavaProject();
    if (JavaProject == null)
    {
        MessageDialog.openWarning(getShell(), "error",
            "请在 Java 项目内运行此向导！");
    }

    SelectionDialog dialog = null;
    try
    {
        dialog = new SelectionDialog(getShell(), JavaProject);
        dialog.open();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

```

        dialog = JavaUI.createPackageDialog(getShell(), JavaProject,
                IJavaElementSearchConstants.CONSIDER_REQUIRED_PROJECTS);
    } catch (JavaModelException e1)
    {
        MessageDialog.openWarning(getShell(), "error", e1.getMessage());
        e1.printStackTrace();
    }
    if (dialog.open() != Window.OK)
    {
        return;
    }
    IPackageFragment pck = (IPackageFragment) dialog.getResult()[0];
    if (pck != null)
    {
        containerText.setText(pck.getElementName());
    }
}

```

2.4.5 修改 dialogChanged 方法

当两个文本框中的内容发生变化的时候就调用 `dialogChanged` 方法，在这个方法中校验界面控件状态是否合法，并给出提示信息。

【代码 2-7】 `dialogChanged` 方法:

```

private void dialogChanged()
{
    String pckName = packageText.getText();
    String fileName = fileText.getText();

    if (pckName==null||pckName.length() == 0)
    {
        updateStatus("请指定包");
        return;
    }

    if (fileName==null||fileName.length() == 0)
    {
        updateStatus("请输入文件名");
        return;
    }
    if (fileName.replace('\\', '/').indexOf('/', 1) > 0)
    {
        updateStatus("文件名不合法");
        return;
    }
    int dotLoc = fileName.lastIndexOf('.');
    if (dotLoc != -1)
    {
        String ext = fileName.substring(dotLoc + 1);
        if (ext.equalsIgnoreCase("Java") == false)
        {
            updateStatus("文件扩展名必须是 \"Java\"");
            return;
        }
    }
    updateStatus(null);
}

```

由于包文本框是不可手工编辑的，所以此处忽略了对包文本框的合法性校验。判断文件名是否以 .java 结尾时，最好使用正则表达式，不过此处为了方便我们就沿用了自动生成的代码。

2.4.6 分析 updateStatus 方法

【代码 2-8】 updateStatus 方法：

```
private void updateStatus(String message)
{
    setErrorMessage(message);
    setPageComplete(message == null);

}
```

这个方法非常简单，就是把传过来的校验信息显示出来，并决定【下一步】按钮是否可用。调用 setErrorMessage 方法设置向导页的错误信息，而 setPageComplete 方法用来设置【下一步】按钮是否可用。

2.4.7 取得界面控件值的方法

【代码 2-9】取得界面控件值：

```
public String getPackageName()
{
    return packageText.getText();
}

public String getFileName()
{
    return fileText.getText();
}
```

2.5 开发枚举项编辑向导页

这个插件的向导目前只能设定文件保存在哪个包下、文件名是什么，我们还缺少一个指定这个枚举类有哪些项的向导界面。

因为本节的目的在于快速帮助我们掌握一个简单的实用插件的开发，因此将此向导页简化：整个向导页只有一个多行文本框，用户在多行文本框中每行输入一个枚举项的名字就可以；我们对输入文本框的内容的校验也做简化，只校验内容是否为空及各项是否重复，不校验枚举项的命名是否合法。

(1) 首先创建向导页类

所有的向导页类都直接或间接地从 `org.eclipse.jface.wizard.WizardPage` 继承，因此我们新建一个从 `WizardPage` 继承的 `EnumGenItemDefWizardPage` 类。

(2) 画界面

这个界面非常简单，只有一个多行文本框，并且此多行文本框充满整个页面。因此我们对此页面采用 `FormLayout` 页面布局管理器。`FormLayout` 是 `SWT 2.0` 中新增加的布局管理器，在其中可以设定控件与容器以及容器与容器之间的“附着关系”，这样界面控件就会随着界面大小的变化自动伸展，从而不会造成控件之间的错位。

【代码 2-10】枚举项编辑向导页界面初始化：

```
public void createControl(Composite parent)
{
    Composite container = new Composite(parent, SWT.NULL);
    FormLayout layout = new FormLayout();
    container.setLayout(layout);

    FormData data = new FormData();
    data.top = new FormAttachment(0, 0);
    data.left = new FormAttachment(0, 0);
    data.right = new FormAttachment(100, 0);
    data.bottom = new FormAttachment(100, 0);

    txtItem = new Text(container, SWT.MULTI | SWT.WRAP | SWT.V_SCROLL);
    txtItem.setLayoutData(data);
    txtItem.addModifyListener(new ModifyListener(){
        public void modifyText(ModifyEvent e)
        {
            dialogChanged();
        }
    });
    setControl(container);
}
```

代码解析：

- | FormData 的实例 data 的作用是保证无论界面如何缩放，文本框控件到容器四周的距离都是 0。
- | Swing 中一个文本框本身是不带滚动条的，如果要它出现滚动条，那么必须把文本框放入 JScrollPane 控件中。而 SWT 中则无需如此，如果要其有垂直滚动条，只要指定其具有 SWT.V_SCROLL 风格就可以了，如果要使其有水平滚动条，只要指定其具有 SWT.H_SCROLL 风格就可以了。
- | 文本框控件中 SWT.MULTI、SWT.WRAP 两种风格分别代表多行和自动换行。

(3) dialogChanged 方法

在 createControl 方法中，我们为 txtItem 控件增加了一个修改监听器，当用户在文本框中敲入内容的时候 dialogChanged 方法就会被调用。我们在这个方法中主要完成文本框中内容合法性的校验，代码如下。

【代码 2-11】 dialogChanged 方法：

```
private void dialogChanged()
{
    String strItems = txtItem.getText();
    if(strItems==null||strItems.trim().length()<=0)
    {
        updateStatus("请输入枚举项！");
        return;
    }
    String[] itemArray = strItems.split(LINESEPRATOR);
    Set<String> set = new HashSet<String>();
    for (String item : itemArray)
    {
        if(item==null||item.trim().length()<=0)
        {
            continue;
        }
        if(set.contains(item))
        {
            updateStatus("项重复："+item);
            return;
        }
        updateStatus(null);
        set.add(item);
    }
}
```

需要注意的是在验证用户输入的枚举项是否重复的时候，运用了一个小技巧，首先把文本框中的字符串按照行分隔符分割 (LINESEPRATOR 是一个常量： static final String LINESEPRATOR = System.getProperty("line.separator"))，这样就可以得到一个个的枚举项了；然后创建一个集合(Set)，遍历这些枚举项，判断每个项是否已经在集合中存在，如果存在则证明有重复，否则将此项存入这个集合中。这样做是比较高效的做法，其时间复杂度为 O(n)。

(4) 将此页面加入向导

前面所做的工作都是在定义页面，而没有将页面加入向导的代码。我们只要为 EnumGeneratorNewWizard 加入 addPages 方法，在最后的位置加入 “addPage(new EnumGen- ItemDefWizardPage());” 即可。

EnumGeneratorNewWizard 从 Wizard 类继承，并实现了 INewWizard 接口。它是插件的“管家”，负责向导页的初始化、相关环境数据的处理以及向导单击“完成”之后的业务处理。

2.5.1 初始化

初始化部分主要完成两个工作：添加向导页、将环境输入保存起来备用。在向导页中添加页面必须通过在 addPages 方法中调用 addPage 方法进行。

【代码 2-12】添加向导页面：

```
public void addPages()
```

```
{  
    genPage = new EnumGeneratoreNewWizardPage(selection);  
    addPage(genPage);  
    itemDefPage = new EnumGenItemDefWizardPage();  
    addPage(itemDefPage);  
}
```

2.5.2 相关环境数据的处理

在一个向导页启动的时候会调用 `public void init(IWorkbench workbench, IStructuredSelection selection)` 方法将工作台、当前选择对象传送给向导，由于我们只用到当前 `selection`，所以只要保存 `selection` 就可以了。

2.5.3 代码生成

单击向导的【完成】按钮以后，`performFinish` 方法就会被调用，我们可以在 `performFinish` 方法中进行代码生成和保存到磁盘上的操作。

(1) 首先读取两个向导界面中的配置参数：

```
final String packageName = genPage.getPackageName();
final String fileName = genPage.getFileName();
final Set<String> itemDefSet = itemDefPage.getEnumItems();
final IPackageFragmentRoot srcFolderPck = genPage
    .getPackageFragmentRoot();
final IPackageFragment pckFragment = srcFolderPck
    .getPackageFragment(packageName);
```

`EnumGeneratoreNewWizardPage` 从 `NewContainerWizardPage` 继承，`NewContainerWizardPage` 中的方法 `getPackageFragmentRoot` 用来取得用户选择的源文件夹，它返回的类型是 `IPackageFragmentRoot`。

`IPackageFragmentRoot` 并不仅仅代表源文件夹，它是一组 `IPackageFragment` 的根，所以它既可以是文件夹，也可以是 jar 包或者 zip 包。那么什么又是 `IPackageFragment` 呢？通俗地说，`IPackageFragment` 代表 Java 中的“包”，但是和“包”又有区别，比如源文件夹 `src` 中的包 `com.cownew.demo` 和 Jar 包中的 `com.cownew.demo` 的名称相同，因此它们是同一个“包”，但是由于这两个包在不同的 `IPackageFragmentRoot` 下，所以它们是不同的 `IPackageFragment`。通过 `IPackageFragmentRoot` 得到其下某个 `IPackageFragment` 的方法非常简单，只要调用 `IPackageFragmentRoot` 的 `getPackageFragment`，并把包名作为参数传递进去就可以了。

(2) 由于各种原因，保存代码文件的时间可能会比较长，因此我们为代码生成添加了进度对话框。JFace 对进度对话框提供了很好的支持，我们只要创建一个实现了 `IRunnableWithProgress` 接口的类的实例，把要运行的任务放到 `IRunnableWithProgress` 的 `run` 方法中即可，代码如下：

```
IRunnableWithProgress op = new IRunnableWithProgress() {
    public void run(IProgressMonitor monitor)
        throws InvocationTargetException
    {
        try
        {
            doFinish(pckFragment, packageName, fileName, monitor, itemDefSet);
        } catch (CoreException e)
        {
            throw new InvocationTargetException(e);
        } finally
        {
            monitor.done();
        }
    }
};

try
{
    getContainer().run(true, false, op);
}
```

```

{
    return false;
} catch (InvocationTargetException e)
{
    Throwable realException = e.getTargetException();
    MessageDialog.openError(getShell(), "Error", realException
        .getMessage());
    return false;
}

```

首先创建一个实现了 `IRunnableWithProgress` 接口的匿名类，把生成代码的操作放到 `run` 方法中，此处为了便于维护，将实际运行的耗时代码放到了自定义的 `doFinish` 方法中。代码生成的过程中有可能发生异常，因此进行异常处理，无论任务是否正常完成进度条都要在任务执行完成之后关闭，因此在 `finally` 中调用 `monitor.done()` 表示任务完成。

现在我们只是创建了一个 `IRunnableWithProgress` 的实例，还必须启动它，Eclipse 的向导中提供了一个很简洁的方式运行此任务，那就是 `getContainer().run(boolean fork, boolean cancelable, IRunnableWithProgress runnable)`，其中 `fork` 代表任务是否要在独立的线程中运行，`cancelable` 表示这个进度对话框是否能被取消，`runnable` 就代表要运行的任务。如果能被取消的话，当用户在任务运行的时候单击【取消】按钮，就会抛出 `InterruptedException` 异常，因此我们捕捉此异常并返回 `false`，表明此次向导操作没有完成。

下面看一看 `doFinish` 方法的代码。

【代码 2-13】 向导完成以后的操作：

```

private void doFinish(IPackageFragment pckFragment, String packageName,
String fileName, IProgressMonitor monitor, Set<String> itemDefSet) throws
CoreException
{
    monitor.beginTask("Creating " + fileName, 2);

    final ICompilationUnit cu = pckFragment.createCompilationUnit(
        fileName, EnumCodeGenUtils.getEnumSourceCode(packageName,
        fileName, itemDefSet), true, monitor);

    monitor.worked(1);
    monitor.setTaskName("Opening file for editing...");
    getShell().getDisplay().asyncExec(new Runnable() {
        public void run()
        {
            try
            {
                JavaUI.openInEditor(cu);
            } catch (PartInitException e)
            {
                Activator.logException(e);
            }
            catch (JavaModelException e)
            {
                Activator.logException(e);
            }
        }
    });
    monitor.worked(1);
}

```

首先调用 `monitor.beginTask ("Creating " + fileName, 2)` 启动任务，`beginTask` 的第一个参数代表此任务的名称，会显示在进度对话框上，此名称可以通过 `setTaskName` 方法动态设置，第二个参数表示此工作有几步。

以后随着任务的一步步完成，我们就同步地调用 `monitor.worked` 来推荐滚动条的前进。

接下来就是最重要的一步：生成代码并保存到磁盘。由于生成代码相对复杂，我们把生成代码的逻辑封装到 `EnumCodeGenUtils.getEnumSourceCode` 方法中，这里假定调用 `EnumCodeGenUtils.getEnumSourceCode` 方法就可得到枚举的 Java 源码。创建 Java 文件有两种方式：①用 IO 把 Java 代码当成普通文本文件一样创建；②把 Java 代码当成编辑单元创建。建议使用第二种方法，第一种方法适用于生成普通的文本类文件。本节使用第二种方式，我们在后面的章节中将会讲解第一种方式的应用。

在得到了其父包的 `IPackageFragment` 以后，我们创建 Java 文件会很方便，只需调用 `IPackageFragment` 的 `ICompilationUnit createCompilationUnit(String name, String contents, boolean force, IProgressMonitor monitor)` 方法即可。第 1 个参数是生成的 Java 文件名(比如 `Test.java`)，第 2 个参数为 Java 文件的内容，第 3 个参数 `force` 代表当要生成的文件已经存在的时候是否覆盖原有文件。

最后一步就是用 Eclipse 的 Java 文件编辑器打开生成的文件。由于打开文件编辑器的操作在另一个线程中，所以此处要采用 `asyncExec` 进行同步。

在 Eclipse 中用代码打开一个文件有两种方式：

- | 使用 `org.eclipse.ui.ide.IDE` 类的静态方法 `openEditor`，这个方法有 9 个重载方法，以其中一个为例：

```
public static IEditorPart openEditor(  
    IWorkbenchPage page, IFile input);
```

第 1 个参数代表要在哪个工作台页中打开，第 2 个参数是要打开的文件对象 `IFile`。这个方法可以用来打开各种类型的文件。

- | 使用 `org.eclipse.jdt.ui.JavaUI` 的 `openInEditor` 方法，其方法声明为：

```
public static IEditorPart openInEditor(IJavaElement element)
```

这种方法只能打开 Java 文件。

如果要使用第一种方法，那么首先需要将 `ICompilationUnit` 转换成 `IFile`，这个转换过程有一点烦琐。`ICompilationUnit` 接口是继承了 `IJavaElement` 接口的，因此我们使用 `JavaUI` 的 `openInEditor` 方法，将 `ICompilationUnit` 传递过去就可以了。

2.6 编写代码生成器

计算机的专家们一直在探寻一种能使得重复代码越来越少的方法，函数封装、面向对象、AOP、MDA、ORM……所有这些相关或者无关的技术都在试图将重复的代码消灭，可是一路走过来，人们突然发现，重复的代码是不可能被完全消灭的，到了更高的层次一定会有更高级的重复的代码需要我们去对付，因此代码生成也逐渐不再被妖魔化。网页编辑器、编译器、IDE 等这些非常重要的工具不就是代码生成器吗？只要是系统经过好的设计，对于剩下的一些重复性的代码与其使用学院派且严重影响性能的方法进行消除，不如使用代码生成器来完成来得更实在一些。

回到现实中来，在我们开发程序的过程中，特别是开发一些业务系统的过程中，一些重复的代码总是不可避免的，比如 ORM 中 `POJO` 代码和配置文件、资料录入界面的代码、数据库 `DDL` 语句等，这些工作如果要开发人员去手动完成话，不仅会降低开发效率，而且会带来很多 `bug`，最重要的是极容易使得开发人员产生厌倦心理从而消极怠工甚至离职，从而提高了项目的人力资源成本、增大了项目的风险。因此在大一些的开发团队中都在使用着各种或公开或自酿的代码生成工具，而且越来越多的人开始选择自酿工具，这是因为使用第三方的代码生成工具往往不能满足自己的个性化需求。

我们可以通过多种方式来写代码生成工具，比如最简单的通过 `StringBuffer` 拼字符串，或者借助 `groovy template`、`velocity` 等工具来完成，这些工具各有千秋，不过由于本书是讲解 Eclipse 的，因此我们就来看一下在 Eclipse 中有哪些代码生成方案。

1. 使用 `StringBuffer` 拼接来生成代码

在一些比较简单的代码生成中，这样的方式是比较方便的，但是当生成的代码结构变得越来越复杂的时候，代码中 `stringbuffer.append()` 与逻辑判断代码搅和在一起，程序变得非常难以维护。

2. 使用 JDT API 中的 AST

JDT 会把 Java 代码编译成 AST(Abstract Syntax Tree 抽象语法树)，这样复杂的 Java 代码就变成了相对简

单的树状结构，我们就可以通过 AST 来遍历 Java 代码，从而解析代码或者对代码进行修改，Eclipse 中的 Java 代码重构就是基于 AST 来进行的。

在 Eclipse 中 AST 被称为 CompilationUnit，对应的接口就是 ICompilationUnit，通过 Java 代码来生成 CompilationUnit 最简单的方法是使用 IPackageFragment.createCompilationUnit。指定编译单元的名称和内容，于是在包中创建了编译单元，并返回新的 ICompilationUnit。我们还可以从头创建一个 CompilationUnit，即生成一个不依赖于 Java 代码的 CompilationUnit，然后在这个 CompilationUnit 上添加类、添加方法、添加代码，然后调用 JDT 的 AST 解析器将 CompilationUnit 输出成 Java 代码。这种方式是最严谨的方式，但是当要生成的代码比较复杂的时候程序就变得臃肿无比，而且只能生成 Java 代码，不能生成 XML 配置文件等文件。

3. 使用 JET

JET 是 Eclipse 中一个非常强大的代码生成工具，使用 JET 你可以运用类似 JSP 一样的语法，这样我们就可以轻松地编写代码模板。用它可以创建 SQL 语句、XML、Java 源代码等文件的代码生成器。本书将把它作为代码生成的工具，因此我们在此处重点讲解 JET 的使用。JET 是 EMF 的一部分，要使用它必须首先安装 EMF 插件。

使用 JET 分为如下几步。

(1) 把项目转化成 JET 项目

要在项目中使用 JET，必须首先把它转化成 JET 项目，方法如下。

① 在【包资源管理器】视图上右击，在弹出的快捷菜单中选择【新建】|【其他】命令，然后在弹出的对话框中选择 Java Emitter Templates 下的 Convert Projects to JET Projects 选项，如图 2.21 所示。

② 单击【下一步】按钮，选择要转化的项目，如图 2.22 所示，然后单击【完成】按钮。向导会在项目的根目录下创建一个名字为 templates 的文件夹，而且给项目添加了一个 JET Builder，这个构建器会自动将 templates 文件夹下的模板文件进行编译，生成代码。

(2) 设置 JET

在项目上右击，在弹出的快捷菜单中选择【属性】命令，打开 JET Settings 选项卡，在这个选项卡中就可以修改模板文件夹和源文件夹了，如图 2.23 所示。注意此处必须输入源文件夹的名字，否则在生成代码的时候就有可能出现代码生成位置出错的问题。

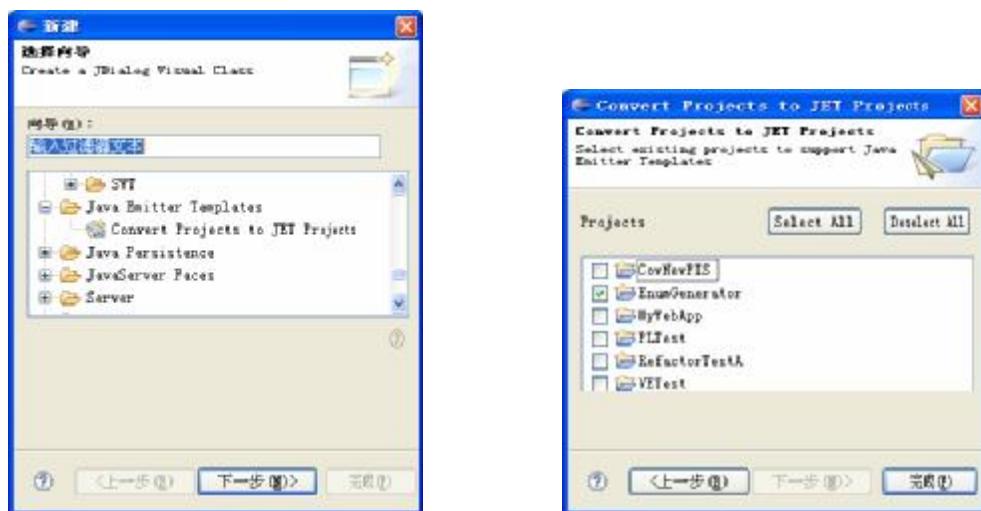


图 2.21 选择 JET 转换向导

转换的项目

图 2.22 选择被



图 2.23 设置 JET 的属性

(3) 创建模板文件

JET 的模板文件的命名规定是在要生成的代码生成器类的文件名后加 jet，比如想命名我们的代码生成器为 MyGen.java，那么只要把模板命名为 MyGen.javajet 就可以了。因此可在 templates 文件夹下创建一个文件 EnumCodeGenerator.javajet，创建完毕之后，系统会弹出一个错误对话框，如图 2.24 所示。



图 2.24 构建出错对话框

不要惊慌，这并不是说明我们的创建过程有错，而是创建完模板文件以后，JET 构建器就去尝试构建 `EnumCodeGenerator.javajet`，由于这个文件是空的，所以当然就构建失败报错了。

在 `EnumCodeGenerator.javajet` 中输入如下代码：

```
<%@ jet package="com.cownew.enumgenerator.wizards" class="EnumCodeGenerator" %>
Hello,<%=argument%>!
```

保存以后，JET 就立即会生成 `EnumCodeGenerator.java` 文件，内容如下：

```
public class EnumCodeGenerator
{
    protected static String nl;
    public static synchronized EnumCodeGenerator create(
        String lineSeparator)
    {
        nl = lineSeparator;
        EnumCodeGenerator result = new EnumCodeGenerator();
        nl = null;
        return result;
    }

    protected final String NL = nl == null ?
        (System.getProperties().getProperty("line.separator")) : nl;
    protected final String TEXT_1 = " Hello, ";
    protected final String TEXT_2 = "!";
    protected final String TEXT_3 = NL;

    public String generate(Object argument)
    {
        final StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append(TEXT_1);
        stringBuffer.append(argument);
        stringBuffer.append(TEXT_2);
        stringBuffer.append(TEXT_3);
        return stringBuffer.toString();
    }
}
```

可以看到 JET 生成的代码采用的也是 `StringBuffer` 拼装的形式，注意此处生成的代码是无法手工修改的，因为每次修改以后保存的时候 JET 会自动把代码替换成未修改之前的代码。

(4) 测试模板代码

在 `EnumCodeGenUtils` 中创建 `main` 方法，然后输入如下代码：

```
EnumCodeGenerator gen = new EnumCodeGenerator();
System.out.println(gen.generate("Eclipse"));
```

运行之后控制台中就打印出了：Hello, Eclipse!

我们来对上边的模板代码和测试代码做一下简要的分析：

① `<%@ jet package="com.cownew.enumgenerator.wizards" class="EnumCodeGenerator" %>`

这是模板的头部分，以 “@ jet” 开头，这部分主要声明此模板的有关信息，比如生成代码的包路径、类名、导入的类等， `package` 属性定义的就是生成代码的包路径，而 `class` 属性定义的是生成的类名。

② `Hello,<%=argument%>!`

这部分就是模板的正文了，和 JSP 语法一样，显示一个变量的方法是`<%=变量名%>`。注意这里的变量 `argument` 是有特殊含义的，它表示传递给模板的参数。

③ 代码生成器生成代码的方法是 `generate`，因为我们经常需要传递一些参数给代码生成器，所以 `generate` 方法有一个类型为 `Object` 的参数，此参数在模板中可以用 `argument` 取得。

对 JET 有了一个感性的认识之后，我们就来通过实战来操练一下。上一节中 `EnumCodeGenUtils.getEnumSourceCode` 方法的实现为空，这一节我们就来完成这项关键性的工作。

经过分析，我们发现需要传递给模板代码如下 3 个参数才可以正确地输出代码：枚举类的包名、枚举类的类名、枚举类的项。因为模板代码的 generate 方法只接受类型为 Object 的一个参数，所以我们需要把这 3 个参数封装到一个 JavaBean 中，如下定义 JavaBean。

【代码 2-14】 模板参数类：

```
public class EnumGenArgInfo
{
    private Set<String> items;
    private String className;
    private String packageName;

    public String getPackageName()
    {
        return packageName;
    }
    public void setPackageName(String packageName)
    {
        this.packageName = packageName;
    }
    public String getClassName()
    {
        return className;
    }
    public void setClassName(String className)
    {
        this.className = className;
    }
    public Set<String> getItems()
    {
        return items;
    }
    public void setItems(Set<String> items)
    {
        this.items = items;
    }
}
```

接下来我们来写模板文件。

【代码 2-15】 模板文件：

```
<%@ jet package="com.cownew.enumgenerator.wizards"
   class="EnumCodeGenerator"
   imports="Java.util.*"
%>
<%
    EnumGenArgInfo argInfo = (EnumGenArgInfo)argument;
    Set<String> enumItems = argInfo.getItems();
    String className = argInfo.getClassName();
    String packageName = argInfo.getPackageName();
%>

    package <%=packageName%>

    public class <%=className%>
    {
        private String type;
```

```

public <%=className%> <%=item%> = new <%=className%>(" <%=item%> ");
<%}%>

private <%=className%>(String type)
{
    super();
    this.type = type;
}

public int hashCode()
{
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + ((type == null) ? 0 : type.hashCode());
    return result;
}

public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    final <%=className%> other = (<%=className%>) obj;
    if (type == null)
    {
        if (other.type != null)
            return false;
    } else if (!type.equals(other.type))
        return false;
    return true;
}
}

```

这个模板文件是非常简单的，有了前面的基础，读懂这个模板文件就非常简单了，这里只讲两点：

- | 文件头的 imports 属性是用来定义生成的代码的 import 列表的，这个模板中用到了集合类 Set，所以要用 imports="Java.util.*" 将其导入，否则生成的代码会编译错误。如果要导入多个类，只要把它们用空格隔开即可，比如：

```
imports= imports="Java.util.* Java.sql.Date"
```

不能使用其他分隔符。

- | 由于传递进来的参数是一个 JavaBean，因此需要把 argument 进行一次转型操作：

```
EnumGenArgInfo argInfo = (EnumGenArgInfo)argument;
```

编写下面的代码测试一下这个代码模板：

```

public static void main(String[] args)
{
    EnumCodeGenerator gen = new EnumCodeGenerator();
    EnumGenArgInfo argInfo = new EnumGenArgInfo();
    argInfo.setClassName("MyEnum");
    gen.generate(argInfo);
}

```

```

        items.add( "VIP" );
        items.add( "MM" );
        argInfo.setItems(items);
        argInfo.setPackageName( "com.cownew" );
        System.out.println(gen.generate(argInfo));
    }
}

```

运行之后发现输出的代码完全正确。

这样我们就可以来完成 `EnumCodeGenUtils` 类的 `getEnumSourceCode` 方法。

【代码 2-16】 完成后的 `getEnumSourceCode` 方法:

```

public static String getEnumSourceCode(String packageName, String fileName,
                                       Set<String> itemDefSet)
{
    Pattern pattern = Pattern.compile("(.).Java");
    Matcher mat = pattern.matcher(fileName);
    mat.find();
    String className = mat.group(1);
    EnumCodeGenerator gen = new EnumCodeGenerator();
    EnumGenArgInfo argInfo = new EnumGenArgInfo();
    argInfo.setClassName(className);
    argInfo.setItems(itemDefSet);
    argInfo.setPackageName(packageName);
    return gen.generate(argInfo);
}

```

这里用到了正则表达式来从 Java 文件名中提取类名，使用的是 JDK 中的正则表达式实现，对于正则表达式，我们可以去查阅相关资料，正则表达式是一个非常好用的工具，掌握以后能轻松解决很多字符串解析相关的问题，并为学习编译原理打下基础。

2.7 功能演示、打包安装

所有的代码编写工作都已经完成，让我们来测试一下。在项目 `EnumGenerator` 上右击，在弹出的快捷菜单中选择【运行方式】|【Eclipse 应用程序】命令，片刻以后测试工作台就会启动完毕。

在要新建枚举的包上右击，在弹出的快捷菜单中选择选择【新建】|【其他】命令，在向导对话框中选择 `EnumGenerator` 下的【枚举创建向导】选项，如图 2.25 所示。

单击【下一步】按钮，创建一个颜色枚举，如图 2.26 所示，进行设置。

单击【下一步】按钮，在定义枚举项界面中输入要定义的颜色项目，如图 2.27 所示。

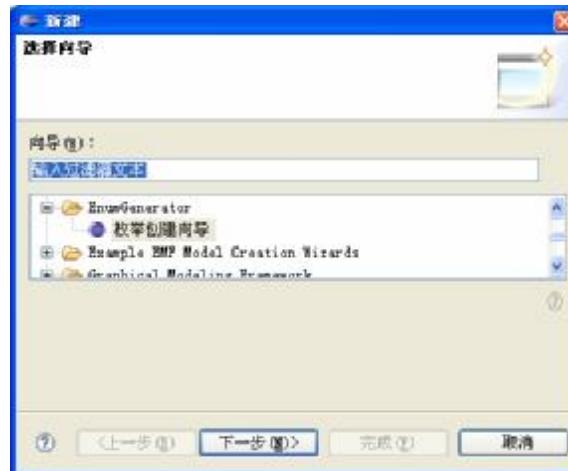


图 2.25 枚举创建向导



图 2.26 设定枚举类的属性



图 2.27 设定枚举项

单击【完成】按钮，就可以看到自动生成了如下的代码。

```
package com.cownew.enumtest;
public class ColorEnum
{
    private String type;
    public ColorEnum RED = new ColorEnum("RED");
    public ColorEnum YELLOW = new ColorEnum("YELLOW");
    public ColorEnum BLUE = new ColorEnum("BLUE");
    public ColorEnum BLACK = new ColorEnum("BLACK");
    public ColorEnum GREEN = new ColorEnum("GREEN");

    private ColorEnum(String type)
    {
        super();
        this.type = type;
    }

    public int hashCode()
    {
        final int PRIME = 31;
        int result = 1;
        result = PRIME * result + ((type == null) ? 0 : type.hashCode());
        return result;
    }
}
```

```

}

public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    final ColorEnum other = (ColorEnum) obj;
    if (type == null)
    {
        if (other.type != null)
            return false;
    } else if (!type.equals(other.type))
        return false;
    return true;
}
}

```

这里运行的例子是运行在插件开发环境中的，如果要是给用户使用的话必须将其导出，成为可以部署的安装包。

导出生成可以部署的安装包是非常简单的，只需要在【包资源管理器】中右击，在弹出的快捷菜单中选择【导出】命令，选中【导出】向导页中的【插件开发】目录下的“可部署的插件和段”，单击【下一步】按钮，如图 2.28 所示。

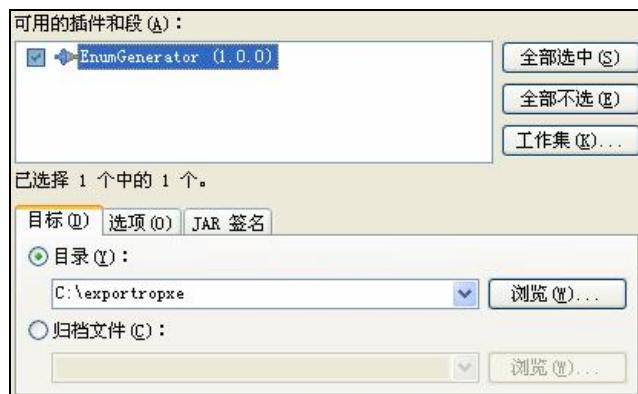


图 2.28 导出插件

选中 EnumGenerator 项，然后在【目录】文本框中填入要导出到哪个目录下(也可以选择下边的【归档文件】而将插件导出为 jar 包)，单击【完成】按钮即可。

本章以一个简单而实用的例子介绍了 Eclipse 的插件开发。这里所写的程序还有很多并不是很完美的地方，甚至有一些明显的 bug，为了不花费太多的精力在这些细节上，我们没有完全展开叙述。相信读者会在学习和实战中发现更多的问题并通过摸索解决这些问题，如果能把学到的东西应用到实际开发中去的话，本章的目的也就达到了。

第 3 章 插件开发导航

在上一章中，介绍了一个完整的插件的开发，相信读者已经掌握了插件开发的基本知识。不过我们只是介绍了每一步怎么去做，却没有介绍为什么要这么做以及相关的知识点。本章将会把插件开发相关的常用知

识学习一遍。插件开发涉及到的知识是非常多的，这些知识都可以通过阅读 Eclipse 的帮助文档以及阅读开源项目的代码获得，所以这里没有必要去介绍每一个细节，我们只会把知识点介绍一下，读者无须强行记忆，只要知道这一章是做什么的，并且遇到问题时能去查询相关的资料就可以了。

3.1 程序界面的基础——SWT/JFace

AWT、Swing 是 Java 标准库中的图形化界面框架，但是由于其在性能、稳定性、美观性等方面有很多问题，导致用 Java 开发出来的成熟 GUI 项目非常少，而且即使是成熟的 Java GUI 系统，比如 JBuilder、NetBeans 等的界面也是遭人诟病的。在 IBM 开始开发 Eclipse 的时候，开发人员都强烈反对使用 Swing 开发 Eclipse，因此 IBM 决定采用 Smalltalk GUI 的实现方式来开发一个新的 Java 界面框架，这个框架也就是 SWT。

与 AWT/Swing 不同，SWT 底层采用的是 JNI native 调用本地操作系统的 API，只有本地操作系统实现不了的部件才去自己绘制，因此 SWT 的效率是非常高的。

经过不断成熟和发展，SWT 现在已经成为与 Eclipse 无关的开发包了，也就是用 SWT 做出来的程序只要没有调用 Eclipse 平台的东西，那么它就可以脱离 Eclipse 运行。由于本章中讲到的 SWT 界面都是运行在 Eclipse 中的，因此我们不再介绍单独运行 SWT 程序的方法。

SWT 是对操作系统 GUI API 的封装，因此没有做更多应用层次的封装，比如要显示一个对话框，就要自己去画【确定】、【取消】按钮，要弹出消息对话框就要自己去写数行代码。为了简化 SWT 的开发，IBM 开发出了 JFace，JFace 不是与 SWT 格格不入的，JFace 就是调用 SWT 实现了更多实际应用开发中要用到的公共类。SWT 和 JFace 的关系就像 Windows 开发中 Windows SDK 和 MFC 的关系一样，我们在开发的时候应当尽量去使用 JFace 的东西，只有当 JFace 的东西不满足我们要求的时候才去直接求助 SWT。

3.1.1 SWT 的类库结构

SWT 的所有类都在 org.eclipse.swt 包下。最重要的类就是 Widget，它是所有界面对象的基类，类图如图 3.1 所示。

Widget 的直接子类有 Caret(插入光标)、Menu(菜单)、ScrollBar(滚动条)、Tray(系统托盘图标)等。Widget 的子类 Item 下的类是一些无法独立于其他部件的部件，比如 MenuItem(菜单项)、TableItem(表格项)、TrayItem(系统托盘图标项)、TreeItem(树项)等。Widget 的子类 Control 是一个比较庞大的基类，大部分 SWT 部件都在此类下，其直接子类有 Button(按钮)、Label(标记)、ProgressBar(进度条)等。Control 的子类 Scrollable 是所有可以带滚动条的对象的基类，比如 Text(文本框)、List(列表框)等。Scrollable 的子类 Composite 是 SWT 中一个重要的类，它是所有可以容纳其他部件的类的基类，其子类有 Browser(浏览器)、Combo(下拉列表框)、Group(组合框)、Table(表格)、Tree(树)等。

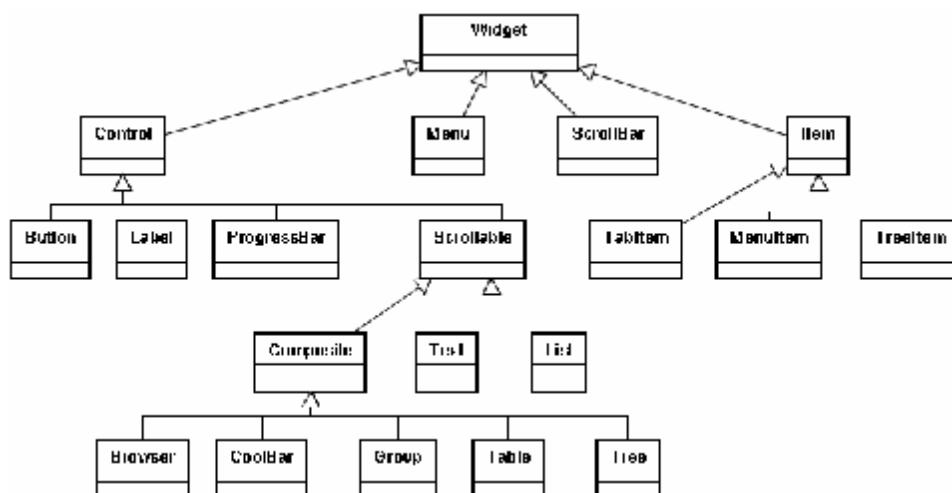


图 3.1 SWT 的类结构图

上面从类层次的角度研究了 SWT 的类结构，下面再来看一下 SWT 的包结构：

括部件风格、消息常量等；SWTException 和 SWTError 则是 SWT 中异常的基类。

- | org.eclipse.swt.widgets 包下定义了常用、核心 SWT 窗口小部件(widget)的公有 API 类定义。如 Display、Shell、Button、Menu 等。一般编写 GUI 程序用到的 Widget 大部分都在这个包下。
- | org.eclipse.swt.events 包中提供了对 SWT 事件监视器(Event Listener)的支持，如 Button 的 SelectionListener、Mouse 的 MouseListener、MouseMoveListener 和 MouseTrackListener 等，还有与这些 Listener 对应的 Adapter 实现类和 Event 类。
- | org.eclipse.swt.layout 包中定义了 SWT 的布局管理器，其中有 FillLayout、GridLayout 和 RowLayout 三种。
- | org.eclipse.swt.graphics 包中包含了 SWT 中 graphic 类，如 Color、Font 和 Image 等，这个包下的类的资源管理方式和其他部件略有不同，3.1.2 节中将会介绍。
- | org.eclipse.swt.printer 提供了对打印的支持。
- | org.eclipse.swt.custom 包中包含了一些可自定义的窗口小部件，它们是学习开发自定义 SWT 部件的很好的例子。
- | org.eclipse.swt.dnd 提供了对拖放操作的支持。

3.1.2 SWT 中的资源管理

AWT/Swing 的资源管理使用的是 Java 提供的垃圾回收机制，但是由于 GUI 是非常消耗资源的，要求对象不被使用的时候立即被回收，而垃圾回收机制是无时间保证的。这对系统资源的处理会是致命的，比如程序在一个循环语句中去加载数万张图片，对其进行加盖印章处理后再进行保存，常规的处理方式是每次调入一张，修改保存，然后就立即释放该图片资源，而后再循环调入下一张图片，这对操作系统而言，任何时刻程序占用的仅仅是一张图片的资源。但如果资源管理完全交给垃圾回收机制去处理，也许会是在循环语句结束后，JVM 才会去释放图片资源，其结果可能是你的程序还没有运行结束，系统就已经内存溢出了。而 SWT 则创新性地抛弃了 Java 提供的垃圾回收机制，让资源由开发者进行生命周期管理，即显式地释放已经分配的任何操作系统资源(调用 `dispose` 方法释放资源)。其法则就是：①如果您创建对象，则您必须销毁它；②父部件被销毁，子部件也同时被销毁。具体实施起来有如下规则：

- | 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销毁。
- | 当调用一个包含子部件的部件(即 `Composite` 的子类)的 `dispose` 方法时，将递归地调用其所有子部件的 `dispose` 方法。因此无需手动去释放这些子部件的资源。
- | 如果部件不是您创建的，而是调用其他类的某个方法得到的，则不要将其除去，这是因为它不是您创建的。

很多开发人员在看到“父部件被销毁，子部件也同时被销毁”这一条的时候就认为自己又不用去管理资源的释放工作了，因为只要销毁根部件，那么所有的子部件就都会被销毁了。这是十分危险的误解，因为 SWT 中还有一部分资源不是继承自 `Widget` 的，也就是不可能有父部件，这样就需要程序员手动去释放资源。例如 `org.eclipse.swt.graphics` 下的类，这些类都继承了 `Resource` 类，`Resource` 中也定义了 `dispose` 方法。这些类有 `Color`(颜色)、`Cursor`(鼠标指针)、`Font`(字体)、`GC`(图形设备设置)、`Image`(图片)等。比如您为按钮设定了一种字体，那么必须在销毁这个按钮的时候手动去释放字体对象。

当然并不是所有的 SWT 对象都需要程序员去释放的，比如 `org.eclipse.swt.graphics` 包下的 `Point`(点)、`Rectangle`(矩形)、`RGB` 等类是没有 `dispose` 方法的，因此只有把它们交给 JVM 的垃圾回收机制去管理了。

3.1.3 在非用户线程中访问用户线程的 GUI 资源

在非用户线程中对用户线程的 GUI 资源进行访问的时候，如果不进行同步的话就会造成不可预料的问题。AWT/Swing 中并没有强制在非用户线程中访问用户线程的 GUI 资源的时候要进行同步，而 SWT 则进行了同步控制，这样就可以预防这些不可预料的问题。在 SWT 中，通常存在一个被称为“用户线程”的唯一线程，只有在这个线程中才能调用对组件或某些图形 API 的访问操作。如果在非用户线程中程序直接调用这些访问操作，那么 `SWTException` 异常会被抛出。

下面看一个例子：

```
Runnable r = new Runnable() {
    public void run() {
        {
            for (int i = 0; i < 100; i++)
            {
                try
                {
                    wait(1000);
                } catch (InterruptedException e)
                {
                }
                text.setText(new Integer(i).toString());
            }
        }
    }
};
```

我们启动一个线程，在这个线程中，每隔一秒为界面文本控件赋值一次，运行后就会抛出 SWT 异常。解决这个问题的方法也是非常简单的，那就是通过 `Display` 类的 `syncExec(Runnable)` 和 `asyncExec`

(Runnable)这两个方法去实现:

```
Runnable r = new Runnable() {
    public void run()
    {
        for (int i = 0; i < 100; i++)
        {
            try
            {
                wait(1000);
            } catch (InterruptedException e) { }
            final int j = i;
            display.asyncExec(new Runnable() {
                public void run()
                {
                    text.setText(new Integer(j).toString());
                }
            });
        }
    }
};
```

方法 syncExec()和 asyncExec()的区别在于前者要在指定的线程执行结束后才返回，而后者无论指定的线程是否执行都会立即返回到当前线程。

3.1.4 访问对话框中的值

在程序中经常会弹出一些对话框，要求用户输入一些值，然后根据用户输入的值来进行后续操作。比如在程序中弹出一个对话框要求用户输入姓名和国家，然后根据输入的值显示问候语。

定义 SettingDialog 类：

```
public class SettingDialog extends Dialog
{
    private Text txtName;
    private Text txtCountry;
    ...
    public String getName()
    {
        return txtName.getText();
    }

    public String getCountry()
    {
        return txtCountry.getText();
    }

}
```

主界面：

```
SettingDialog dlg = ... ;
dlg.open();
if(dlg.open() == Window.OK)
{
    MessageDialog.openInformation(shell, "", dlg.getName() + " is from " +
        dlg.getCountry());
}
```

当运行的时候就会抛出如下异常：

```
org.eclipse.swt.SWTException: Widget is disposed
```

这是为什么呢？

让我们来看一下 `org.eclipse.jface.window.Window` 类，它是 `SettingDialog` 的间接父类，在 `Window` 类的 `close` 方法中将界面控件全部销毁掉了，当我们关闭一个界面的时候就调用了 `close` 方法，这样当窗口已经关闭的时候，我们再去调用 `dlg.getName()` 的话，`getName` 方法就会去访问 `txtName` 控件，可是 `txtName` 已经被销毁掉了，不能被访问了，所以就抛出了 `Widget is disposed` 这个异常消息。

那么我们应该怎么修改呢？既然不能在窗口关闭以后访问界面控件对象，那么只有在关闭之前来把要访问的控件值提前保存起来了。做如下修改：

```
public class SettingDialog extends Dialog
{
    private Text txtName;
    private Text txtCountry;
    private String name;
    private String country;
    ...

    protected void okPressed()
    {
        name = txtName.getText();
        country = txtCountry.getText();
        super.okPressed();
    }
}
```

```

public String getName()
{
    return name;
}
public String getCountry()
{
    return country;
}
}

```

当单击【确定】按钮的时候, `okPressed` 方法会被调用, 我们在调用父类的 `okPressed` 之前将控件的值保存起来就可以了, 并且改写了 `get` 方法, 让它返回我们保存的值。

3.1.5 如何知道部件支持哪些 style

SWT 中的部件的构造函数都有一个 `int style` 参数, 这个参数表示要创建的部件的风格。

这个参数的类型是整数而非枚举, 那么如何确定它支持哪些值呢? 答案就是查看 JavaDoc。以 `Text` 部件为例, 查看 `Text` 的源码, 定位到它的构造函数处, 如图 3.2 所示。

```

/*
 * @see SWT#SINGLE
 * @see SWT#MULTI
 * @see SWT#READ_ONLY
 * @see SWT#WRAP
 * @see Widget#checkSubclass
 * @see Widget#getStyle
 */
public Text (Composite parent, int style) {
    super (parent, checkStyle (style));
}

```

图 3.2 构造函数的 JavaDoc

JavaDoc 中明确地指出支持如下的风格:

`SWT.SINGLE`、`SWT.MULTI`、`SWT.READ_ONLY`、`SWT.WRAP`。

SWT 中定义的常量都是掩码形式的, 比如:

```

public static final int MULTI = 1 << 1;      //即二进制的 10
public static final int WRAP = 1 << 6;        //即二进制的 1000000

```

可以用“或”操作符来进行风格的组合, 比如指定文本框为多行并且自动换行, 只要如下调用即可:

```
Text txt = new Text (parent, SWT.MULTI | SWT.WRAP);
```

3.2 SWT 疑难点

SWT 的 API 数量是非常多的, 不过对于熟悉 Swing 或者其他语言的界面开发者来说, 只要借助帮助文档一般都可以很快掌握其使用。所以就不打算把所有部件的使用方式再重复一遍了, 这里只介绍一些需要重点注意的控件方法。

3.2.1 Button 部件

在 GUI 术语中,  和  这两个部件分别被叫做复选框和单选按钮, 也就是它们也是按钮的一种, 但是无论是 Delphi 中的 VCL、.NET Framework 还是 Swing, 都把它们看作是与 `Button` 不同的部件。当程序员开始学习 SWT 的时候, 突然发现 SWT 中没有 `Checkbox`、`RadioButton` 控件了, 难道 SWT 不支持吗?

当然是不是了, SWT 中把 、 和  看成了按钮部件的不同样式, 不再用不同的类区分它们。当新建一个 `Button` 实例的时候, 只要指定风格为 `SWT.CHECK` 或者 `SWT.RADIO` 就可以了。

3.2.2 Text 部件

与 Swing 中不同, SWT 中的单行文本框和多行文本框都是用 Text 部件表示, 只是通过 SWT.SINGLE、SWT.MULTI 两种不同的风格来区分。Text 本身支持滚动条(SWT.H_SCROLL 和 SWT.V_SCROLL 风格), 无需像 Swing 中那样要把文本框包在 ScrollPane 中才可以。

3.2.3 Tray

Tray 是 SWT 提供的一个系统托盘类, 通过这个类可以在系统托盘中增加图标。这看起来很酷, 不过要尽力避免使用它, 因为目前此部件还不能做到完全地跨平台。

3.2.4 Table

表格控件是一个非常常用的控件, 可以用来实现大数据量的展示和编辑, SWT 中的 Table 控件就是提供这样功能的一个控件, 与 Swing 的 JTable 比起来 Table 的可用性更好。下面就来看一下对 Table 的主要操作方式。

(1) 给 Table 增加列:

```
TableColumn colName = new TableColumn(table, SWT.LEFT);
colName.setText("名称");
colName.setWidth(100);
```

(2) 给 Table 增加行:

```
TableItem item = new TableItem(table, SWT.NONE);
```

(3) 为单元格增加编辑器:

```
TableEditor editor = new TableEditor(table);
Text text = new Text(table, SWT.NONE);
editor.grabHorizontal = true;
editor.setEditor(text, item, 5);
editor = new TableEditor(table);
```

Table 的单元格默认是没有任何编辑器的, 当然单元格也就是只读的了, 如果我们要编辑单元格的话就必须为单元格设定编辑器, 编辑器可以是文本框、复选框、单选按钮等。

下面这段代码就是为 item 这一行的第 5 列增加一个文本编辑器, 同理也可以为其增加复选框编辑器:

```
editor = new TableEditor(table);
Button button = new Button(table, SWT.CHECK);
button.pack();
editor.minimumWidth = button.getSize().x;
editor.horizontalAlignment = SWT.CENTER;
editor.setEditor(button, item, 5);
```

3.2.5 在 SWT 中显示 AWT/Swing 对象

SWT 在设计之初是想兼容 AWT/Swing 的, 但是由于 SUN 的极度不配合, 导致 SWT 最终没有能兼容 AWT。这就造成 AWT/Swing 中原有的一些很好用的代码无法移植到 SWT 中的问题。比如做图形化报表通常使用 JFreeChart, 但是 JFreeChart 只能显示在 AWT/Swing 中, 没有提供 SWT 的支持, 是否代表我们在 SWT 中就不能使用它了呢? 当然不是了。我们可以用 SWT_AWT 桥接器来解决这个问题。

调用 SWT_AWT 的 getFrame 方法就可以把一个 Composite 面板转换成 AWT 中的 Frame, 这样就可以在这个 Frame 中进行任何 AWT/Swing 相关的操作了。

比如使用 JFreeChart:

```
Composite drawarea = new Composite(parent, SWT.EMBEDDED);
drawarea.setLayout(new FillLayout());
Frame canvasFrame = SWT_AWT.new_Frame(drawarea);
canvas = new Java.awt.Canvas() {
    public void paint(Graphics g) {
        super.paint(g);
        if (chart != null)
            chart.draw((Graphics2D)g, getBounds());
    }
};
canvasFrame.add(canvas);
```

如果在 Frame 中使用了 AWT/Swing 部件，我们也完全可以在 SWT 代码中访问 AWT/Swing 部件，由于这些 AWT/Swing 部件不是在一个 UI 线程中的，所以在访问的时候要进行同步。

```
Display.getDefault().asyncExec(new Runnable{
    public void run() {
        awtButton.setText("I am AWT Button");
    }
});
```

还有一些 Java 的图形化应用也可以通过这种方式来支持 SWT，比如一些成熟的 Java GIS 应用。

3.3 异步作业调度

编程的时候经常会遇到一些长时间的操作，比如读取大量文件并进行解析、从远端服务器读取文件、进行复杂的数据库操作等，如果处理不好的话，会造成程序好像死掉了一样。令人震惊的是，很多程序员对此并不在乎，因为他们知道程序为什么而“死掉了”，并向用户解释说程序在做什么，不用担心，只要等就可以了。如果站在用户的角度思考一下就知道这种想法有多么可怕。

这里讲作者经历过的事情：曾经开发过一个从超大 XML 文件(大于 10M)中导入数据并插入到数据库中的功能，由于在导入每一条数据的时候都要把和这条数据有关的数据从数据库中取出来，然后进行一定的处理后再插入到数据库中，所以耗时是非常长的，一般都要耗时半个小时以上。在做第一个版本的时候没有考虑进度条，当把程序发给用户的时候，用户用了一会儿就打电话过来：“那个程序死掉了，帮我看看吧！”，通过向他解释这是正常的，他这才将信将疑地放下电话，没过了 5 分钟，又打电话过来“怎么还是死的，你们怎么做的程序，我要投诉你！”。后来终于导入成功了，但是从用户的反馈来看，他们是十分的不满意。后来在给这个程序开发 bug 修复补丁的时候顺手给程序加上了进度条的功能，随时报告当前的进度，几乎没有增加工作量。谁知发给客户以后，客户赞扬说：“这个版本改进比较大呀，好多了，不错！现在我都是单击完【导入】按钮以后就去做别的事情了，时不时地回来看看导入进度！”——作者这才深刻的意识到“进度条”这个在技术人员看起来微不足道的小功能在改善用户体验方面有多么重要的作用。

后来在去客户现场做支持的时候看到的一幕又感到猛然一惊。所做的那个数据导入功能是 ERP 系统中的一部分，这个 ERP 系统是一次可以打开多个内部窗口的(类似于 Windows 中的 MDI)，用户可以在一个窗口中录单，切换到另一个窗口中制作报表，或者切换到另一个窗口发邮件。看到用户在打开那个数据导入窗口，单击【导入】按钮后就切换到另外一个窗口进行录单操作了。天呀，如果没有提供那个进度条的功能，那么用户单击【导入】按钮以后整个 ERP 系统就“死掉了”，用户就无法进行任何操作，也就无法做任何工作，难道这半个多小时要他去上网聊 QQ、翻纸牌吗？

在这一点上 Eclipse 做的无疑是非常好的。当我们新建一个项目的时候，如果项目的初始化时间比较长，Eclipse 就会弹出一个带滚动条的窗口，提示用户正在初始化；对于一些耗时非常长的操作，比如从 CVS 检出代码，Eclipse 会弹出一个带有【在后台运行】按钮的进度对话框，如图 3.3 所示，用户单击【在后台运行】按钮以后，这个对话框就会关闭，这样用户就可以在 Eclipse 中进行其他的操作了，避免了长时间等待所造成的时间浪费。

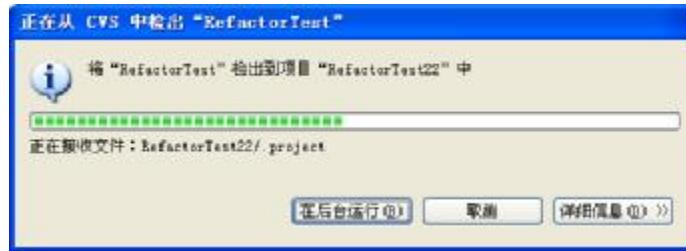


图 3.3 进度条

我们最常接触的就是 `IProgressMonitor` 了，在很多方法中都要求传递此接口的实例，比如编辑器的 `doSave` 方法就是如下声明的：

```
public void doSave(IProgressMonitor monitor)
```

通过这个接口就可以操控进度条来显示我们当前的保存进度了。不过 `IProgressMonitor` 并不是进度条对话框，它要“依靠”一个进度显示器来把进度显示出来，比如最常见的进度对话框 `ProgressMonitorDialog`。

部分任务在运行的时候可以由用户选择取消，当用户取消任务的时候，`IProgressMonitor` 的 `isCanceled` 方法会返回 `true`，因此我们在任务进行的时候要实时地去调用 `isCanceled` 方法，当发现任务被取消的时候要尽快结束任务。

我们可以使用 Java 的标准接口 `Runnable` 来实现多线程任务运行，不过在 Eclipse 中又有了新的选择，那就是 `IRunnableWithProgress`，其声明如下：

```
public interface IRunnableWithProgress {
    public void run(IProgressMonitor monitor)
        throws InvocationTargetException, InterruptedException;
}
```

这个类的使用和 `Runnable` 非常相似，只要把任务放到 `run` 方法中就可以了，最重要的是可以调用 `monitor` 来对当前进度显示进行控制。下面就是一个完整的进度条演示例子。

```
ProgressMonitorDialog dialog = new ProgressMonitorDialog(shell);
dialog.run(true, true, new IRunnableWithProgress() {
    public void run(IProgressMonitor monitor)
        throws InvocationTargetException, InterruptedException
    {
        final int ticks = 10000;
        monitor.beginTask("开始操作", ticks);
        try
        {
            for (int i = 0; i < ticks; i++)
            {
                if (monitor.isCanceled())
                    throw new InterruptedException();
                monitor.worked(1);
            }
        } finally
        {
            monitor.done();
        }
    }
});
```

调用 `beginTask` 方法来完成任务，`ticks` 参数表示此任务有多少工作量，调用 `worked` 方法报告自上次报告以来当前完成的任务数量，在循环中不断通过 `isCanceled` 方法判断当前任务是否被用户取消。需要注意，要在 `finally` 中调用 `done` 方法完成任务，否则会出现进度对话框无法正常关闭的情况。

除了 `ProgressMonitorDialog` 外，在 Eclipse 中还可以通过其他方式显示进度，比如 `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` 在向导状态行中显示长时间运行的操作。

除了可以自己构造进度对话框来显示进度之外，我们还可以调用平台的进度服务，而且 Eclipse 也推荐使用平台的进度服务，这样可以使所有插件都将具有一致的进度表示。平台的进度服务定义为接口 `IProgressService`，我们可以通过 `PlatformUI.getWorkbench().getProgressService` 方法来调用系统的进度服务，例如：

```
IProgressService progressService = PlatformUI.getWorkbench()
    .getProgressService();
progressService.busyCursorWhile(new IRunnableWithProgress() {
    public void run(IProgressMonitor monitor) {
        {
            //执行耗时的操作
        }
    }
});
```

在调用 Eclipse 的方法或者第三方插件的一些方法的时候，有的方法要求传递一个实现了 `IProgressMonitor` 的实例进去，如果我们无法传递或者无需传递的时候，最好不要传递 `null` 值进去，而是要传递 `NullProgressMonitor` 的一个实例进去，此类位于 `org.eclipse.core.runtime` 包下，它实现了 `IProgressMonitor` 接口，但是所有方法都是给的空实现，传递此类就避免了被调用方法没有进行空指针判断而造成的麻烦。

3.4 对话框

3.4.1 信息提示框

信息提示框对应的类为 `org.eclipse.jface.dialogs.MessageDialog`，它定义了如下主要方法。

(1) 确认对话框(如图 3.4 所示):

```
public static boolean openConfirm(Shell parent, String title, String message)
```

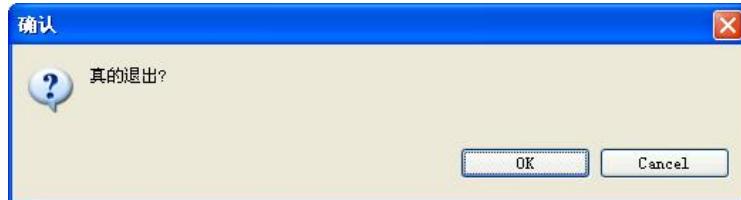


图 3.4 确认对话框

(2) 错误信息框:

```
public static void openError(Shell parent, String title, String message)
```

(3) 普通消息框:

```
public static void openInformation(Shell parent, String title, String message)
```

(4) 询问对话框(如图 3.5 所示):

```
public static boolean openQuestion(Shell parent, String title, String message)
```

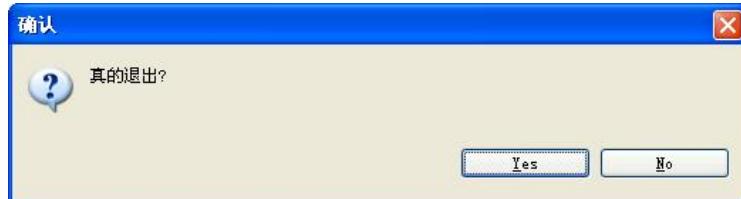


图 3.5 询问对话框

注意询问对话框与确认对话框方法的区别。

(5) 警告对话框：

```
public static void openWarning(Shell parent, String title, String message)
```

3.4.2 值输入对话框

在和用户交互的时候，对于一些复杂的信息，可能需要通过自定义的对话框进行采集，而对于像简单的字符串之类的信息，则可以通过弹出值输入对话框的方式进行采集。

值输入对话框定义在 `org.eclipse.jface.dialogsInputDialog` 中，与消息对话框不同，这个类是必须实例化才能使用的，其构造函数为：

```
public InputDialog(Shell parentShell, String dialogTitle,  
String dialogMessage, String initialValue, IInputValidator validator)
```

参数 `dialogTitle` 为标题，`dialogMessage` 为显示的消息，`initialValue` 为对话框中的初始值，`validator` 为值校验器。当 `validator` 为 `null` 的时候，不对对话框中的值做校验，而非 `null` 的时候需要做校验。

`IInputValidator` 接口定义如下：

```
public interface IInputValidator {  
    public String isValid(String newText);  
}
```

当 `isValid` 返回非空的时候，值校验不通过，并且把 `isValid` 返回的值作为错误信息显示。

使用值输入对话框的例子如下：

```
InputDialog inputDlg = new InputDialog(shell, "输入", "请输入您的年龄", "20",  
new IInputValidator(){  
    public String isValid(String newText)  
    {  
        int i;  
        try  
        {  
            i = Integer.parseInt(newText);  
        } catch (NumberFormatException e)  
        {  
            return "年龄必须为整数!";  
        }  
  
        if(i<0)  
        {  
            return "兄弟来自反物质世界?年龄不可能为负吧!";  
        }  
  
        if(i>150)  
        {  
            return "您也太高寿了吧!";  
        }  
  
        return null;  
    }  
});  
  
if(inputDlg.open()==Window.OK)  
{
```

```
System.out.println(inputDlg.getValue());  
}
```

运行以后当在对话框中输入“-20”的时候就会提示错误，如图 3.6 所示。

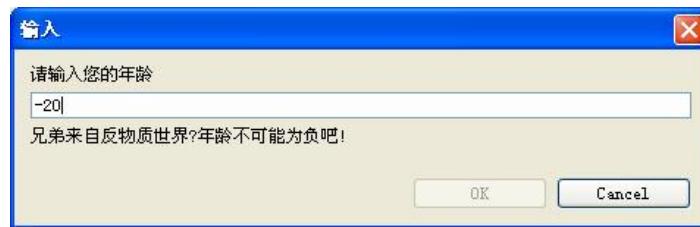


图 3.6 输入对话框

3.4.3 错误对话框

错误对话框定义在 `org.eclipse.jface.dialogs.ErrorDialog` 中，它有两个重载的 `openError` 静态方法，与其他对话框不同的是有一个 `IStatus status` 参数，这个参数用来设置错误信息，一般我们使用它的一个实现类 `org.eclipse.core.runtime.Status`，`Status` 中定义了两个静态常量 `OK_STATUS`、`CANCEL_STATUS`，我们可以使用它们，如果它们不能满足要求，就要调用 `Status` 的构造函数进行实例化，其构造函数如下：

```
public Status(int severity, String pluginId, int code, String message, Throwable exception)  
| severity 表示错误的程度，可取值为 OK、ERROR、INFO、WARNING、CANCEL。  
| pluginId 为调用插件的插件 id，一般定义在对应插件的 Activator 中。  
| code 为错误代码。  
| message 为错误消息。  
| exception 为要抛出的异常。
```

显示效果如图 3.7 所示。



图 3.7 错误对话框

3.4.4 颜色选择对话框

颜色选择对话框定义在 `org.eclipse.swt.widgets.ColorDialog` 中，其调用方法与普通对话框没有什么不同。例如：

```
ColorDialog colorDlg = new ColorDialog(shell);  
RGB rgb = colorDlg.open();  
if(rgb!=null)  
{  
    Color color = null;  
    try  
    {  
        color = new Color(shell.getDisplay().getRGB());  
    }
```

```
//使用color...
}
finally
{
    if(color!=null)
        color.dispose();
}
}
```

这里得到返回值的方式非常值得研究，对话框并没有直接返回 Color，而是返回一个 RGB 对象的实例，由调用者来根据 RGB 构造 Color，这正好符合了 SWT 中资源管理的一个原则：“谁创建谁销毁”。如果 ColorDialog 返回值是 Color 类型，那么必须由 ColorDialog 负责销毁，可是 ColorDialog 不知道什么时候去销毁，所以 ColorDialog 就返回了一个由 JVM 去负责销毁的对象 RGB，此对象包含了需要的信息，由调用者去构造，类似的用法在下面的字体对话框中也可以看到。

ColorDialog 还有一个 setRGB 方法可以用来给颜色对话框设置初始值。

3.4.5 字体对话框

字体对话框定义在 org.eclipse.swt.widgets.FontDialog 中，调用方法如下：

```
FontDialog fontDlg = new FontDialog(shell);
FontData fontData = fontDlg.open();
if(fontData!=null)
{
    Font font = null;
    try
    {
        font = new Font(shell.getDisplay(),fontData);
        //使用 font...
    }
    finally
    {
        if(font!=null)
            font.dispose();
    }
}
```

和颜色对话框类似，字体对话框返回的字体信息是保存在由 JVM 负责资源回收的 FontData 对象中的，由调用者来根据 FontData 对象构造字体对象。FontDialog 有一个 setFontList 方法可以用来设置初始值。

3.4.6 目录选择对话框

目录选择对话框定义在 org.eclipse.swt.widgets.DirectoryDialog 中，调用方法如下：

```
DirectoryDialog dirDlg = new DirectoryDialog(shell);
String dir = dirDlg.open();
if(dir!=null)
{
    System.out.println(dir);
}
```

DirectoryDialog 中定义了如下几个方法。

- | **setText:** 为对话框设置窗口标题。
- | **setMessage:** 为对话框设置提示信息。
- | **setFilterPath:** 为对话框设置初始路径。

下面的代码执行以后的效果如图 3.8 所示。

```
DirectoryDialog dirDlg = new DirectoryDialog(shell);
dirDlg.setText("这里是 Text");
dirDlg.setMessage("这里是 Message");
dirDlg.setFilterPath("c:/Downloads");
String dir = dirDlg.open();
```



图 3.8 目录选择对话框

3.4.7 文件选择对话框

与 .Net、Delphi、VB 等框架中的文件对话框不同，SWT 中的保存对话框和打开对话框都定义在 org.eclipse.swt.widgets.FileDialog 类中，只要在构造函数中指定不同的风格即可。

打开对话框：

```
FileDialog fileDlg = new FileDialog(shell, SWT.OPEN);
```

保存对话框：

```
FileDialog fileDlg = new FileDialog(shell, SWT.SAVE);
```

FileDialog 中定义了如下几个方法。

- | setFileName: 设定初始文件名。
- | setFilterExtensions: 设定文件名过滤器。
- | setFilterPath: 设定初始路径。
- | setText: 设定对话框标题。
- | getFileNames: 以数组形式返回选中的多个文件名。
- | getFilterPath: 返回选中的路径。

调用例子：

```
FileDialog fileDlg = new FileDialog(shell, SWT.OPEN | SWT.MULTI);
fileDlg.setFilterExtensions(new String[]{"*.mp3", "*.wmv", "*.rm"});
fileDlg.setFilterPath("F:/资料/My Music");
fileDlg.setText("请选择要打开的音乐文件(支持多选)");
String filePath = fileDlg.open();
if(filePath!=null)
{
    System.out.println("路径:"+fileDlg.getFilterPath());
    String[] files = fileDlg.getFileNames();
    for(int i=0,n=files.length;i<n;i++)
    {
        System.out.println(files[i]);
    }
}
```

3.4.8 自定义对话框及配置保存与加载

程序中经常会需要一些自定义对话框，我们可以从 SWT 的 Dialog 类(位于包 org.eclipse.swt.widgets 中)派生，也可以从 JFace 的 Dialog(位于包 org.eclipse.jface.dialogs 中)中派生。建议我们如果没有特别的需要最好从 JFace 的 Dialog 继承，因为它提供了更多 SWT 对话框中没有提供的功能。

此功能，不过 Eclipse 提供了更好的功能，那就是使用对话框配置服务。对话框值的保存与加载的核心类就是 org.eclipse.jface.dialogs 包下的 IDialogSettings，它可以用来保存和检索与键名相关联的基本数据类型和字符串值。

每个插件都有自己的 Activator，这些 Activator 的基类就是 AbstractUIPlugin，我们可以通过 AbstractUIPlugin 的 getDialogSettings 方法来取得此插件对应的对话框配置服务，此服务并不是只有在对话框中才能调用，我们可以在插件代码中的任何位置访问对话框设置。取得服务的方式非常简单：

```
IDialogSettings settings = Activator.getDefault().getDialogSettings();
```

配置中的值是使用 get 和 put 方法来存储和检索的，可以存储和加载布尔、长型、双精度、浮点、整型、数组和字符串值。以下代码段说明可以如何使用对话框设置来初始化对话框中的控件值：

```
protected Control createDialogArea(Composite parent) {  
    IDialogSettings settings = Activator.getDefault().getDialogSettings();  
    checkbox.setSelection(settings.getBoolean("isOpenWhenLoad"));  
}
```

单击【确定】按钮时，就可以存储设置值：

```
protected void okPressed() {  
    IDialogSettings settings = Activator.getDefault().getDialogSettings();  
    settings.put("isOpenWhenLoad", checkbox.getSelection());  
    super.okPressed();  
}
```

3.5 首 选 项

Eclipse 中的首选项是整个 Eclipse 的配置中心，插件的主要配置都在这个地方完成。首选项也是可以定制的，也就是说我们可以将我们自己的首选项页面加入这个首选项对话框中。

Eclipse 中提供了一个首选项开发的向导，我们可以通过这个向导生成的代码来理解首选项的开发。

这里重点讲解一下首选项的配置保存。org.eclipse.core.runtime.preferences 包提供了用于访问首选项的类。与上面讲解的对话框配置保存一样，插件首选项也是以键值对的形式保存的，其中键描述首选项的名称，而值必须是几种不同类型中的一种(boolean、double、float、int、long 或 string)。

通过 AbstractUIPlugin 的 getPreferenceStore 方法可以取得首选项配置服务。

读取：

```
IPreferenceStore store = getPreferenceStore();
checkBox1.setSelection(store.getBoolean("isLoad"));
```

保存：

```
IPreferenceStore store = getPreferenceStore();
store.setValue("isLoad", checkBox1.getSelection());
```

首选项页都直接或者间接地从 PreferencePage 类继承，在初始化的时候需要首先调用 setPreferenceStore 方法为此页设定一个首选项配置服务，当【应用】、【取消】、【默认值】或【确定】按钮被单击的时候，performApply、performCancel、performDefaults、performOk 方法将会分别被调用，我们就可以在这些方法中保存配置，而在 createContents 中构造控件的时候去加载这些参数。

如果我们要自己编写首选项配置界面的话，不仅要处理页面布局，还要自己处理属性的保存、加载，幸好 Eclipse 为我们提供了一个更好用的配置界面基类 FieldEditorPreferencePage。FieldEditorPreferencePage 将每一个配置项看成一个字段编辑器，整个页面就是由不同类型的字段编辑器组成的。FieldEditorPreferencePage 提供了常见字段编辑器：

- | BooleanFieldEditor——布尔字段编辑器。
- | IntegerFieldEditor——整数编辑器，可调用 setValidRange 来限制整数的范围。
- | StringFieldEditor——文本编辑器，可以调用 setEmptyStringAllowed 来限制是否能为空。
- | RadioGroupFieldEditor——单选按钮组编辑器。
- | ColorFieldEditor——颜色编辑器。
- | FontFieldEditor——字体编辑器。
- | DirectoryFieldEditor——文件夹编辑器。
- | FileFieldEditor——文件编辑器。
- | ScaleFieldEditor——步进范围整数编辑器。

各个插件还可以从 FieldEditor 继承来编写符合自己个性化要求的字段编辑器，图 3.9 是一个字段编辑器页面的典型应用。

Eclipse 的插件开发中“首选项向导”生成的就是基于 FieldEditorPreferencePage 的代码，可以仔细研究一下。

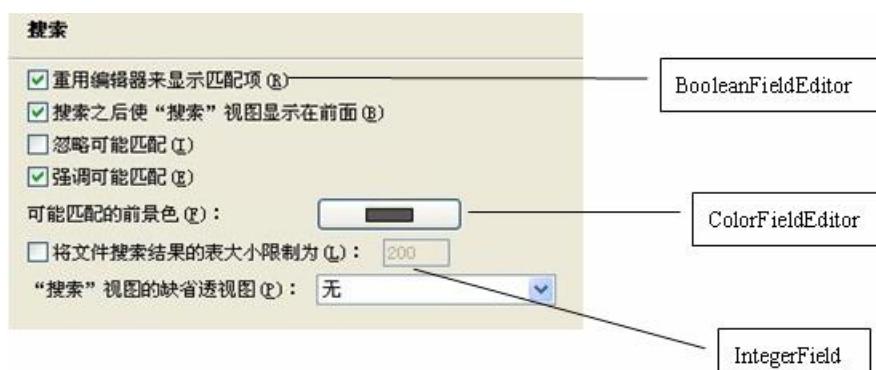


图 3.9 字段编辑器示例

首先看 SamplePreferencePage.java:

```
public class SamplePreferencePage extends FieldEditorPreferencePage implements IWorkbenchPreferencePage
{
    public SamplePreferencePage()
    {
        super(GRID);
        setPreferenceStore(Activator.getDefault().getPreferenceStore());
    }

    public void createFieldEditors()
    {
        addField(new DirectoryFieldEditor(PreferenceConstants.P_PATH,
            "&Directory preference:", getFieldEditorParent()));
        addField(new BooleanFieldEditor(PreferenceConstants.P_BOOLEAN,
            "&An example of a boolean preference", getFieldEditorParent()));
        addField(new RadioGroupFieldEditor(PreferenceConstants.P_CHOICE,
            "An example of a multiple-choice preference", 1,
            new String[][] { { "&Choice 1", "choice1" },
            { "C&hoice 2", "choice2" } }, getFieldEditorParent()));
        addField(new StringFieldEditor(PreferenceConstants.P_STRING,
            "A &text preference:", getFieldEditorParent()));
    }

    public void init(IWorkbench workbench)
    {
    }
}
```

我们需要做的只是在构造函数中给它传递一个首选项配置存取接口并在 createFieldEditors 方法中调用 addField 方法添加各种各样的编辑器，其他的事情根本无需我们处理。图 3.10 是运行效果图。

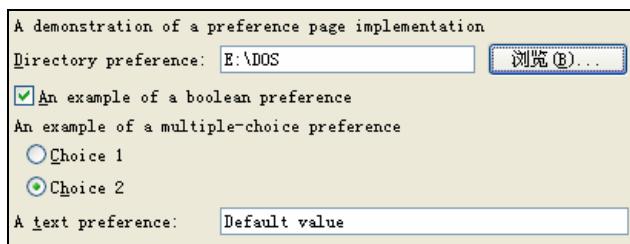


图 3.10 示例首选项页

插件代码还生成了一个初始化类，用来设置页面的默认值:

```
public class PreferenceInitializer extends AbstractPreferenceInitializer
{
    public void initializeDefaultPreferences()
    {
        IPreferenceStore store = Activator.getDefault().getPreferenceStore();
        store.setDefault(PreferenceConstants.P_BOOLEAN, true);
        store.setDefault(PreferenceConstants.P_CHOICE, "choice2");
        store.setDefault(PreferenceConstants.P_STRING, "Default value");
    }
}
```

如果您的首选项页面非常复杂的话，可能只使用 `addField` 添加简单的编辑器解决不了问题，您可以直接使用 SWT 代码来对首选项页进行更精细化、更复杂的调整。

3.6 Eclipse 资源 API 和文件系统

说到 Eclipse 中与资源相关的最重要的概念就是：工作空间、项目、文件夹和文件。工作空间的资源组织成树结构，项目位于顶部，而文件夹和文件在下面。特殊资源、工作空间根目录资源充当资源树的根目录。工作空间可以有任意数目的项目，每个项目都可以存储在磁盘上的不同位置。工作空间负责管理用户资源，组织一个或多个顶级项目。每个项目对应于工作空间目录中的子目录。每个项目都可以包含文件和文件夹。图 3.11 是项目中不同资源的示意图。

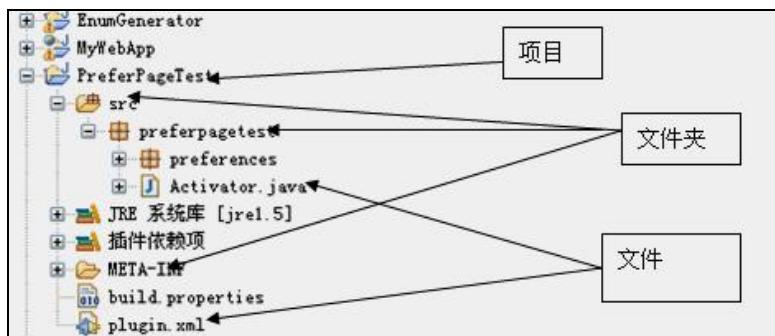


图 3.11 项目中的不同资源

相信上图我们大部分都能看懂，需要注意的是 Java 工程中的包、源文件夹、普通文件夹在 Eclipse 资源这一个层次看起来都属于文件夹，它们的不同其实是由 JDT 来标识和区分的。

工作空间下可能有一个 `.metadata` 目录，它是一个特殊的文件夹，其中存储的是工作空间相关的配置文件，我们不能使用一般文件系统 API 来编辑或处理这些文件。与此相似的就是每个项目目录下的 `.project`。

工作空间、项目、文件夹、文件对应的类型接口分别为 `IWorkspace`、`IProject`、`IFolder`、`IFile`。资源相关的接口都继承了 `IResource` 接口，由于工作空间并不只是资源的管理者，因此 `IWorkspace` 并没有继承 `IResource` 接口，为了将工作空间作为资源的管理者这一功能体现出来，抽象出了 `IWorkspaceRoot` 的接口(即工作空间根目录)，通过 `IWorkspace` 的 `getRoot` 就可以得到对应的工作空间根目录。由于工作空间根目录、项目、文件夹都是可以容纳其他资源的，因此为它们抽取了一个基类接口 `IContainer`。图 3.12 为 Eclipse 中资源相关类的继承结构图。

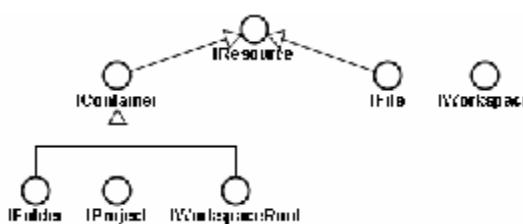


图 3.12 资源的继承结构图

3.6.1 资源相关接口的常见方法

为了方便地找到有用的方法，这里我们简单介绍一下资源相关接口的常见方法。

(1) IResource

- | `delete`: 删除此资源。
- | `getFileExtension`: 返回文件的扩展名。

- | `getFullPath`: 返回此资源相对于工作空间根目录的相对路径, 返回值类型是 `IPath`。
- | `getLocation`: 返回此资源在文件系统中的绝对路径, 返回值类型是 `IPath`。
- | `exists`: 判断此资源是否存在。处理资源与使用 `Java.io.File` 处理文件非常相似。`IResource` 只是一个句柄。当调用像 `getProject`、`getFolder` 这样的方法时, 会将句柄返回给资源, 即使指定的资源并不存在。因此在必要的时候要使用 `exists` 方法来确定资源是否存在。
- | `getParent`: 得到父资源容器, 返回值类型为 `IContainer`。
- | `getProject`: 返回此资源所属的项目, 返回值类型为 `IProject`。
- | `getProjectRelativePath`: 返回此资源在项目中的相对路径, 返回值类型是 `IPath`。
- | `getWorkspace`: 返回此资源所属的工作空间, 返回值类型为 `IWorkspace`。
- | `isSynchronized`: 判断资源是否与文件系统同步。

(2) `IContainer`

- | `exists(IPath path)`: 判断指定的路径 `path` 是否在本容器内存在。
- | `findMember`: 返回指定路径的资源, 返回值类型为 `IResource`。
- | `getCharset`: 返回此容器内资源的默认编码。
- | `getFile`: 返回指定路径的文件, 返回值类型为 `IFile`。
- | `getFolder`: 返回指定路径的文件夹, 返回值类型为 `IFolder`。
- | `members`: 返回容器下的所有直接资源, 返回值类型为 `IResource[]`。

(3) `IFolder`

- | `create`: 创建此文件夹所代表的资源。
- | `getFile`: 返回此文件夹下指定路径下的文件, 返回值类型为 `IFile`。
- | `getFolder`: 返回此文件夹下指定路径下的文件夹, 返回值类型为 `IFolder`。

(4) `IFile`

- | `appendContents`: 向文件中附加数据流, 方法中有一个参数是 `InputStream` 类型, 要添加的数据流就是在此流中。
- | `create`: 用数据流创建文件, 方法中有一个参数是 `InputStream` 类型, 要创建的文件的数据流就是在此流中。
- | `getCharset`: 返回文件的编码。
- | `setCharset`: 设定文件的编码。
- | `getContents`: 返回文件的流, 返回值类型为 `InputStream`。
- | `setContents`: 用数据流设定文件的内容, 方法中有一个参数是 `InputStream` 类型, 要设定的文件的数据流就是在此流中。

(5) `IWorkspaceRoot`

- | `getProject`: 返回指定名字的项目, 返回值类型为 `IProject`。
- | `getProjects`: 返回工作空间中所有的项目, 返回值类型为 `IProject[]`。

3.6.2 方法中 force 参数的意义

上面列出的许多资源相关接口的方法都包括一个 force 布尔参数，它指定是否将更新与本地文件系统中的相应文件不同步的资源。很多刚刚接触 Eclipse 的开发人员错误的认为工作空间中的资源是和文件系统中同步的，而事实并非如此。当添加、删除或编辑资源时，资源和文件系统就不再同步；在工作空间外对工作空间中的资源进行添加、删除或编辑后也会造成不同步。API 中的其他方法允许使用程序控制文件系统刷新，例如：

```
IResource.refreshLocal(int depth, IProgressMonitor monitor);
```

用户还可以在工作台的资源导航器视图中显式地强制执行刷新。

3.6.3 资源相关接口的方法使用示例

下面来看一看上面介绍的资源相关接口的方法的使用示例。

(1) 取得工作空间

用如下代码取得工作空间：

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
```

其中的 ResourcesPlugin 定义在 org.eclipse.core.resources.ResourcesPlugin 中。

(2) 访问工作空间中的资源

要访问工作空间中的资源必须首先获得 IWorkspaceRoot，它表示工作空间的根目录，也就是资源层次结构的最顶部：

```
IWorkspaceRoot myWorkspaceRoot = ResourcesPlugin.getWorkspace().getRoot();
```

一旦得到工作空间根目录，就可以访问工作空间中的项目，由于项目必须被打开才能被方法访问，因此我们要首先打开项目：

```
IProject myProject = myWorkspaceRoot.getProject("MyProject");
if (myProject.exists() && !myProject.isOpen())
    myProject.open(null);
```

一旦有打开的项目，我们就可以访问它的文件夹和文件。在下面的代码中，我们根据位于工作空间外部的文件的内容来创建文件资源：

```
IFolder imagesFolder = myProject.getFolder("images");
if (imagesFolder.exists()) {
    IFile newLogo = imagesFolder.getFile("newLogo.png");
    FileInputStream fileStream = new FileInputStream(
        "c:/MyOtherData/newLogo.png");
    newLogo.create(fileStream, false, null);
}
```

第一行获得图像文件夹的句柄。必须检查该文件夹是否存在，然后才能对它执行任何操作。同样地，当我们获得文件 newLogo 时，在我们在最后一行创建文件之前，句柄不代表实际的文件。

3.6.4 在 Eclipse 中没有当前项目

经常听到有人问这种问题：

“我怎么在当前项目下创建一个文件呢？”

“如何取得当前项目的源文件夹呢？”

这通常是受其他 IDE 使用的惯性思维影响的开发人员提出来的问题。在 VS.NET Studio、JBuilder 等 IDE 中也有与 Eclipse 中类似的工作空间、项目的概念的，允许在一个 IDE 中同时打开多个项目，在这多个项目中，有一个主项目，其他项目都是为此项目提供服务的，启动的时候启动的也是这个主项目。Eclipse 则不同，
在 Eclipse 工作空间中的每个项目都有自己的本地文件系统，而且对于此项目而言，项目目录的路径是唯一的，不存在于其他项目的目录中。

可以启动。

这样一来，我们也许会发出疑问了：如果没有当前项目的概念，为什么我新建 Java 文件的时候是建在项目 A 下而不是项目 B 下呢？这就是使用了当前选择项目的概念，可以参考前面“枚举代码生成器”中根据 selection 得到当前包的代码。

3.7 Java 项目模型

在上节中我们讲解的是 Eclipse 的资源模型，讲解到的项目也是普通的 Eclipse 项目，Eclipse 的项目有很多，包括 Java 项目、C++项目、C#项目等，每种项目都有自己的特点。我们最常接触到的项目就是 Java 项目，因此我们重点来讲解一下 Java 项目模型。

Java 模型是用来对与创建、编辑和构建 Java 程序相关联的对象进行建模的一些类。Java 模型类是在 org.eclipse.jdt.core 中定义的。这些类实现资源的特定于 Java 的行为，并进一步将 Java 资源分解成模型元素。

3.7.1 类结构

Java 模型的继承结构图如图 3.13 所示。

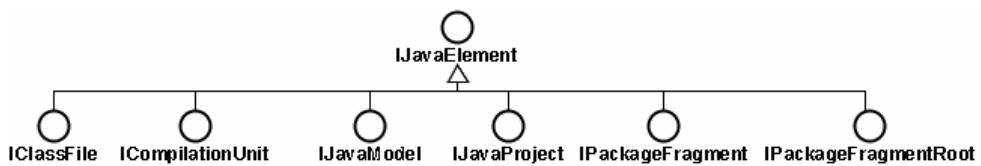


图 3.13 Java 模型的继承结构图

IJavaElement 的子类接口还有 IMethod、IType 等，在这里没有全部列出。Java 模型中的类结构比较简单，层级也非常少。

下面介绍一下各个接口的主要方法。

(1) IJavaElement

- | exists: 判断元素是否存在。处理 Java 元素与处理资源对象相似。当使用 Java 元素时，实际上是在使用某些底层的模型对象的句柄。必须使用 exists() 来确定元素是否真正存在于工作空间中。
- | getElementName: 返回元素的名称。
- | getJavaModel: 返回其对应的 JavaModel，返回值类型是 IJavaModel。
- | getJavaProject: 返回元素对应的 Java 工程，返回值类型是 IJavaProject。
- | getParent: 返回父元素，返回值类型是 IJavaElement。
- | getResource: 返回元素对应的资源，返回值类型是 IResource。

(2) IClassFile

此接口代表编译后的 class 二进制文件。

- | isClass: 判断是否是 Java 类。
- | isInterface: 判断是否是接口。

(3) ICompilationUnit

此接口代表 Java 源文件。

- | getAllTypes: 返回此文件中定义的所有类型，返回值类型是 IType[]。一个 Java 文件中可以定义多个类型。
- | getPrimary: 返回主类型，返回值类型是 ICompilationUnit。

(4) IJavaModel

此接口表示根 Java 元素，对应于工作空间。是所有具有 Java 性质的项目的父类。它对于 Java 项目的作用和 IWorkspaceRoot 对于 IProject 的作用相似。

- | contains: 判断是否存在指定的资源。
- | getJavaProject: 返回指定名字的 Java 项目，返回值类型是 IJavaProject。

| I getJavaProjects: 返回所有的 Java 项目, 返回值类型是 IJavaProject[]。

| I getWorkspace: 返回所在的工作空间。

(5) IJavaProject

此接口表示 Java 项目。

| I IJavaElement createElement(IPath path): 返回项目的 path 路径下的 Java 元素。

| I IPackageFragment findPackageFragment(IPath path): 返回项目的 path 路径下的 IPackageFragment。

| I IPackageFragmentRoot findPackageFragmentRoot(IPath path): 返回项目的 path 路径下的 IPackageFragmentRoot。

| I findType: 根据一个全名取得此元素的类型, 此类有数个重载方法, 返回值类型为 IType。

| I getAllPackageFragmentRoots : 返回所有的 IPackageFragmentRoot , 返回值类型是 IPackageFragmentRoot[]。

| I getOutputLocation: 返回输出路径, 返回值类型是 IPath。

| I getRequiredProjectNames: 返回依赖项目, 返回值类型是字符串数组。

| I setOutputLocation: 设定输出路径。

(6) IPackageFragment

此接口表示整个包或者包的一部分。

| I createCompilationUnit: 创建一个 ICompilationUnit, 返回值类型是 ICompilationUnit。

| I getClassFile: 返回指定名称对应的 IClassFile, 返回值类型是 IClassFile。

| I getClassFiles: 返回所有的 IClassFile, 返回值类型是 IClassFile[]。

| I getCompilationUnit: 返回指定名称对应的 ICompilationUnit, 返回值类型是 ICompilationUnit。

| I getCompilationUnits: 返回所有 ICompilationUnit, 返回值类型是 ICompilationUnit[]。

| I getKind: 判断此包是源码包还是普通包, 返回值是 int 型, 如等于 IPackageFragmentRoot.K_SOURCE 则是源文件包, 如等于 IPackageFragmentRoot.K_BINARY 则为普通包。

| I hasSubpackages: 是否有子包。

(7) IPackageFragmentRoot

此接口表示一组包段, 并将各段映射至底层资源, 它可以是文件夹、JAR 或 ZIP 文件。

| I createPackageFragment: 创建一个 IPackageFragment, 返回值类型是 IPackageFragment。

| I getKind : 此包段是源码包段还是二进制包段, 返回值类型是 int , 如果等于 IPackageFragmentRoot.K_SOURCE 则是源文件包段, 如果等于 IPackageFragmentRoot.K_BINARY 则为二进制包段。

| I getPackageFragment: 根据包名返回对应的 IPackageFragment。

3.7.2 常用工具类

(1) JavaCore(定义在 org.eclipse.jdt.core 包下)

JavaCore 从 Plugin 继承, 它是 JDT 插件的生命周期管理器。不过对于第三方插件开发人员来说, 它的重要性更多地体现在它提供的一些工具类方法中。

| I IJavaElement create(IFile file): 从文件创建对应的 Java 元素。

| I IJavaElement create(IFolder folder): 从文件夹创建对应的 Java 元素。

| I IJavaProject create(IProject project): 得到 IProject 对应的 IJavaProject。

| I IJavaElement create(IResource resource): 从资源创建对应的 Java 元素。

| I IJavaModel create(IWorkspaceRoot root): 从工作空间根目录得到对应的 IJavaModel。

| I IClassFile createClassFileFrom(IFile file): 从文件创建对应的 IClassFile。

| I ICompilationUnit createCompilationUnitFrom(IFile file): 从文件创建对应的 ICompilationUnit。

(2) JavaUI(定义在 org.eclipse.jdt.ui 包下)

JavaUI 中定义了常用的 Java 插件界面相关的方法。

| I createPackageDialog: 创建一个包选择对话框, 返回值是 SelectionDialog。

| I createTypeDialog: 创建一个类型选择对话框, 返回值是 SelectionDialog。

- | IEditorPart openInEditor(IJavaElement element): 用编辑器打开指定的 Java 元素并返回编辑器实例。
- | revealInEditor(IEditorPart part, IJavaElement element): 在编辑器中定位元素 element。

3.7.3 常用技巧

插件开发中经常会碰到一些常用的技巧，掌握这些技巧可以极大地提高插件的开发效率，并且可以减小插件的体积。下面列出一些常见的技巧。

(1) 由一个普通项目得到 Java 项目

Java 项目是一种特殊的项目，需要注意的是 IJavaProject 并不是从 IProject 继承的。不能将一个 IProject 对象强制转换成一个 IJavaProject 对象，也不能把一个 IJavaProject 实例赋值给 IProject 变量。

- | 由 IProject 项目得到 Java 项目的方式：

```
IJavaProject javaProject = JavaCore.create(IProject);
```

- | 由 IJavaProject 得到 IProject 的方式：

```
调用 IJavaProject 的 IProject getProject();
```

(2) 得到工作空间中的所有 Java 项目

我们可以首先得到工作空间中的所有项目，然后逐个进行转换。不过这不免麻烦了一些，下面介绍更好的方式。IJavaModel 是所有 Java 项目的根，通过它就可以得到所有的 Java 项目：

```
IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
IJavaModel jModel = JavaCore.create(root);
IJavaProject jProject[] = jModel.getJavaProjects();
```

(3) 打开 Java 编辑器并显示 Java 编译单元的特定成员

代码如下：

```
void showMethod(IMember member) {
    ICompilationUnit cu = member.getCompilationUnit();
    IEditorPart JavaEditor = JavaUI.openInEditor(cu);
    JavaUI.revealInEditor(JavaEditor, member);
}
```

(4) 在工程下创建一个 com.cownew 包，并创建一个 Hello.java 文件

代码如下：

```
IPackageFragmentRoot pkroot = JavaProject
    .getPackageFragmentRoot(JavaProject.getResource());
IPackageFragment pkg = pkroot.createPackageFragment("com.cownew", true,
    new NullProgressMonitor());
pkg.createCompilationUnit("Hello.Java", "package com.cownew;", true,
    new NullProgressMonitor());
```

(5) 打开【打开类型】对话框

以下代码段使用 JavaUI 类来打开【打开类型】对话框：

```
SelectionDialog dialog = JavaUI.createTypeDialog(parent,
    new ProgressMonitorDialog(parent),
    SearchEngine.createWorkspaceScope(),
    IJavaElementSearchConstants.CONSIDER_ALL_TYPES, false);
dialog.setTitle("打开类型");
dialog.setMessage("请选择要打开的类型");
if (dialog.open() == IDialogConstants.CANCEL_ID)
    return null;
```

```

if (types == null || types.length == 0)
    return null;
System.out.println(types[0]);

```

用类似方法还可以创建【打开包】和【打开主要类型】对话框。

(6) 打包指定的文件

我们写一些工具的时候也许需要把文件打成 jar 包，然后进行发布到应用服务器等操作，调用 JDT 提供的类可简化这个操作(用到的打 Jar 包的类都在 org.eclipse.ui.jarpackager 下):

```

void exportToJar(IType mainType, IFile[] filestoExport) {
    JarPackageData description= new JarPackageData();
    IPath location= new Path("C:/cownew.jar");
    description.setJarLocation(location);
    description.setSaveManifest(true);
    description.setManifestMainClass(mainType);
    description.setElements(filestoExport);
    IJarExportRunnable runnable=
        description.createJarExportRunnable(parentShell);
    new ProgressMonitorDialog(parentShell).run(true,true, runnable);
}

```

参数 mainType 表示 Jar 包的 main 类， filestoExport 为要打包的文件。

(7) 自动设置 Java 项目的构建路径

有一些插件会将需要的 jar 包自动设置到构建路径上，比如使用 WTP 的新建向导新建 web 项目的时候就会把 web 开发需要的 jar 包自动放入项目的构建路径，使用 PDE 的“将项目转换为插件项目”功能后项目的构建路径中就增加了插件依赖项的库。那么它们是怎么实现的呢？

Java 项目的构建路径有如下几种：源文件夹、二进制库、依赖项目、类路径变量和类路径容器。

- | 源文件夹：一个包含源代码编译单元的文件夹，这些源代码编译单元组织在它们的相应包目录结构下面。源文件夹用来更好地在大型项目中组织源文件，并且只能在包含项目内引用源文件夹。如图 3.14 所示。
- | 二进制库：类文件文件夹(包含在工作空间内部)或类文件归档文件(包含在工作空间内部或外部)。
- | 依赖项目：另一个 Java 项目。依赖项目总是将它的源文件夹提供给从属项目使用。(可选)它还可以提供它的任何标记为已导出的类路径条目。这意味着除了将它的源添加至其从属项之外，项目还将导出这样标记的所有类路径条目。这将允许先决条件项目更好地隐藏它们自己的结构更改。例如，给定的项目可以选择从使用源文件夹切换到导出库。完成此操作并不要求其从属项目更改它们的类路径。如图 3.15 和图 3.16 所示。
- | 类路径变量：可以相对于类路径变量来动态解析项目或库的位置，类路径变量是作为条目路径的第一个段指定的。条目路径的其余部分被追加至已解析的变量路径。
- | 类路径容器：对一组结构化项目或库的间接引用。类路径容器用来引用一组描述复杂库结构的类路径条目。



图 3.14 源文件夹

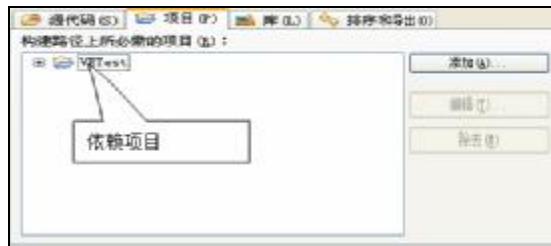


图 3.15 构建依赖项目

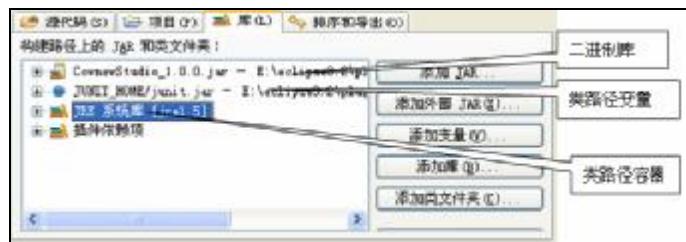


图 3.16 Jar 和类文件夹依赖

每种不同的构建路径都有不同的作用：源文件夹是把源码进行构建的途径，二进制库是导入少量 jar 包的方式，依赖项目是供多项目分模块开发使用的，使用类路径变量可以避免二进制包的路径依赖，而类路径容器则为大量二进制库的引入提供了方便。

JDT 为这些不同的构建路径提供了一个统一的接口：IClassPathEntry，只要调用 IJavaProject 的 setRawClasspath 方法就可以为项目设定构建路径。

```
IProject project = ... // 获取一些项目资源
IJavaProject JavaProject = JavaCore.create(project);
IClasspathEntry[] newClasspath = ...;
JavaProject.setRawClasspath(newClasspath, someProgressMonitor);
```

可以看到 setRawClasspath 方法需要一个 IClasspathEntry 数组，数组中的元素就是要设置的每一个构建路径。前面提到的 JavaCore 类提供了一系列的静态方法来帮助我们生成不同的 IClasspathEntry，而无须关注生成的细节。下面来看不同构建路径的添加方式。

① 源文件夹。使用 JavaCore.newSourceEntry 方法。下面的代码的作用是构造项目 MyProject 的源文件夹 src 的类路径条目：

```
IClassPathEntry srcEntry =
JavaCore.newSourceEntry(new Path("/MyProject/src"));
```

② 二进制库 IClasspathEntry。使用 JavaCore.newLibraryEntry 方法。下面的代码就是构造 MyProject 的类文件 lib 的类路径条目：

```
IClassPathEntry libEntry = JavaCore.newLibraryEntry(new
Path("/MyProject/lib"), null, null, false);
```

以下类路径条目具有源代码连接：

```
IClassPathEntry libEntry = JavaCore.newLibraryEntry(
new Path("d:/lib/foo.jar"), // jar 包路径
new Path("d:/lib/foo_src.zip"), // jar 包的源码包的路径
new Path("src"), // 源归档根路径
true);
```

设定关联源代码包有利于代码的跟踪调试。

③ 依赖项目。使用 JavaCore.newProjectEntry 方法。下面的代码就是构造依赖项目 MyFramework：

```
IClassPathEntry prjEntry = JavaCore.newProjectEntry(new
Path("/MyFramework"), true);
```

④ 类路径变量。使用 `JavaCore.newVariableEntry` 方法。类路径变量对于整个工作空间来说是全局的，并且可以通过 `JavaCore` 方法 `getClasspathVariable` 和 `setClasspathVariable` 来处理。

可能会注册自动的类路径变量初始化方法，当启动工作空间时，通过扩展点 `org.eclipse.jdt.core.classpathVariableInitializer` 来调用该类路径变量初始化方法。

以下类路径条目指示一个库，该库的位置存放在变量 `HOME` 中。使用变量 `SRC_HOME` 和 `SRC_ROOT` 来定义源代码连接：

```
IClassPathEntry varEntry = JavaCore.newVariableEntry(
    new Path("HOME/foo.jar"), //库路径
    new Path("SRC_HOME/foo_src.zip"), //源码归档路径
    new Path("SRC_ROOT"), //源码归档根路径
    true);
JavaCore.setClasspathVariable("HOME", new Path("d:/myInstall"), null);
```

⑤ 类路径容器。通过 `JavaCore` 的 `getClasspathContainer` 和 `setClasspathContainer` 两个方法来处理类路径容器。

可能会注册一个自动的类路径容器初始化方法，当需要绑定容器时，通过扩展点 `org.eclipse.jdt.core.classpathContainerInitializer` 来被动地调用类路径容器初始化方法。

以下类路径条目指示系统类库容器：

```
IClassPathEntry varEntry = JavaCore.newContainerEntry(
    new Path("JDKLIB/default"), false);

JavaCore.setClasspathContainer(
    new Path("JDKLIB/default"),
    new IJavaProject[]{ myProject },
    new IClasspathContainer[] {
        new IClasspathContainer() {
            public IClasspathEntry[] getClasspathEntries() {
                return new IClasspathEntry[] {
                    JavaCore.newLibraryEntry(
                        new Path("d:/rt.jar"), null, null, false);
                };
            }
            public String getDescription() {
                return "Basic JDK library container";
            }
            public int getKind() { return IClasspathContainer.K_SYSTEM; }
            public IPath getPath() { return new Path("JDKLIB/basic"); }
        }
    },
    null);
```

我们只要调用相应的方法创建我们的类路径条目就可以了，然后把这些条目组成的数组通过 `setRawClasspath` 方法设定到项目中。需要注意的是如果我们只把要添加的类路径条目传入 `setRawClasspath` 方法的话，就会替换原有的项目构建路径，这常常是我们不希望的。可以调用 `IJavaProject` 的 `readRawClasspath` 方法读取项目已有的设置，把我们要设置的构建路径添加到它的后面，然后再调用 `setRawClasspath` 方法设定新的项目构建路径。

3.7.4 设定构建路径实战

在这个例子中，将要实现一个“为项目添加 lucene 支持”的功能，用户在项目上右击，选择菜单中的【为项目添加 lucene 支持】命令以后，插件就会把 lucene 的 jar 包和源码包复制到项目的 lib 目录下，并且将 jar 包加入构建路径。如图 3.17 所示为增加 lucene 支持前的项目结构。



图 3.17 增加 lucene 支持之前的项目结构

用户在项目上右击，在弹出的快捷菜单中选择【为项目添加 lucene 支持】命令后的项目结构如图 3.18 所示。

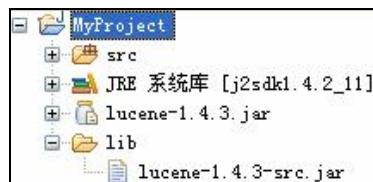


图 3.18 增加 lucene 支持之后的项目结构

图 3.19 是项目的构建路径。

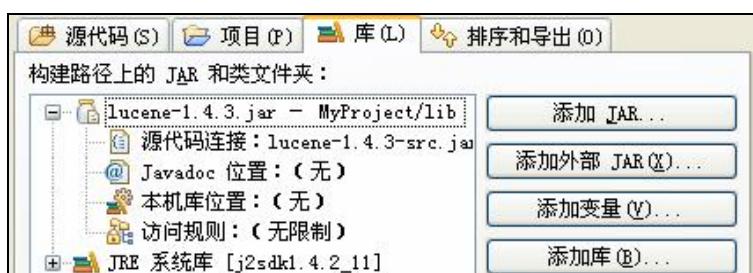


图 3.19 增加的 lucene 包

首先新建一个插件工程，并将 JDT 相关的依赖项加入。然后添加一个 `org.eclipse.ui.popupMenus` 的扩展点，如果不熟悉怎么添加，可以使用插件向导中的“弹出菜单”向导。

需要注意 contribution 的配置，如图 3.20 所示。



图 3.20 contribution 的配置

此插件只针对 Java 项目起作用，因此 objectClass 中填入 org.eclipse.jdt.core.IJavaProject; adaptable 选择 true; 如果是用向导生成的那么请记住清空 nameFilter。下面是核心类 ActionAddLucene 的实现代码。

```
public class ActionAddLucene implements IObjectActionDelegate
{
    private static final String FILESEPARATOR =
        System.getProperty("file.separator", "/");
    private static final String LUCENESRCJAR = "lucene-1.4.3-src.jar";
    private static final String LUCENEJAR = "lucene-1.4.3.jar";
    private static final String LIB = "lib";
    private static final String RESOUCELIB = "resoucelib";
    private IStructuredSelection structSelection;

    public ActionAddLucene()
    {
        try
        {
            URL luceneSrcJarURL = new URL("file://" + FILESEPARATOR + LIB + FILESEPARATOR + LUCENESRCJAR);
            URL luceneJarURL = new URL("file://" + FILESEPARATOR + LIB + FILESEPARATOR + LUCENEJAR);
            URL resouceLibURL = new URL("file://" + FILESEPARATOR + LIB + FILESEPARATOR + RESOUCELIB);
            URL[] urls = {luceneSrcJarURL, luceneJarURL, resouceLibURL};
            URLClassLoader classLoader = new URLClassLoader(urls);
            Class luceneClass = classLoader.loadClass("org.apache.lucene.analysis.standard.StandardAnalyzer");
            Class luceneAnalyzerClass = luceneClass.getConstructor(new Class[] {String.class});
            luceneAnalyzerClass.newInstance(new Object[] {stopWords});
            structSelection = new StructuredSelection(luceneAnalyzerClass);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```

        super();
    }

    public void setActivePart(IAction action, IWorkbenchPart targetPart)
    {
    }

    public void run(IAction action)
    {
        Object selectObj = structSelection.getFirstElement();
        if (selectObj instanceof IProject)
        {
            IProject project = (IProject) selectObj;
            IJavaProject JavaProject = JavaCore.create(project);
            IClasspathEntry[] oldPaths = JavaProject.readRawClasspath();
            IClasspathEntry luceneLibEntry =
                JavaCore.newLibraryEntry(project
                    .getFile(LIB + FILESEPARATOR + LUCENEJAR)
                    .getFullPath(), project
                    .getFile(LIB + FILESEPARATOR +
LUCENESRCJAR).getFullPath(), null,
                    false);

            if(classPathExists(oldPaths,luceneLibEntry))
            {
                return;
            }

            URL luceneLib = Activator.getDefault().getBundle().getEntry(
                RESOUCELIB + FILESEPARATOR + LUCENEJAR);
            URL luceneSrc = Activator.getDefault().getBundle().getEntry(
                RESOUCELIB + FILESEPARATOR + LUCENESRCJAR);
            IClasspathEntry[] newPaths =
                new IClasspathEntry[oldPaths.length + 1];
            System.arraycopy(oldPaths, 0, newPaths, 0, oldPaths.length);

            IFolder libFolder = project.getFolder(LIB);
            if (!libFolder.exists())
            {
                try
                {
                    libFolder.create(true, true, null);
                } catch (CoreException e)
                {
                    handleException(e);
                }
            }

            copyURLToFile(luceneLib, project,
                LIB + FILESEPARATOR + LUCENEJAR);
            copyURLToFile(luceneSrc, project,
                LIB + FILESEPARATOR + LUCENESRCJAR);

            newPaths[oldPaths.length] = luceneLibEntry;
            try
            {
                JavaProject.setRawClasspath(newPaths, null);
            }
        }
    }
}

```

```

        } catch (JavaModelException e)
        {
            handleException(e);
        }
    }

private static boolean classPathExists(IClasspathEntry[] entrys,
    IClasspathEntry entry)
{
    for(int i=0,n=entrys.length;i<n;i++)
    {
        if(entrys[i].getPath().equals(entry.getPath()))
        {
            return true;
        }
    }
    return false;
}
private static void handleException(Exception e)
{
    Activator.getDefault().getLog().log(
        new Status(IStatus.ERROR, Activator.PLUGIN_ID, 0,
            e.getMessage(), e));
}

private static void copyURLToFile(URL url, IProject project,
    String destFileName)
{
    InputStream inStream = null;
    try
    {
        inStream = url.openStream();
        IFile file = project.getFile(destFileName);
        if (!file.exists())
        {
            file.create(inStream, true, null);
        }
    } catch (IOException e)
    {
        handleException(e);
    } catch (CoreException e)
    {
        handleException(e);
    } finally
    {
        try
        {
            if (inStream != null)
                inStream.close();
        } catch (IOException e)
        {
            handleException(e);
        }
    }
}

```

```

public void selectionChanged(IAction action, ISelection selection)
{
    structSelection = (IStructuredSelection)selection;
}
}

```

下面解释一下代码中的重点部分。

- (1) IClasspathEntry[] oldPaths = JavaProject.readRawClasspath();
读取项目原有的构建路径条目。
- (2) IClasspathEntry luceneLibEntry = JavaCore.newLibraryEntry(
 project.getFile(LIB + FILESEPARATOR + LUCENEJAR).getFullPath(),
 project.getFile(LIB + FILESEPARATOR + LUCENESRCJAR).getFullPath(),
 null, false);

这一句构建 lucene 的 jar 包。

第 1 个参数是二进制 jar 包的位置，我们的二进制 jar 包的位置为项目路径下的 lib/lucene-1.4.3-src.jar。

第 2 个参数是 jar 包对应的源码包的位置。

第 3 个参数为源码包的根路径，因为有的源码 jar 包的源码根路径不是 jar 包的根路径，比如 simplejta 的源码 jar 包的格式如图 3.21 所示。

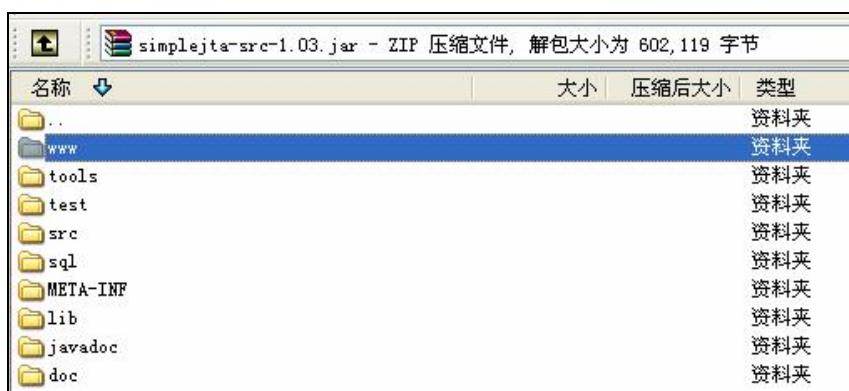


图 3.21 Jar 包的结构

对于这种情况就要指定第 2 个参数为“src”，lucene 的源码包的源码根路径就是 jar 包的根路径，因此我们设置此参数为 null。

第 4 个参数表示是否导出，我们设置为 false。

- (3) URL luceneLib = Activator.getDefault().getBundle().getEntry(RESOUCELIB+ FILE- SEPARATOR + LUCENEJAR);

我们把“lucene-1.4.3.jar”、“lucene-1.4.3-src.jar”放到我们插件的“resoucelib”目录下，当用户单击【为项目添加 lucene 支持】的时候要把这两个文件复制到项目的 lib 目录下，因此需要首先读取插件路径“resoucelib”目录下的这两个 jar 包。

读取插件路径下的文件时我们使用插件 Activator 类提供的方法即可，比如如下调用：

```
Activator.getDefault().getBundle().getEntry("config/my.xml");
```

就可以读取到插件根目录下的文件“config/my.xml”，返回类型是 Java.net.URL。

- (4) copyURLToFile(luceneLib, project, LIB + FILESEPARATOR + LUCENEJAR);

Activator.getDefault().getBundle().getEntry 读取到的文件位置是 URL 类型的，我们需要把这个 URL 对应的文件复制到项目的 lib 下。下面看一下 copyURLToFile 的主干代码：

```

inStream = url.openStream();
IFile file = project.getFile(destFileName);
if (!file.exists())
{
    file.create(inStream, true, null);
}

```

```
}
```

URL 类有一个 `openStream` 可以打开文件的输入流, IFile 也有一个接受输入流的 `create` 方法用来创建文件, 因此我们只需要把 url 的输入流输出给 IFile 的 `create` 方法即可。

这里我们也可以由 url 得到其对应的磁盘上的路径, 也可以得到 IFile 对应的磁盘上的路径, 然后使用 Java IO 来进行文件复制操作。但是这样做不仅代码数量变多了, 而且由于使用的不是 Eclipse 的资源管理 API, 会带来无法自动刷新等问题, 因此建议尽量使用 Eclipse 提供的 API 来完成此功能。

3.7.5 如何研读 JDT 代码

学习 Eclipse 插件开发的最好的方式就是研读 Eclipse 的源码, 而对插件开发者最有参考价值的就是 JDT(Java Development Tools)的代码, 相信把所有的包研读一遍以后就会成为插件开发的高手了。下面是各个主要包的内容, 读者可以根据需要有选择地进行研读。

- | org.eclipse.jdt.ui——提供用于在用户界面中显示 Java 元素的支持类。
- | org.eclipse.jdt.ui.actions——提供 JDT 的大部分菜单和按钮的相关代码, 为针对 Java 用户界面行为的应用程序编程接口。
- | org.eclipse.jdt.ui.refactoring——重构相关的类。
- | org.eclipse.jdt.ui.search——搜索相关的类。
- | org.eclipse.jdt.ui.text——提供用于显示 Java 文本的支持类。
- | org.eclipse.jdt.ui.text.folding——Java 代码编辑器的代码折叠的实现代码。
- | org.eclipse.jdt.ui.text.java——Java 代码编辑器代码补全的实现代码。
- | org.eclipse.jdt.ui.text.Java.hover——Java 编辑器中显示文本悬浮的实现代码。
- | org.eclipse.jdt.ui.wizards——创建和配置 Java 元素的向导页。

3.8 插件开发常见的问题

3.8.1 InvocationTargetException 异常的处理

`InvocationTargetException` 是一种包装由调用方法或构造方法所抛出异常的受查异常。这个异常并不是 Eclipse 插件开发特有的, 而是标准 JDK 中的, 它定义在 `java.lang.reflect` 包下。在进行 Java 开发的时候很少会接触到这个异常, 不过在进行 Eclipse 插件开发中则不同, 很多 API 都声明抛出此类异常, 因此必须对此异常进行处理。

例如, 我们开发一个方法用来统一处理异常:

```
private static void handleException(Exception e)
{
    MessageDialog.openError(Activator.getDefault().getWorkbench()
        .getDisplay().getActiveShell(), "error", e.getMessage());
    e.printStackTrace();
}
```

我们发现当传递来的参数 `e` 为 `InvocationTargetException` 的时候弹出的对话框中的消息是空的, 查看 `InvocationTargetException` 的源码得知 `InvocationTargetException` 并没有覆盖 `getMessage` 方法, 所以消息当然是空的了。我们需要调用 `InvocationTargetException` 的 `getTargetException` 方法得到要被包装的异常, 这个异常才是真正我们需要的异常。修改代码如下所示:

```
private static void handleException(Exception e)
{
    String msg = null;
    if (e instanceof InvocationTargetException)
    {
        Throwable targetEx = ((InvocationTargetException) e)
            .getTargetException();
        msg = targetEx.getMessage();
    }
}
```

```

        .getTargetException();
    if (targetEx != null)
    {
        msg = targetEx.getMessage();
    }
} else
{
    msg = e.getMessage();
}
MessageDialog.openError(Activator.getDefault().getWorkbench()
    .getDisplay().getActiveShell(), "error", msg);
e.printStackTrace();
}
}

```

3.8.2 Adaptable 与 Extension Object/Interface 模式

Java 是强类型的静态语言，这种特性在给我们的系统带来稳定性的同时也给我们系统的灵活性带来了问题，而如果 Java 能吸取动态语言的一些特性，则能够使得系统架构变得更加灵活。下面举一个例子。

由于某种需要，我们编写了一个 CowNewFile 类来表示磁盘文件，代码如下：

```

public class CowNewFile
{
    private String file;
    public CowNewFile(String file)
    {
        super();
        this.file = file;
    }
    public String getFile()
    {
        return file;
    }
}

```

当使用此类与 Java IO 类库交互的时候出现了问题，因为 Java IO 只能接受 Java.io.File 类型的文件，因此为 CowNewFile 类增加一个方法 toJavaFile，用这个方法来将 CowNewFile 转化成 Java.io.File 类型；当我们使用此类与 Java 的 Net 类库交互的时候又出现了问题，因为 Java Net 只接受 URL 类型的格式，因此为 CowNewFile 类增加一个方法 toURL，用这个方法来将 CowNewFile 转化成 Java.net.URL 类型；我们想使用一个能自动显示类的属性的第三方开源类，不过这个类要求类必须实现它的 IPropertyDescriptor 接口，因此又让类实现了 IPropertyDescriptor 接口；我们想使用一个另外项目组同事开发的一个接口来进行值校验，不过这个接口要类实现 IValidator 接口，因此我们又让类实现了 IValidator 接口……。

随着系统的进化 CowNewFile 增加了很多 toJavaFile 这样的方法，实现了很多 IPropertyDescriptor 这样的接口，代码变得臃肿无比，还可能出现接口的命名冲突、实现冲突等一系列的问题。

面向对象思想中有一条重要原则，那就是单一职责原则(Single Responsibility Principle, SRP)。就一个类而言，应该仅有一个引起其变化的原因。大致的意思就是要把职责分配到不同的对象当中去。如果我们把每一个职责的变化看成是变化的一个轴线，当需求变化时，该变化会反映为类的职责的变化。如果一个类承担了多个职责，那么引起它变化的原因就多于一个轴线，也就是把这些职责耦合到了一起，当变化发生时，设计会遭受到意想不到的破坏。

在上边这个例子中的 CowNewFile 类承担了过多的责任：要能把自己转化成 File、URL 等与自己没有直接关系的对象，要让自己有显示类属性的能力、自我校验的能力等。该是对其动刀子的时候了。

(1) 定义接口 IAdaptable：

```

public interface IAdaptable
{
    public Object getAdapter(Class adapter);
}

```

```
}
```

(2) 定义类 CowNewFilePropertyDescription, 实现 IPropertyDescriptor 接口:

```
class CowNewFilePropertyDescription implements IPropertyDescriptor
{
    private CowNewFile file;
    public CowNewFilePropertyDescription(CowNewFile file)
    {
        super();
        this.file = file;
    }

    public String[] getPropertyValues()
    {
        return new String[] { "file=" + file };
    }
}
```

(2) 定义类 CowNewFileValidator, 实现 IValidator 接口:

```
class CowNewFileValidator implements IValidator
{
    private CowNewFile cnFile;
    public CowNewFileValidator(CowNewFile file)
    {
        super();
        this.cnFile = file;
    }

    public boolean isValidate()
    {
        String fileStr = cnFile.getFile();
        if (fileStr == null || fileStr.trim().length() <= 0)
        {
            return false;
        }
        File file = new File(fileStr);
        return file.exists();
    }
}
```

(3) 修改 CowNewFile 类:

```
public class CowNewFile implements IAdaptable
{
    private String file;

    public CowNewFile(String file)
    {
        super();
        this.file = file;
    }

    public String getFile()
    {
        return file;
    }

    public Object getAdapter(Class adapter)
    {
```

```

        if (adapter.equals(File.class))
        {
            return new File(file);
        } else if(adapter.equals(URL.class))
        {
            try
            {
                return new File(file).toURL();
            } catch (MalformedURLException e)
            {
                throw new IllegalArgumentException(e);
            }
        } else if (adapter.equals(IPropertyDescriptor.class))
        {
            return new CowNewFilePropertyDescriptor(this);
        } else if (adapter.equals(IValidator.class))
        {
            return new CowNewFileValidator(this);
        }
        return null;
    }
}

```

(4) 编写测试代码:

```

CowNewFile cnFile = new CowNewFile("c:/test.txt");
File file = (File) cnFile.getAdapter(File.class);
try
{
    InputStream inStream = new FileInputStream(file);
} catch (FileNotFoundException e)
{
    e.printStackTrace();
}

URL url = (URL) cnFile.getAdapter(URL.class);
System.out.println(url.getProtocol());

IPropertyDescriptor propDesc = (IPropertyDescriptor) cnFile
    .getAdapter(IPropertyDescriptor.class);
ObjectPropertyDispalyer.showObject(propDesc);

IValidator validator = (IValidator) cnFile.getAdapter(IValidator.class);
DataValidator.validate(validator);

```

可以看到整个改造的核心就是 `IAdaptable` 接口，这个接口把从 `CowNewFile` 拆分出去的职责类适配成 `CowNewFile`。我们交互的时候还是和 `CowNewFile` 类交互，不用意识到其他类的存在，直接向它索取 `IPropertyDescriptor` 接口等接口或者类即可，`CowNewFile` 再把这些请求转发给合适的类。

`Eclipse` 中的类非常繁多，经常需要向一个类请求各种服务，因此 `Eclipse` 中就大量的使用了 `Extension Object/Interface` 模式，很多类都直接或间接的实现了 `IAdaptable` 接口(定义在 `org.eclipse.core.runtime` 包下)，这在后边的程序中将会有相应的代码实例。

看到这里您已经明白为什么在“设定构建路径实战”一节中我们要设定“`adaptable=true`”了，因为也许我们选择的对象没有实现 `IJavaProject` 接口，但是它可以通过 `getAdapter` 来适配成 `IJavaProject`，那么我们的菜单也应该可以使用。从这一点可以看出我们的代码写得是有错误的，因为直接把选择的对象转换成了 `IJavaProject`，正确的方式应该是首先判断它是不是 `IJavaProject` 的实例，如果不是的话再判断能否通过

getAdapter 适配成 IJavaProject, 这一点工作就留给读者去完成吧。

3.8.3 千万不要使用 internal 包

Eclipse 源码中有很多包名中含有 internal 的类(比如 org.eclipse.ui.internal.EditorSite)，这些类是 Eclipse 的内部实现，不是向外提供的 API，Eclipse 不保证在后续版本中不对这些类进行修改。这些类都通过公共的 API 向外提供调用它的接口，完全没有必要调用它们，很多 Eclipse 插件在新版本 Eclipse 中无法运行的原因大部分都是因为这些插件使用了 internal 包中的类，而这些类在新版本没有继续提供。

3.8.4 打开视图

在程序中我们有的时候需要得到某个视图或者打开某个视图，方法如下：

```
IWorkbenchPage page = Activator.getActivePage();
IViewPart part = page.findView("org.eclipse.ui.views.PropertySheet");
```

这样就得到了视图的实例 part，如果需要此视图特有的方法，则需要把 part 强制转换成相应的视图对象。注意 findView 中的参数表示视图的 id，而不是视图的类名。findView 只对打开的视图有效，如果视图没有打开则返回值为 null，需要调用代码去打开这个视图：

```
part = page.showView("org.eclipse.ui.views.PropertySheet");
```

3.8.5 查找扩展点的实现插件

插件在启动的时候都向平台扩展注册表中注册自己，因此如果需要查找实现了某个扩展点的插件，我们只要向平台扩展注册表(通过 Platform.getExtensionRegistry()方法取得)发出请求即可，下面的代码是查找实现了“视图”扩展点的实现插件：

```
IExtensionRegistry registry = Platform.getExtensionRegistry();
IExtensionPoint point =
    registry.getExtensionPoint("org.eclipse.ui.views");
if (point == null) return;
IExtension[] extensions = point.getExtensions();
```

这里的 extensions 就是所有的“视图”扩展点的插件信息数组。

3.8.6 项目 nature

Eclipse 中的项目有很多种，比如 Java 项目、C++项目、WTP 项目、Python 项目，这些项目的不同之处是每种项目都有自己不同的特性，为了方便地标识和辨认这些特性，Eclipse 为项目维护了一个特性标识数组，插件只要读取这个数组就可以知道此项目是否拥有某个特性。比如 Java 相关的插件如果在非 Java 项目中被调用，由于项目没有 Java 特性，所以调用就会不成功了。一些插件还可以改变项目的 nature，比如 SpringIDE 就提供了一个为项目增加 Spring 特性的功能。为项目增加特性不仅涉及到修改项目的特性标识数组，还会进行项目的初始化等操作，不过本例中只讲解如何修改项目的 nature。下面就演示如何为项目增加一个“cownewStudio”的 natureId：

```
IProject project = ... ;
IProjectDescription projectDesc = project.getDescription();
String[] oldIds = projectDesc.getNatureIds();
String[] newIds = new String[oldIds.length];
System.arraycopy(oldIds, 0, newIds, 0, oldIds.length);
newIds[oldIds.length] = "cownewStudio";
projectDesc.setNatureIds(newIds);
```

3.8.7 透视图开发

经常听到一些刚刚学会插件开发的朋友这样说：我会做 Eclipse 插件了，不过前面的路看起来好长呀，我现在做的只是一个小窗口而已，什么时候才能做一个透视图呀，那样我的插件就看起来像模像样了。

在很多人心中，Eclipse 中的“透视图”是个比视图、编辑器等更高级的东西，因为一个透视图中经常管理着大量的菜单、视图、编辑器等，因此也就认为透视图开发难度非常大。

透视图包含一组视图和编辑器并可以方便地对它们进行布局，其实透视图做的工作并不多，大部分工作都是由菜单、视图、编辑器等来完成的，透视图的作用只是将一些视图打开并摆好位置、显示菜单、添加快捷键等工作，并没有做任何与功能相关的操作。

下面演示一个“演示透视图”的开发，在这个透视图中我们打开“包资源管理器”、“JavaDoc”、“属性”三个视图。

新建一个扩展，透视图从 org.eclipse.ui.perspectives 扩展点扩展，配置文件部分如下：

```
<extension
    point="org.eclipse.ui.perspectives">
    <perspective name="演示透视图"
        icon="icons/emf.ico"
        class="cownew.cownew.perspectiveTest.TestPerspective"
        id="cownew.cownew.perspectiveTest.TestPerspective">
    </perspective>
</extension>
```

新建一个类 TestPerspective，实现 IPerspectiveFactory 接口，代码如下：

```
package cownew.cownew.perspectiveTest;

import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

public class TestPerspective implements IPerspectiveFactory
{
    public void createInitialLayout(IPageLayout layout)
    {
        layout.addView("org.eclipse.jdt.ui.PackagesView",
                       IPageLayout.LEFT, 0.3f, IPageLayout.ID_EDITOR_AREA);
        layout.addView("org.eclipse.jdt.ui.JavadocView", IPageLayout.TOP,
                      0.3f, IPageLayout.ID_EDITOR_AREA);
        layout.addView("org.eclipse.ui.views.PropertySheet",
                      IPageLayout.BOTTOM, 0.3f,
                      IPageLayout.ID_EDITOR_AREA);
    }
}
```

IPageLayout 的 addView 方法的作用是向透视图中添加视图，它需要以下 4 个参数：

- | 视图的唯一标识，与 plugin.xml 中定义的一致。
- | 参考部分中的相对位置，可以是 IPageLayout.TOP、IPageLayout.BOTTOM、IPageLayout.LEFT 或 IPageLayout.RIGHT。
- | 参考部分中当前占有的空间比率，值范围在 0.05f~0.95f 之间。
- | 参考部分唯一标识；例中使用的是编辑区域(IPageLayout.ID_EDITOR_AREA)。

运行以后，效果如图 3.22 所示。

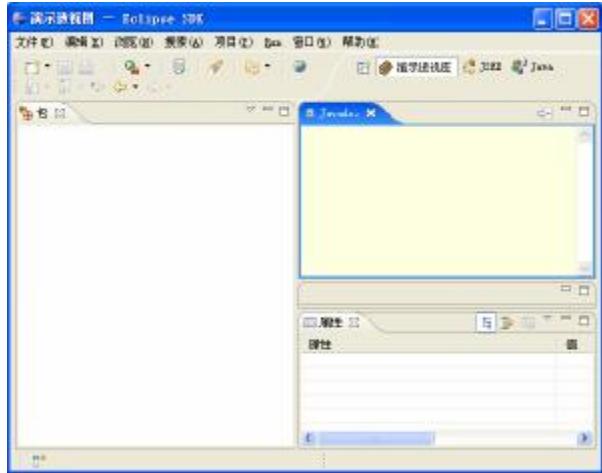


图 3.22 演示透视图

还可以调用 `addActionSet` 等方法向透视图中添加菜单、工具条等。

3.8.8 关于工具条路径

在运行 Eclipse 插件向导中的“Hello, World!”向导的时候，很多读者都会提出这样的问题：菜单和工具栏按钮只能显示在那个地方吗？能不能把菜单项加入主菜单的【文件】选项下呢？

下面来看一下 `plugin.xml` 文件。

```

<actionSet
    label="样本操作集"
    visible="true"
    id="PlugTest.actionSet">
    <menu
        label="样本菜单(&M)"
        id="sampleMenu">
        <separator
            name="sampleGroup">
        </separator>
    </menu>
    <action
        label="样本操作(&S)"
        icon="icons/sample.gif"
        class="plugtest.actions.SampleAction"
        tooltip="Hello, Eclipse world"
        menubarPath="sampleMenu/sampleGroup"
        toolbarPath="sampleGroup"
        id="plugtest.actions.SampleAction">
    </action>
</actionSet>

```

`menu` 标记中定义了一个新的菜单，并且用 `separator` 定义了子菜单。

`action` 标记中 `menubarPath` 定义了此 `action` 对应的菜单项的路径，`toolbarPath` 定义了此 `action` 对应的工具条的路径，这两个属性的值都为本插件特有的，只要修改它们就可以把它们放到其他位置。Eclipse 的标准菜单都定义了固定的 path，只要把我们的 `action` 的 path 指向它们即可，下面来稍作修改：

```

<actionSet
    label="样本操作集"
    visible="true"
    id="PlugTest.actionSet">
    <action
        label="样本操作(&S)"
```

```

icon="icons/sample.gif"
class="plugtest.actions.SampleAction"
tooltip="Hello, Eclipse world"
menubarPath="file/fileStart"
toolbarPath="org.eclipse.ui.workbench.file/new.ext"
id="plugtest.actions.SampleAction">
</action>
</actionSet>

```

运行后效果图如图 3.23 所示，可以看到菜单已经放到了我们期望的位置。

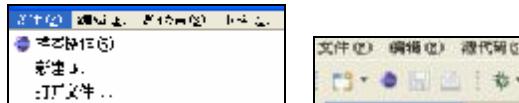


图 3.23 运行后的效果图

(1) 下面是常见的菜单的菜单路径。

- | file/fileStart: 【文件】的开始区。
- | file/new/additions: 【文件】的【新建】菜单内部的【附加】组。
- | file/new.ext: 【文件】的【新建】区。
- | file/close.ext: 【文件】的【关闭】区。
- | file/save.ext: 【文件】的【保存】区。
- | file/print.ext: 【文件】的【打印】区。
- | file/open.ext: 【文件】的【打开】区。
- | file/import.ext: 【文件】的【导入】区。
- | file/additions: 【文件】的【附加】区。
- | file/mru: 【文件】的【最近的文档】区。
- | file/fileEnd: 【文件】的【结束】区。
- | edit/editStart: 【编辑】的【开始】区。
- | edit/undo.ext: 【编辑】的【撤销】区。
- | edit/cut.ext: 【编辑】的【剪切】区。
- | edit/find.ext: 【编辑】的【查找】区。
- | edit/add.ext: 【编辑】的【添加】区。
- | edit/fileEnd: 【编辑】的【结束】区。
- | edit/additions: 【编辑】的【附加】区。
- | org.eclipse.jdt.ui.refactoring.menu: 【重构】区。
- | project/projStart: 【项目】的【开始】区。
- | project/open.ext: 【项目】的【打开】区。
- | project/build.ext: 【项目】的【建立】区。
- | project/additions: 【项目】的【附加】区。
- | project/projEnd: 【项目】的【结束】区。
- | org.eclipse.ui.run: 【运行】区。
- | window/additions: 【窗口】的【附加】区。
- | window/additionsend: 【窗口】的【结束】区。
- | help/helpStart: 【帮助】的【开始】区。
- | help/group.main.ext: 【帮助】的【主要组】区。
- | help/group.tutorials: 【帮助】的【教程组】区。
- | help/group.tools: 【帮助】的【工具组】区。
- | help/group.updates: 【帮助】的【更新组】区。
- | help/helpEnd: 【帮助】的【结束】区。
- | help/additions: 【帮助】的【附加】区。
- | help/group.about.ext: 【帮助】的【关于】区。

(2) 下面是常见的工具条的工具条路径。

- | org.eclipse.ui.workbench.file/new.ext: 【文件】的【新建】区。
- | org.eclipse.ui.workbench.file/save.ext: 【文件】的【保存】区。
- | org.eclipse.ui.workbench.file/print.ext: 【文件】的【打印】区。
- | org.eclipse.ui.workbench.file/build.ext: 【文件】的【建立】区。
- | org.eclipse.ui.workbench.navigate: 【导航】区。
- | org.eclipse.debug.ui.launchActionSet: 【启动】区。
- | org.eclipse.search.searchActionSet: 【搜索】区。

3.8.9 Eclipse 的日志

在调试插件的时候经常遇到插件运行异常的情况, Eclipse 默认并不会将异常打印到控制台而是记录到它的日志系统中去; 我们把插件给用户使用的时候也许出现各种问题, 这时候我们最希望得到的就是其系统日志, 这样可以对插件的运行状况进行分析。查看日志的方法如下。

选择【窗口】|【显示视图】|【其他】|【PDE 运行时】|【错误日志】命令, 在这个视图中就显示了系统的日志。



图 3.24 错误日志视图

每一条消息就是一条日志, 双击每一条就可以查看消息的详细信息(对于异常消息一般显示的是异常堆栈), 如图 3.25 所示。

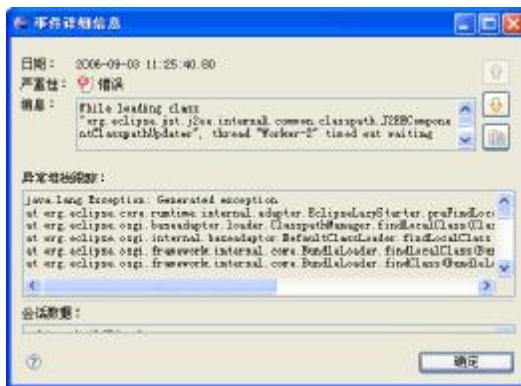


图 3.25 事件详细信息

还可以单击导出日志”图标  将日志导出成文件。建议在开发调试插件的时候打开此视图, 以便及时发现系统运行异常。

第 6 章 基于 GEF 的界面设计工具^①

在有的情况下, 像 CowNewStudio 这样的普通界面的编辑器是不能满足要求的, 比如工作流编辑器、界面设计器、UML 建模工具等, 如果要完成这样功能的插件就必须使用图形界面来完成。为了方便开发图形

化的插件, Eclipse 提供了 GEF 框架这个图形编辑框架, 使用 GEF 可以很容易地实现一个实用的图形化编辑器。本章我们将会以一个界面设计工具来讲解 GEF 的使用, 用户可以很容易地将这个界面设计工具改造为报表设计器、工作流编辑器等实用的工具。

6.2 系统需求

本章选择界面设计工具作为 GEF 的开发案例, 这主要是考虑到报表设计器、工作流编辑器等与具体的业务结合过于紧密, 在开发过程中会涉及到很多与 GEF 没有直接关系的东西, 而界面设计器则相对比较单纯, 并且会涉及到 GEF 中大部分主流的内容, 我们可以根据需要很容易地将界面设计器改造成符合特定业务需要的工具。

6.2.1 界面设计工具的分类

以往一切界面代码都是要开发人员手工书写, 这无疑增加了开发难度, Delphi、VB 等工具的出现扭转了这个局面, 使用这些工具开发人员只要在组件面板上拖拖拽拽就可以完成界面的设计, 做到了“所见即所得”的开发方式。按照界面的保存方式来划分, GUI 设计工具可以分成如下 3 类:

- | 基于界面文件的纯代码生成方式。
- | 代码生成与界面文件结合的方式。
- | 无界面文件方式。

(1) 基于界面文件的纯代码生成: NetBeans 是这类工具的典型代表, NetBeans 中与界面设计有关的有两种文件: *.form 文件和*.java 文件。*.form 文件中是以 XML 格式描述界面布局和组件的属性等信息; *.java 文件则是通过解析*.form 文件生成的代码, 生成的界面代码主要位于 initComponents 方法中, 这个方法在 NetBeans IDE 中是无法手工编辑的。在用户拖拉组件的时候, NetBeans 就将拖拉的组件描述增加到*.form 文件中, 并且即时将新的代码生成到*.java 文件中。这样实现的好处有如下几点: IDE 实现容易, IDE 的开发人员只要关注于如何将界面信息转化为*.form 文件和如何将*.form 文件解析生成*.java 代码即可, 无需关心用户修改*.java 代码造成的反向解析问题; *.java 文件可以脱离*.form 而存在, 也就是*.form 文件只是在设计期有意义, 而在运行期是无用的。其缺点是用户无法手工修改生成的代码。

(2) 代码生成与界面文件结合: Delphi 和 VB 是这类工具的典型代表。以 Delphi 为例, 在 Delphi 中新建以后界面以后将会存在两个文件: .dfm 和.pas, .dfm 描述了界面布局和组件的属性等信息, .pas 则定义了组件的变量和事件处理函数。在编译的时候.dfm 被编译到可执行文件中, 运行的时候动态解析.dfm 文件来构建界面。与 NetBeans 不同的就是.dfm 文件是有运行期的意义的, 如果没有.dfm 文件, 程序将无法编译运行。这样的方式通常只适用于 Delphi、VB 这样代码和 IDE 结合过于紧密的语言, 很难将生成的代码进行手工修改。

(3) 无界面文件方式: Eclipse 的 Visual Editor 是最经典的例子。使用 Visual Editor 进行 GUI 绘制的时候, 只存在一个*.java 文件, Visual Editor 将用户绘制的界面直接解析为*.java 代码, 如果用户修改了*.java 代码, Visual Editor 会运行一个虚拟机, 在虚拟机中运行用户修改后的*.java 文件, 然后就可以得到运行时的程序界面并将这个界面绘制到窗口设计器中了。这样做可以将所有的界面信息都集成到一个文件中, 并且支持用户手工修改生成的代码; 由于设计器中的界面是通过另外一个虚拟机运行而得到的, 在界面设计器中看到的界面就是运行时的界面, 这样保证了真正的“所见即所得”。这样做的坏处也是明显的, 由于需要重新启动一个虚拟机, 导致了速度很慢, 资源占用比较高, 使用 Visual Editor 的时候经常造成 Eclipse 内存不足退出。

由于 HTML 代码本身就是一个树状模型, 无需进行代码和模型间的转换, 所以网页设计器就不存在上面所说的这些类别了。

为了降低开发难度, 我们在案例系统中采用基于界面文件的纯代码生成方式, 用户绘制的界面信息保存在单独的*.ui 文件中, 在用户修改界面并保存的时候, 我们自定义的界面构建器就会被启动, 界面构建器会读取并解析*.ui 文件然后生成*.java 文件。如果今后由于系统需求(主要是要求允许开发人员修改生成的代码)需要改用无界面文件方式的话就可以借鉴 Visual Editor 的思路。不过如果完全采用 Visual Editor 的无界面文件方式的话会导致资源占用太大, 因此可以采用了另外一种思路, 也就是在内存中为每个界面维护一个对象模型(树状结构), 在用户绘制界面的时候去修改这个对象模型, 在用户保存界面的时候去解析这个对象模型生成*.java 源代码; 在由*.java 源代码加载绘制设计器中的界面的时候, 首先通过解析*.java 源代码生成源代码的抽象语法树(将*.java 源代码解析为抽象语法树的过程可以使用 JDT 来完成), 然后解析这个抽象语法树

生成界面的对象模型，这样就可以很轻松地绘制界面了。这样做不仅有 Visual Editor 的优点，而且占用资源比较小；不过由于手工修改代码的千差万别，如果开发人员修改的代码采用了比较生僻的语法，有可能造成用户修改的代码无法正确地解析为对象模型，造成*.java 源代码加载绘制设计器中的界面的时候发生异常，解决这个问题的唯一一个办法就是建议开发人员尽量采用常用的代码来修改生成的界面代码。

6.2.2 功能描述

作为界面设计器必须的功能，系统必须提供对按钮、编辑框、标签、复选按钮、单选按钮、下拉列表框、列表框等基本组件及其基本属性的支持，为了降低系统开发的复杂性，这里不要求提供对用户自定义组件的支持，不提供对 JPanel、JScrollPane、JTabbedPane 等复合组件的支持。

用户可以将组件从组件面板中拖放到界面编辑器中，并能够对组件进行移动、缩放、删除、修改等操作，为了使得开发人员能够更加容易地对组件进行布局，编辑器必须提供对选定的多个组件进行对齐(上下左右 4 个方向)、等宽、等高等操作。

当用户将组件从组件面板中拖放到界面编辑器的时候，编辑器必须为组件提供一个默认的 Id，此 Id 不能与其他组件的 Id 重复；用户可以修改生成的 Id，但是修改后的 Id 同样不能与其他组件的 Id 重复。

界面编辑器只须提供按绝对坐标进行定位的 XYLayout 即可；当用户对界面进行修改并且保存以后会立即生成的对应的 Java 代码，Java 代码使用 Swing 图形框架进行实现。

为了降低系统的开发难度，可以将窗体模型对象直接序列化作为界面文件，当打开界面文件的时候再将此文件反序列化为窗体模型对象。这样做会导致多版本序列化问题，当模型发生较大变化的时候有可能造成系统工作不正常，因此应该考虑提供对以后使用其他方式保存界面文件的支持。

6.3 构建模型

完成一个完整的 GEF 应用通常需要以下几个步骤。

- (1) 建立模型。
- (2) 建立视图(Figure)。
- (3) 建立控制器(EditPart)。
- (4) 建立编辑策略和 Command。

可以看到建立模型是完成系统的第一步。模型是根据具体系统的需求来得到的，比如界面设计器中的模型包括代表窗口的 Form、代表按钮的 Button、代表编辑框的 Edit 和代表复选按钮的 CheckBox 等对象。模型的一个职责是负责将自身的改变通知给控制器，因此可编写一个提供这些功能的抽象类 Element，其他模型必须从 Element 继承。

【代码 6-1】 模型抽象类：

```
package com.cownew.uidesigner.model;

import Java.beans.PropertyChangeListener;
import Java.beans.PropertyChangeSupport;
import Java.io.IOException;
import Java.io.ObjectInputStream;
import Java.io.Serializable;

abstract public class Element implements Serializable
{
    static final long serialVersionUID = 1;

    transient protected PropertyChangeSupport listeners =
    new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener)
    {
```

```

        listeners.addPropertyChangeListener(listener);
    }

    protected void firePropertyChange(String prop,
        Object old, Object newValue)
    {
        listeners.firePropertyChange(prop, old, newValue);
    }

    protected void fireStructureChange(String prop, Object child)
    {
        listeners.firePropertyChange(prop, null, child);
    }

    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException
    {
        in.defaultReadObject();
        listeners = new PropertyChangeSupport(this);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener)
    {
        listeners.removePropertyChangeListener(listener);
    }
}

```

由于模型对象要被序列化保存到界面文件中，所以 Element 类要实现 Serializable 接口；Element 类中提供了 addPropertyChangeListener、firePropertyChange、fireStructureChange 和 removePropertyChangeListener 方法用来供控制器注册监听器、通知监听器模型改变、移除监听器，这几个方法使用 JDK 内置的 PropertyChangeSupport 来实现；添加的监听器有可能是无法序列化的对象，为了防止序列化的时候造成异常，这里将 listeners 声明为不可序列化的，并且编写了 readObject 方法用来改变默认的序列化方式。

表示整个界面的类为 Form，Form 用来容纳所有定义的组件。

【代码 6-2】界面模型类：

```

package com.cownew.uidesigner.model;

import Java.io.ByteArrayInputStream;
import Java.io.ByteArrayOutputStream;
import Java.io.IOException;
import Java.io.InputStream;
import Java.io.ObjectInputStream;
import Java.io.ObjectOutputStream;
import Java.util.ArrayList;
import Java.util.List;

public class Form extends Element
{
    static final long serialVersionUID = 1;

    public static String COMPONENTS = "components";

    protected List<Component> components = new ArrayList<Component>();
}

```

```

public void addComponent(Component component)
{
    components.add(component);
    fireStructureChange(COMPONENTS, components);
}

public void removeComponent(Component component)
{
    components.remove(component);
    fireStructureChange(COMPONENTS, components);
}

public List<Component> getComponents()
{
    return this.components;
}

public InputStream getAsStream() throws IOException
{
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(os);
    out.writeObject(this);
    out.close();
    InputStream istream = new ByteArrayInputStream(os.toByteArray());
    os.close();
    return istream;
}

public static Form makeFromStream(InputStream istream) throws IOException,
    ClassNotFoundException
{
    ObjectInputStream ois = new ObjectInputStream(istream);
    Form form = (Form) ois.readObject();
    ois.close();
    return form;
}
}

```

components 属性中保存的是界面中所有的组件，可以通过 addComponent 和 removeComponent 方法向界面中添加组件，也可以通过 getComponents 方法得到界面中所有的组件；为了支持将界面序列化为界面文件，Form 类提供了 getAsStream 方法用来返回界面对象序列化的流；为了将界面对象序列化的流反向解析为 Form 对象，相应地提供了静态方法 makeFromStream。getAsStream 方法和 makeFromStream 方法的实现都是简单的流操作，对 Java 中的流操作不熟悉的读者可以参考相应的资料。

系统中所有的组件都有 Id、位置等属性，为了抽象出来这些公共的属性，我们提供了组件基类 Component。

【代码 6-3】组件模型基类：

```

package com.cownew.uidesigner.model;

import Java.util.List;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.jface.viewers.ICellEditorValidator;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.IPropertySource;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;
import com.cownew.uidesigner.common.ComponentIdManager;
abstract public class Component extends Element implements IPropertySource
{

```

```
static final long serialVersionUID = 4;

public static final String BOUNDS = "BOUNDS";
public static final String ID = "ID";

private Rectangle bounds;
private String id = "component";
private Form form;

public Form getForm()
{
    return form;
}

public void setForm(Form form)
{
    this.form = form;
}

public Rectangle getBounds()
{
    return bounds;
}

public void setBounds(Rectangle bounds)
{
    this.bounds = bounds;
    firePropertyChange(BOUNDS, null, bounds);
}

public void setId(String id)
{
    if (this.id.equals(id))
    {
        return;
    }

    this.id = id;
    firePropertyChange(ID, null, id);
}

public String getId()
{
    return id;
}

public Object getEditableValue()
{
    return this;
}

public IPropertyDescriptor[] getPropertyDescriptors()
{
    TextPropertyDescriptor idDesc = new TextPropertyDescriptor(ID, "Id");
    idDesc.setValidator(new ICellEditorValidator() {

        public String isValid(Object value)
    });
}
```

```

    {
        String id = (String) value;
        ComponentIdManager idMgr = new ComponentIdManager(getForm());
        // 防止 id 重名
        if (idMgr.isIdConflicted(id, Component.this))
        {
            return "组件 Id: " + id + "已经存在";
        }
        return null;
    }

    });

IPropertyDescriptor[] descriptors = new IPropertyDescriptor[] { idDesc };
return descriptors;
}

public Object getPropertyValue(Object id)
{
    if (id.equals(ID))
    {
        return getId();
    }
    return null;
}

public boolean isPropertySet(Object id)
{
    return true;
}

public void resetPropertyValue(Object id)
{
}

public void setPropertyValue(Object id, Object value)
{
    if (id == ID)
    {
        setId((String) value);
    }
}

/**
 * 生成代码片段(生成代码的逻辑本不应该和模型混在一起, 不过由于代码生成逻辑目前不是很复杂, 所以暂时这么做
 * @param parentId 父组件的 id
 * @return 代码段, List 中每一个元素是一个类型为 String 的代码行
 */
public abstract List<String> generateCode(String parentId);
}

```

为了简化实现, 这里 Component 直接实现了 IPropertySource 接口, 而不像上一章讲的那样使用适配对象来实现 IPropertySource 接口; bounds、id 属性分别代表组件的位置和 Id; 为了能在组件中方便地引用组件所在的窗口, 将组件所在的窗口对象保存在 form 属性中。

为了校验用户输入的组件 Id 是否重复, 在 getPropertyDescriptors 方法中我们为代表 id 的属性描述器

idDesc 增加了验证器，在验证器中调用 ComponentIdManager 进行 Id 唯一性验证，关于 ComponentIdManager 的实现我们将在后边介绍。

generateCode 是用来进行代码生成的，参数 parentId 表示组件的父组件 Id，返回值为 List 类型，List 中每一个元素为一个代码行，代码行只用处理本行内的缩进即可，无需考虑其上下文环境的缩进。按照面向对象的理论，生成代码不属于模型的职责，应该将 generateCode 抽取到一个单独的类中完成，不应该和模型混在一起，不过由于目前代码生成逻辑比较简单，所以暂时将代码生成逻辑放在模型中，以后可以根据需要进行重构。

Component 中我们进行 Id 唯一性验证的时候使用了 ComponentIdManager，Component- IdManager 是用来进行唯一组件 Id 生成和组件 Id 唯一性校验的管理器，其实现如下。

【代码 6-4】组件 Id 管理器：

```
package com.cownew.uidesigner.common;
import Java.util.List;
import com.cownew.ctk.common.NumberUtils;
import com.cownew.uidesigner.model.Component;
import com.cownew.uidesigner.model.Form;

/**
 * 组件 Id 管理器
 * @author 杨中科
 */
public class ComponentIdManager
{
    private Form form;

    public ComponentIdManager(Form form)
    {
        super();
        this.form = form;
    }

    /**
     * 用来生成此 UI 中唯一的组件 id
     * @param component
     * @return
     */
    public String generateId(Component component)
    {
        String modelName = component.getClass()
            .getSimpleName().toLowerCase();
        List<Component> components = form.getComponents();

        // 同类组件的最大序号
        int maxSeqNum = 0;
        for (Component comp : components)
        {
            String compId = comp.getId();
            if (compId.startsWith(modelName))
            {
                String tail = compId.substring(modelName.length(), compId
                    .length());

                if (NumberUtils.isInteger(tail))
                {
                    int intTail = Integer.parseInt(tail);
                    if (intTail > maxSeqNum)

```

```

        {
            maxSeqNum = intTail;
        }
    }
}

return modelName + (maxSeqNum + 1);
}

/**
 * id 是否重复
 * @param id
 * @param component
 * @return
 */
public boolean isIdConflicted(String id, Component component)
{
    List<Component> components = form.getComponents();
    for (Component comp : components)
    {
        String compId = comp.getId();
        if (compId.equals(id) && component != comp)
        {
            return true;
        }
    }
    return false;
}
}

```

ComponentIdManager 接受一个窗体的对象作为参数，这样这个管理器就成为这个窗体的组件 Id 管理器了；generateId 方法用来创建组件 component 的唯一 Id，在 generateId 方法中遍历窗体中的每一个组件，并计算出所有组件中序列号最大的序列号，然后将这个序列号作为这个组件的 Id；isIdConflicted 方法用来判断组件 component 的 Id 是否重复，在 isIdConflicted 方法中遍历窗体中的每一个组件，如果一个不是 component 的组件的 Id 也等于 id 的话则说明 id 重复了。

6.8 代码生成和构建器

上一节我们实现了界面编辑器，但是如果只有界面文件的话这个界面设计工具是没有任何意义的，我们必须提供由界面文件生成 Java 代码的功能，为了能够及时地将用户绘制的界面转换为 Java 代码，我们可以使用构建器来保证在界面文件保存的时候立即能够生成 Java 代码。

6.8.1 代码生成

与前面的章节类似，这里的代码生成同样使用 JET 来完成，下面即是生成 Java 代码的 JET 代码。

【代码 6-27】 代码生成 JET 代码：

```

<%@ jet package="com.cownew.uidesigner.builder"
   imports = "Java.util.* com.cownew.uidesigner.model.*"
   class="JavaCodeGenerator" %>

<%
    ArgInfo argInfo = (ArgInfo)argument;
    Form form = argInfo.getForm();
    String className = argInfo.getClassName();

```

```

String packageName = argInfo.getPackageName();
%>
package <%=packageName%>

import Javax.swing.*;
import Java.awt.*;

public class <%=className%> extends JFrame
{
    private static final long serialVersionUID = 1L;

    private JPanel jContentPane = null;

    public static void main(String[] args)
    {
        <%=className%> frame = new <%=className%>();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public <%=className%>()
    {
        super();
        initialize();
    }

    private void initialize()
    {
        this.setSize(800, 600);
        this.setContentPane(getJContentPane());
        this.setTitle("JFrame");
    }

    private JPanel getJContentPane()
    {
        if (jContentPane != null)
        {
            return jContentPane;
        }
        jContentPane = new JPanel();
        jContentPane.setLayout(null);
    <%
        List<Component> components = form.getComponents();
        for(Component component:components)
        {
            List<String> srcList = component.generateCode("jContentPane");
            for(String src:srcList)
            {
    %>
        <%=src%>

    <%
            }
        }
    >
    return jContentPane;
}

```

```
 }
}
```

这段 JET 代码是非常简单的，稍微有 Swing 基础的人都能看懂，这里不再做过多解释。需要注意的就是传递给 JET 代码参数的 ArgInfo 类是定义的一个 JavaBean，通过这个 JavaBean 可以将要生成代码的 Form 对象、生成的类名和生成的包名等信息传递给 JET。

【代码 6-28】 代码生成信息类：

```
package com.cownew.uidesigner.builder;
import com.cownew.uidesigner.model.Form;
public class ArgInfo
{
    private Form form;
    private String className;
    private String packageName;
    protected String getPackageName()
    {
        return packageName;
    }
    protected void setPackageName(String packageName)
    {
        this.packageName = packageName;
    }
    protected String getClassName()
    {
        return className;
    }
    protected void setClassName(String className)
    {
        this.className = className;
    }
    protected Form getForm()
    {
        return form;
    }
    protected void setForm(Form form)
    {
        this.form = form;
    }
}
```

6.8.2 构建器

构建器又叫增量式项目构建器，只要相关项目中的资源发生改变，构建器就会自动执行。比如，当创建或者修改 Java 源代码文件的时候，Java 构建器就会构建这个 Java 源代码文件生成类文件。如果批处理这些修改的话，构建器将会收到包含所有发生变化的资源列表的唯一一条通知消息，而不是针对每一个变化的资源均收到一条通知消息。需要注意的就是构建器必须增量地执行，也就是只有那些发生改变的派生资源进行构建，如果每次改变一个资源都要对所有资源进行构建的话就会非常占用资源。

Eclipse 中的构建器一般从 IncrementalProjectBuilder 派生，IncrementalProjectBuilder 类中定义了一个抽象方法 build，这个方法的签名如下：

```
protected IProject[] build(int kind, Map args, IProgressMonitor monitor)
```

参数含义如下：

- | kind——表示构建类型，有以下几个有效值：FULL_BUILD、INCREMENTAL_BUILD 和 AUTO_BUILD。FULL_BUILD 指示构建器应当重新构建所有的资源；INCREMENTAL_BUILD 指示

构建器只重新构建有更新的资源；AUTO_BUILD 和 INCREMENTAL_BUILD 一样，不过构建过程由增量式构建自动触发(自动构建选项要打开)。

- | args——指定构建器参数的一个映射，这些参数以参数名为 key，这个参数也可以为空，表示空映射。
- | monitor——进度监视器。

除了 build 方法之外，IncrementalProjectBuilder 类中还定义了下面几个重要方法：

- | forgetLastBuildState()——请求构建器忘记以前构建的时候可能缓存的状态。如果构建过程被中断或者取消，则需要子类调用此方法以防止下次构建出错。
- | getDelta()——返回上次运行构建器以后资源的更改情况，如果没有任何更改则返回 null。
- | getProject()——返回当前被构建的项目。
- | isInterrupted()——返回是否对该构建过程发出了中断请求。

界面文件构建器 UIBuilder 同样从 IncrementalProjectBuilder 派生出来，第一个要实现的方法就是 build。

【代码 6-29】界面文件构建器：

```
public class UIBuilder extends IncrementalProjectBuilder
{
    // BuilderId 比较奇怪，它是由“插件 id” + “.” + “构建器 id” 组成的
    public static final String BUILDER_ID =
        "com.cownew.uidesigner.UIBuilder";

    protected IProject[] build(int kind, Map args, IProgressMonitor monitor)
        throws CoreException
    {
        if (kind == FULL_BUILD)
        {
            fullBuild(monitor);
        } else
        {
            IResourceDelta delta = getDelta(getProject());
            if (delta == null)
            {
                fullBuild(monitor);
            } else
            {
                incrementalBuild(delta, monitor);
            }
        }
        return null;
    }

    private void fullBuild(final IProgressMonitor monitor) throws
        CoreException
    {
        getProject().accept(new UIFullBuildVisitor(monitor));
    }

    private void incrementalBuild(IResourceDelta delta,
        IProgressMonitor monitor) throws CoreException
    {
        delta.accept(new UIDeltaVisitor(monitor));
    }
}
```

UIBuilder 的核心方法就是 build，在 build 中根据 kind 的不同取值来调用 fullBuild 或者 incrementalBuild 进行全面构建或者增量构建。当进行全面构建的时候，我们在项目上调用 accept 方法，使用自定义的 UIFullBuildVisitor 遍历构建所有资源；当进行增加构建的时候我们在增量资源上调用 accept 方法，使用自定义的 UIDeltaVisitor 方法。

这里的 UIFullBuildVisitor 和 UIDeltaVisitor 都使用的是设计模式中最常用的访问者模式。首先来看较简单的 UIFullBuildVisitor。

【代码 6-30】全构建访问者：

```
class UIFullBuildVisitor implements IResourceVisitor
{
    private IProgressMonitor monitor;

    public UIFullBuildVisitor(IProgressMonitor monitor)
    {
        this.monitor = monitor;
    }

    public boolean visit(IResource resource)
    {
        if (!(resource instanceof IFile))
        {
            return true;
        }

        String ext = resource.getFileExtension();
        if (!ext.equalsIgnoreCase("ui"))
        {
            return true;
        }
        BuilderUtils.buildUI((IFile) resource, monitor);
        return true;
    }
}
```

此类的核心就是 visit 方法，在 visit 方法中判断以后被构建的文件以.ui 为扩展名才进行构建，构建的具体过程委托给自定义的 BuilderUtils 工具类的 buildUI 方法来完成。

UIDeltaVisitor 的实现与 UIFullBuildVisitor 比起来略显复杂。

【代码 6-31】增量构建访问者：

```
class UIDeltaVisitor implements IResourceDeltaVisitor
{
    private IProgressMonitor monitor;

    public UIDeltaVisitor(IProgressMonitor monitor)
    {
        this.monitor = monitor;
    }

    public boolean visit(IResourceDelta delta) throws CoreException
    {
        IResource resource = delta.getResource();

        if (!(resource instanceof IFile))
        {
            // 返回值表示是否继续遍历
            return true;
        }

        String ext = resource.getFileExtension();
        if (!ext.equalsIgnoreCase("ui"))
        {
            return true;
        }
        BuilderUtils.buildUI((IFile) resource, monitor);
        return true;
    }
}
```

```

        }

        IFile file = (IFile) resource;
        switch (delta.getKind())
        {
            // 增加了一个 AUI 文件
            case IResourceDelta.ADDED:
                BuilderUtils.buildUI(file, monitor);
                break;
            // AUI 文件被删除
            case IResourceDelta.REMOVED:
                BuilderUtils.deleteJava(file, monitor);
                break;
            // AUI 文件被编辑
            case IResourceDelta.CHANGED:
                BuilderUtils.buildUI(file, monitor);
                break;
        }
        return true;
    }
}

```

在 visit 方法中调用 delta 的 getKind() 方法来判断更改的类型是什么, 如果更改的类型是文件增加(ADDED)或者文件修改(CHANGED)的话则调用工具类 BuilderUtils 的 buildUI 方法重新构建界面文件; 如果更改的类型是文件被删除(REMOVED)则调用工具类 BuilderUtils 的 deleteJava 方法删除界面文件生成的 Java 源代码。

构建器工具类 BuilderUtils 的实现非常清晰明了, 可以直接阅读代码进行研究, 这里不做过多的解释。

【代码 6-32】构建器工具类:

```

package com.cownew.uidesigner.builder;

import Java.io.IOException;
import Java.io.InputStream;
import org.eclipse.core.resources.IFile;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.Path;
import org.eclipse.jdt.core.IJavaElement;
import org.eclipse.jdt.core.IPackageFragment;
import org.eclipse.jdt.core.JavaCore;
import com.cownew.ctk.io.ResourceUtils;
import com.cownew.uidesigner.Activator;
import com.cownew.uidesigner.common.CodeGenUtils;
import com.cownew.uidesigner.model.Form;

public class BuilderUtils
{
    /**
     * 构建界面文件 file
     *
     * @param file
     * @param monitor
     */
    public static void buildUI(IFile file, IProgressMonitor monitor)
    {
        IJavaElement JavaElement = JavaCore.create(file.getParent());
        // 当第二次构建的时候 bin 目录中的 ui 也会被构建一次
        // 这样就造成 JavaElement 为 null 了, 所以需要判断一下
    }
}

```

```
if ((JavaElement instanceof IPackageFragment) == false)
{
    return;
}

IPackageFragment pckFragment = (IPackageFragment) JavaElement;

// 得到短文件名
String fileName = file.getName();
int extLen = file.getFileExtension().length() + 1;
// 去掉扩展名就得到文件对应的类名
String className = fileName.substring(0, fileName.length() - extLen);

// 构建参数
ArgInfo argInfo = new ArgInfo();
argInfo.setClassName(className);
argInfo.setPackageName(pckFragment.getElementName());

InputStream instream = null;
try
{
    instream = file.getContents();
    // 得到界面文件的模型对象
    Form form = Form.makeFromStream(instream);

    argInfo.setForm(form);
    JavaCodeGenerator codeGen = new JavaCodeGenerator();
    // 生成界面文件对应的代码
    String code = codeGen.generate(argInfo);

    // 得到界面文件对应的 Java 源代码文件名
    IFile JavaPath = uiFileToJavaFile(file);
    // 将代码保存到 Java 源代码文件
    CodeGenUtils.saveToFile(JavaPath, code, monitor);
} catch (CoreException e)
{
    Activator.logException(e);
} catch (IOException e)
{
    Activator.logException(e);
} catch (ClassNotFoundException e)
{
    Activator.logException(e);
} finally
{
    ResourceUtils.close(instream);
}
}

private static IFile uiFileToJavaFile(IFile file)
{
    String uiPathName = file.getName();
    // 将 ui 后缀替换为 Java 后缀就得到源码文件名了
    // 有点不严谨，有待改进
    String JavaName = uiPathName.replace(".ui", ".Java");
    IFile pyPath = file.getParent().getFile(new Path(JavaName));
    return pyPath;
}
```

```

    }

    /**
     * 删除界面文件 file 对应的 Java 源码文件
     *
     * @param file
     * @param monitor
     */
    public static void deleteJava(IFile file, IProgressMonitor monitor)
    {
        // 得到界面文件对应的 Java 文件
        IFile JavaPath = uiFileToJavaFile(file);
        try
        {
            // 删除对应的 Java 文件
            JavaPath.delete(true, monitor);
        } catch (CoreException e)
        {
            Activator.logException(e);
        }
    }
}

```

这样界面文件构建器就完成了，最后将构建器添加到 plugin.xml 中即可：

```

<extension
    id="UIBuilder"
    name="UI 构建器"
    point="org.eclipse.core.resources.builders">
    <builder hasNature="false">
        <run class="com.cownew.uidesigner.builder.UIBuilder"/>
    </builder>
</extension>

```

6.8.3 为项目增加构建器

构建器必须被安装到具体的项目中才能发挥作用。最常见的为项目增加构建器的方式是自定义一个项目新建向导，在这个向导中为新建的项目增加构建器，但是这样做对于我们的界面设计工具来说就过于复杂了。为项目增加构建器其实只要修改项目的项目描述即可，因此我们准备为系统增加一个右键菜单项【添加 UI 构建器】，当用户选择某个项目的时候这个菜单项就显示出来，用户单击这个菜单项就可以轻松地为已有的项目增加构建器。

Eclipse 中的菜单项一般都对应一个 Action 类，这里的菜单项就是 AddBuilderAction，它实现了 IObjectActionDelegate 接口。

【代码 6-33】增加构建器的 Action:

```

package com.cownew.uidesigner.builder;

import Java.util.Iterator;
import org.eclipse.core.resources.ICommand;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IProjectDescription;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IAdaptable;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;

```

```
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;
import com.cownew.uidesigner.Activator;
public class AddBuilderAction implements IObjectActionDelegate
{
    private ISelection selection;

    public void run(IAction action)
    {
        if (selection instanceof IStructuredSelection)
        {
            for (Iterator it = ((IStructuredSelection) selection)
                .iterator(); it.hasNext();)
            {
                Object element = it.next();
                IProject project = null;
                if (element instanceof IProject)
                {
                    project = (IProject) element;
                } else if (element instanceof IAdaptable)
                {
                    project = (IProject) ((IAdaptable) element)
                        .getAdapter(IProject.class);
                }
                if (project != null)
                {
                    addUIBuilder(project);
                }
            }
        }
    }

    public void selectionChanged(IAction action, ISelection selection)
    {
        this.selection = selection;
    }

    public void setActivePart(IAction action, IWorkbenchPart targetPart)
    {
    }

    private void addUIBuilder(IProject project)
    {
        try
        {
            //得到项目描述
            IProjectDescription desc = project.getDescription();
            //得到项目所有的构建器
            ICommand[] commands = desc.getBuildSpec();

            for (int i = 0; i < commands.length; ++i)
            {
                if (commands[i].getBuilderName()
                    .equals(UIBuilder.BUILDER_ID))
                {

```

```
        return;
    }
}

 ICommand[] newCommands = new ICommand[commands.length + 1];
System.arraycopy(commands, 0, newCommands, 0, commands.length);
//新建一个构建器
 ICommand command = desc.newCommand();
command.setBuilderName(UIBuilder.BUILDER_ID);
newCommands[newCommands.length - 1] = command;
//将新设置的构建器添加进去
desc.setBuildSpec(newCommands);
project.setDescription(desc, null);
} catch (CoreException e)
{
    Activator.logException(e);
}
}
```

可以看到代码的核心方法就是 `addUIBuilder`，在 `addUIBuilder` 中首先从项目的项目描述中得到所有定义好的构建器，然后调用 `IProjectDescription` 的 `newCommand` 方法得到一个新的构建器，并把构建器设置为这个新构建器的 `Id`，最后将设置后的构建器重新添加到项目描述中。需要特别注意的是这里设置的新构建器的 `Id` 的组成方式是“插件 `id`” + “.” + “构建器 `id`”，也就是“`com.cownew.uidesigner.UIManager`”，而非构建器的 `ID` “`UIBuilder`”，这一点是值得特别注意的。

最后需要将 `AddBuilderAction` 添加到 `plugin.xml` 中去：

```
<extension
  point="org.eclipse.ui.popupMenus">
<objectContribution
  adaptable="true"
  id="com.cownew.uidesigner.contributionUIDesignerNature"
  nameFilter="*"
  objectClass="org.eclipse.core.resources.IProject">
<action
  class="com.cownew.uidesigner.builder.AddBuilderAction"
  enablesFor="+"
  id="com.cownew.uidesigner.builder.AddBuilderAction"
  label="添加 UI 构建器"
  menubarPath="additions" />
</objectContribution>
</extension>
```

6.9 实现常用组件

前面实现了界面设计工具的所有基础功能，但是没有提供常用的组件，比如按钮、标签和编辑框的，因此本节来实现这些组件，这样界面设计工具就能成为实用的工具了。为了减少不必要的篇幅，这里只实现标签、按钮、复选按钮、编辑按钮和列表框等5种组件，可以根据需要增加其他的组件。

6.9.1 标签组件

标签是程序中最常用的组件之一，标签组件的一个基本特征就是能够设定标签上的文字，为了简化实现，这里只为标签组件设定一个“标签文字”的属性。实现一个组件需要实现模型、视图(Figure)、控制器(Part)3个类，首先来看一下标签组件的模型类。

【代码 6-34】 标签模型：

```
package com.cownew.uidesigner.components.label;

import Java.util.ArrayList;
import Java.util.List;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;
import com.cownew.ctk.common.StringUtils;
import com.cownew.uidesigner.commonCodeGenUtils;
import com.cownew.uidesigner.common.ComponentUtils;
import com.cownew.uidesigner.model.Component;
public class Label extends Component
{
    private static final long serialVersionUID = 1L;

    public static final String TEXT = "TEXT";

    private static IPropertyDescriptor[] descriptors =
        new IPropertyDescriptor[]{
            new TextPropertyDescriptor(TEXT, "Text") };

    private String text;

    public Label()
    {
        super();
        setText("Label");
    }

    public void setText(String text)
    {
        String oldValue = this.text;
        this.text = text;
        //通知视图改变
        this.firePropertyChange(TEXT, oldValue, text);
    }

    protected String getText()
    {
        return text;
    }
}
```

```

public IPropertyDescriptor[] getPropertyDescriptors()
{
    IPropertyDescriptor[] sd = super.getPropertyDescriptors();
    return ComponentUtils.mergePropDesc(sd, descriptors);
}

public Object getPropertyValue(Object id)
{
    Object v = super.getPropertyValue(id);
    if (v != null)
    {
        return v;
    }
    if (id.equals(TEXT))
    {
        return getText();
    }
    return null;
}

public void setPropertyValue(Object id, Object value)
{
    super.setPropertyValue(id, value);
    if (id.equals(TEXT))
    {
        setText((String) value);
    }
}

public List<String> generateCode(String parentId)
{
    List<String> list = new ArrayList<String>();

    StringBuffer line1 = new StringBuffer();
    line1.append("JLabel ").append(getId()).append(" = new JLabel()");
    .append(StringUtils.doubleQuoted(getText())).append(";");
    list.add(line1.toString());

    StringBuffer line2 = new StringBuffer();
    line2.append(getId()).append(".setBounds(").append(
        CodeGenUtils.translateBounds(getBounds())).append(");");
    list.add(line2.toString());

    StringBuffer line3 = new StringBuffer();
    line3.append(parentId).append(".add(").append(
        getId()).append(");");
    list.add(line3.toString());

    return list;
}
}

```

模型类 Label 提供了设置和读取标签文字的方法 setText、getText， 并且覆盖了 getPropertyDescriptors、get PropertyValue 和 set PropertyValue 等方法以提供对 Text 属性的属性视图支持。由于父类 Component 已经提供了对 Id、Bounds 等属性的属性视图支持，所以在 getPropertyDescriptors 方法中首先取得父类中的属性描述器，然后调用 ComponentUtils 工具类的 mergePropDesc 方法来将父类的属性描述器与 Text 属性的属性描述器融合。

在 generateCode 方法中实现了代码的生成，生成的代码有 3 行，第一行是调用 JLabel 的构造函数生成标签组件实例，第二行调用 setBounds 方法为组件设定大小和位置，最后一行是将标签组件添加到父组件中去。

接着来看标签视图类的实现。

【代码 6-35】 标签视图：

```
package com.cownew.uidesigner.components.label;

import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.PositionConstants;
public class LabelFigure extends Figure
{
    private Label label;
    private String text;

    public LabelFigure()
    {
        super();
        label = new Label();
        label.setAlignment(PositionConstants.LEFT);
        add(label);
    }

    public void setText(String text)
    {
        this.text = text;
        this.repaint();
    }

    protected void paintFigure(Graphics gc)
    {
        super.paintFigure(gc);
        label.setBounds(getBounds());
        label.setText(text);
    }
}
```

Draw2d 内置的 Label 提供了模拟标签组件的功能，因此我们将 Label 作为我们标签组件的显示组件。在构造函数中，首先创建 Label 类的实例，然后设定 Label 为左对齐，最后调用 add 方法将 Label 添加到视图中去；setText 方法用来修改视图中的标签文字，当文字被修改以后就调用 repaint 方法进行界面重绘；当界面第一次绘制或者被要求重绘的时候 paintFigure 方法就会被调用，在这个方法中设定 label 的大小、位置和标签文字。

最后需要实现的就是组件的控制器。

【代码 6-36】 标签控制器：

```
package com.cownew.uidesigner.components.label;

import org.eclipse.draw2d.IFigure;
import com.cownew.uidesigner.model.Component;
import com.cownew.uidesigner.parts.ComponentPart;
public class LabelPart extends ComponentPart
{
    protected IFigure createFigure()
    {
        return new LabelFigure();
    }
}
```

```

protected void doRefreshFigure(IFigure figure, Component component)
{
    LabelFigure lf = (LabelFigure)figure;
    Label label = (Label)component;
    lf.setText(label.getText());
}
}

```

LabelPart 的实现相对来说比较简单，首先实现 createFigure 方法以供生成控制器对应的视图，接着实现 doRefreshFigure 方法用来根据模型刷新视图。

6.9.2 按钮组件

按钮也是程序中经常用到的组件之一，下面就来实现按钮组件。熟悉 Swing 的朋友都知道，Swing 中的按钮、单选按钮、复选按钮等组件都从一个抽象类 AbstractButton 中派生而来，为了抽象出这些组件的共同特征，我们也同样抽象出一 AbstractButton 模型类出来，这样按钮、单选框、复选框等组件就可以从 AbstractButton 派生并很容易地实现各自的模型类了。

【代码 6-37】 抽象按钮模型：

```

package com.cownew.uidesigner.model;

import org.eclipse.ui.views.properties.IPropertyDescriptor;
import org.eclipse.ui.views.properties.TextPropertyDescriptor;
import com.cownew.uidesigner.common.ComponentUtils;
public abstract class AbstractButton extends Component
{
    public static final String TEXT = "TEXT";

    private static IPropertyDescriptor[] descriptors = new
        IPropertyDescriptor[] { new TextPropertyDescriptor(
            TEXT, "Text" ) };

    private String text;

    public AbstractButton()
    {
        super();
        text = "";
    }

    public String getText()
    {
        return text;
    }

    public void setText(String text)
    {
        String oldValue = getText();
        this.text = text;
        this.firePropertyChange(TEXT, oldValue, text);
    }

    public IPropertyDescriptor[] getPropertyDescriptors()
    {
        IPropertyDescriptor[] sd = super.getPropertyDescriptors();
        return ComponentUtils.mergePropDesc(sd, descriptors);
    }
}

```

```

public Object getPropertyValue(Object id)
{
    Object v = super.getPropertyValue(id);
    if (v != null)
    {
        return v;
    }
    if (id.equals(TEXT))
    {
        return getText();
    }
    return null;
}

public void setPropertyValue(Object id, Object value)
{
    super.setPropertyValue(id, value);
    if (id.equals(TEXT))
    {
        setText((String) value);
    }
}
}

```

可以看到 AbstractButton 中提供了对 Text 这个属性的设置、读取和属性视图的支持，这样子类组件模型只要实现各自特有的逻辑即可。

有了 AbstractButton 以后按钮组件的模型类就很容易实现了。

【代码 6-38】 按钮模型：

```

package com.cownew.uidesigner.components.button;

import Java.util.ArrayList;
import Java.util.List;
import com.cownew.ck.common.StringUtils;
import com.cownew.uidesigner.commonCodeGenUtils;
import com.cownew.uidesigner.model.AbstractButton;
public class Button extends AbstractButton
{
    private static final long serialVersionUID = 1L;

    public List<String> generateCode(String parentId)
    {
        List<String> list = new ArrayList<String>();
        StringBuffer line1 = new StringBuffer();
        line1.append("JButton ").append(getId()).append("= new JButton(")
            .append(StringUtils.doubleQuoted(getText())).append(");");
        list.add(line1.toString());

        StringBuffer line2 = new StringBuffer();
        line2.append(getId()).append(".setBounds(").append(
            CodeGenUtils.translateBounds(getBounds())).append(");");
        list.add(line2.toString());

        StringBuffer line3 = new StringBuffer();
        line3.append(parentId).append(".add(").append(getId()).append(");");
        list.add(line3.toString());
    }
}

```

```
        return list;
    }
}
```

由于 `AbstractButton` 中已经提供了对 `Text` 属性的支持，所以 `Button` 类中只需要提供代码生成的逻辑，这里代码生成的逻辑实现和 `Label` 是非常相似的，这里就不做过多的介绍。

下面来看一下按钮视图的实现。

【代码 6-39】 按钮视图：

```
package com.cownew.uidesigner.components.button;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.SchemeBorder;
import org.eclipse.swt.graphics.Color;
public class ButtonFigure extends Figure
{
    private Label label;

    private String text;

    protected static final SchemeBorder.Scheme SCHEME_FRAME =
        new SchemeBorder.Scheme(
            new Color[] {ColorConstants.button,
                        ColorConstants.buttonLightest,
                        ColorConstants.button },
            new Color[] {ColorConstants.buttonDarkest,
                        ColorConstants.buttonDarker,
                        ColorConstants.button });

    public ButtonFigure()
    {
        super();
        label = new Label();
        label.setBorder(new SchemeBorder(SCHEME_FRAME));
        add(label);
    }

    public void setText(String text)
    {
        this.text = text;
        this.repaint();
    }

    protected void paintFigure(Graphics g)
    {
        super.paintFigure(g);
        label.setBounds(getBounds());
        label.setText(text);
    }
}
```

Draw2d 中的 `Button` 是从 `Clickable` 派生出来的，也就是这个 `Button` 是可以响应鼠标事件的，但是我们却希望视图中的按钮是只可以具有按钮的外观而不可以有按钮的行为的。经过研究 `Button` 的代码我们发现，Draw2d 的 `Button` 的视图部分同样是使用 `Label` 来实现的，只是为 `Label` 增加了一个边框而已，这样 `Label` 就有了按钮的外观。由此 `ButtonFigure` 中我们同样使用 `Label` 来作为视图组件，唯一的区别就是将 `Label`

添加了 SchemeBorder 类型的边框。

最后需要实现的就是组件的控制器。

【代码 6-40】 按钮控制器:

```
package com.cownew.uidesigner.components.button;

import org.eclipse.draw2d.IFigure;
import com.cownew.uidesigner.model.Component;
import com.cownew.uidesigner.parts.ComponentPart;
public class ButtonPart extends ComponentPart
{
    protected IFigure createFigure()
    {
        return new ButtonFigure();
    }

    protected void doRefreshFigure(IFigure figure, Component component)
    {
        ButtonFigure bf = (ButtonFigure) figure;
        Button btn = (Button) component;
        bf.setText(btn.getText());
    }
}
```

ButtonPart 的实现和标签控制器的实现非常类似。首先实现 createFigure 方法以供生成控制器对应的视图，接着实现 doRefreshFigure 方法用来根据模型刷新视图。

6.9.3 复选框

Swing 中的 JCheckBox、JRadioButton 等组件都从 JToggleButton 派生，为了抽象出这些组件的共同特征，我们也同样抽象出一 ToggleButton 模型类出来，这样单选按钮、复选按钮等组件的模型就可以从 ToggleButton 派生并很容易的实现各自的模型类了。

【代码 6-41】 开关按钮模型:

```
package com.cownew.uidesigner.model;

import org.eclipse.ui.views.properties.IPropertyDescriptor;
import com.cownew.Eclipse.properties.BooleanPropertyDescriptor;
import com.cownew.uidesigner.common.ComponentUtils;
public abstract class ToggleButton extends AbstractButton
{

    private static final long serialVersionUID = 1L;

    public static final String SELECTED = "Selected";

    private static IPropertyDescriptor[] descriptors =
        new IPropertyDescriptor[] {
            new BooleanPropertyDescriptor(SELECTED, "Selected") };

    private boolean selected;

    public ToggleButton()
    {
        super();
        selected = true;
    }
```

```

public boolean isSelected()
{
    return selected;
}

public void setSelected(boolean selected)
{
    Boolean oldValue = Boolean.valueOf(isSelected());
    this.selected = selected;
    this.firePropertyChange(SELECTED, oldValue,
                           Boolean.valueOf(selected));
}

public IPropertyDescriptor[] getPropertyDescriptors()
{
    IPropertyDescriptor[] sd = super.getPropertyDescriptors();
    return ComponentUtils.mergePropDesc(sd, descriptors);
}

public Object getPropertyValue(Object id)
{
    Object v = super.getPropertyValue(id);
    if (v != null)
    {
        return v;
    }
    if (id.equals(SELECTED))
    {
        return Boolean.valueOf(isSelected());
    }
    return null;
}

public void setPropertyValue(Object id, Object value)
{
    super.setPropertyValue(id, value);
    if (id.equals(SELECTED))
    {
        Boolean b = (Boolean) value;
        setSelected(b.booleanValue());
    }
}
}

```

可以看到 ToggleButton 中提供了对 Selected 这个属性的设置、读取和属性视图的支持，这样子类组件模型只要实现各自特有的逻辑即可。

有了 ToggleButton 以后复选按钮组件的模型类就很容易实现了。

【代码 6-42】 复选框模型：

```

package com.cownew.uidesigner.components.checkbox;

import Java.util.ArrayList;
import Java.util.List;
import com.cownew.ctl.common.StringUtils;
import com.cownew.uidesigner.commonCodeGenUtils;

```

```

public class CheckBox extends ToggleButton
{
    public List<String> generateCode(String parentId)
    {
        List<String> list = new ArrayList<String>();

        StringBuffer line1 = new StringBuffer();
        line1.append("JCheckBox ").append(getId()).append("= new JCheckBox(")
            .append(StringUtils.doubleQuoted(getText())).append(");");
        list.add(line1.toString());

        StringBuffer line2 = new StringBuffer();
        line2.append(getId()).append(".setBounds(").append(
            CodeGenUtils.translateBounds(getBounds())).append(");");
        list.add(line2.toString());

        StringBuffer line3 = new StringBuffer();
        line3.append(getId()).append(".setSelected(").append(isSelected())
            .append(");");
        list.add(line3.toString());

        StringBuffer line4 = new StringBuffer();
        line4.append(parentId).append(".add(").append(getId()).append(");");
        list.add(line4.toString());

        return list;
    }
}

```

由于 ToggleButton 中已经提供了对 Selected 属性的支持，所以 CheckBox 类中只需要提供代码生成的逻辑，这里代码生成的逻辑实现和 Label 是非常相似的，不同的地方就是提供了对 Selected 属性的代码生成的支持。

下面来看一下复选按钮视图的实现。

【代码 6-43】 复选框视图：

```

package com.cownew.uidesigner.components.checkbox;

import Java.io.InputStream;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.PositionConstants;
import org.eclipse.swt.graphics.Image;
import com.cownew.ctk.io.ResourceUtils;
public class CheckBoxFigure extends Figure
{
    private Label label;
    private boolean selected;
    private String text;

    static final Image UNCHECKED = createImage("off.gif");

    static final Image CHECKED = createImage("on.gif");

    private static Image createImage(String name)
    {
        InputStream stream =

```

```
        (CheckBoxFigure.class).getResourceAsStream(name);
        Image image = new Image(null, stream);
        ResourceUtils.close(stream);
        return image;
    }

    public CheckBoxFigure()
    {
        super();
        label = new Label();
        label.setAlignment(PositionConstants.LEFT);
        add(label);
    }

    public void setText(String text)
    {
        this.text = text;
        this.repaint();
    }

    public void setSelected(boolean selected)
    {
        this.selected = selected;
        this.repaint();
    }

    protected void paintFigure(Graphics g)
    {
        super.paintFigure(g);
        label.setBounds(getBounds());
        label.setIcon(selected ? CHECKED : UNCHECKED);
        label.setText(text);
    }
}
```

Draw2d 的 Label 可以设置一个图标，这个图标将会显示在标签的左边，根据这个特性，可以使用 Label 来实现复选按钮的外观。将选中、非选中两种状态的图标(分别是 on.gif和 off.gif)放在 CheckBoxFigure 包下，这样使用 createImage 方法就可以得到这个图标，在 paintFigure 中根据 selected 属性值来为 Label 设置不同的图标，这样就可以达到复选框的外观效果。

最后需要实现的就是组件的控制器。

【代码 6-44】复选框控制器:

```
package com.cownew.uidesigner.components.checkbox;

import org.eclipse.draw2d.IFigure;
import com.cownew.uidesigner.model.Component;
import com.cownew.uidesigner.parts.ComponentPart;
public class CheckBoxPart extends ComponentPart
{
    protected IFigure createFigure()
    {
        return new CheckBoxFigure();
    }

    protected void doRefreshFigure(IFigure figure, Component component)
    {
        CheckBoxFigure cbf = (CheckBoxFigure) figure;
        CheckBox label = (CheckBox) component;
        cbf.setText(label.getText());
        cbf.setSelected(label.isSelected());
    }
}
```

6.9.4 编辑框

编辑框的实现是比较简单的，首先看编辑框模型类。

【代码 6-45】编辑框模型:

```
package com.cownew.uidesigner.components.edit;

import Java.util.ArrayList;
import Java.util.List;
import com.cownew.uidesigner.commonCodeGenUtils;
import com.cownew.uidesigner.model.Component;
public class Edit extends Component
{
    private static final long serialVersionUID = 1L;

    public List<String> generateCode(String parentId)
    {
        List<String> list = new ArrayList<String>();

        StringBuffer line1 = new StringBuffer();

        line1.append("JTextField ").append(getId())
            .append(" = new JTextField(\"").append(") ;" );
        list.add(line1.toString());

        StringBuffer line2 = new StringBuffer();
        line2.append(getId()).append(".setBounds(").append(
            CodeGenUtils.translateBounds(getBounds())).append(") ;" );
        list.add(line2.toString());

        StringBuffer line3 = new StringBuffer();
        line3.append(parentId).append(".add(\"").append(getId()).append(") ;" );
        list.add(line3.toString());
    }
}
```

```
        return list;
    }
}
```

编辑框视图的实现也同样使用 Label 来实现编辑框的外观。

【代码 6-46】 编辑框视图:

```
package com.cownew.uidesigner.components.edit;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.SchemeBorder;
import org.eclipse.swt.graphics.Color;
public class EditFigure extends Figure
{
    private Label label;

    protected static final SchemeBorder.Scheme SCHEME_FRAME =
        new SchemeBorder.Scheme(
            new Color[] { ColorConstants.lightGreen, ColorConstants.lightGreen,
            ColorConstants.lightGreen });

    public EditFigure()
    {
        super();
        label = new Label();
        label.setBorder(new SchemeBorder(SCHEME_FRAME));
        add(label);
    }

    protected void paintFigure(Graphics g)
    {
        super.paintFigure(g);
        label.setBounds(getBounds());
    }
}
```

最后是编辑框的控制器。

【代码 6-47】 编辑框控制器:

```
package com.cownew.uidesigner.components.edit;

import org.eclipse.draw2d.IFigure;
import com.cownew.uidesigner.model.Component;
import com.cownew.uidesigner.parts.ComponentPart;
public class EditPart extends ComponentPart
{

    protected IFigure createFigure()
    {
        return new EditFigure();
    }

    protected void doRefreshFigure(IFigure figure, Component component)
    {
    }
}
```

```
}
```

6.9.5 列表框

列表框最基本的特征就是允许设置列表框中的数据，因此为列表框组件模型提供 `Items` 属性，列表框组件模型实现如下。

【代码 6-48】列表框模型：

```
package com.cownew.uidesigner.components.listbox;

import Java.util.ArrayList;
import Java.util.List;
import org.eclipse.ui.views.properties.IPropertyDescriptor;
import com.cownew.ctk.common.EnvironmentUtils;
import com.cownew.ctk.common.StringUtils;
import com.cownew.Eclipse.properties.MultiLineStringPropertyDescriptor;
import com.cownew.uidesigner.common.CodeGenUtils;
import com.cownew.uidesigner.common.ComponentUtils;
import com.cownew.uidesigner.model.Component;
public class ListBox extends Component
{
    private static final long serialVersionUID = 1L;

    public static final String ITEMS = "ITEMS";

    private static IPropertyDescriptor[] descriptors =
        new IPropertyDescriptor[] {
            new MultiLineStringPropertyDescriptor(ITEMS, "items") };

    private String items;

    public ListBox()
    {
        super();
        items = "";
    }

    public String getItems()
    {
        return items;
    }

    public void setItems(String items)
    {
        String oldValue = getItems();
        this.items = items;
        this.firePropertyChange(ITEMS, oldValue, items);
    }

    public IPropertyDescriptor[] getPropertyDescriptors()
    {
        IPropertyDescriptor[] sd = super.getPropertyDescriptors();
        return ComponentUtils.mergePropDesc(sd, descriptors);
    }

    public Object getPropertyValue(Object id)
```

```

{
    Object v = super.getPropertyValue(id);
    if (v != null)
    {
        return v;
    }
    if (id.equals(ITEMS))
    {
        return getItems();
    }
    return null;
}

public void setPropertyValue(Object id, Object value)
{
    super.setPropertyValue(id, value);
    if (id.equals(ITEMS))
    {
        setItems((String) value);
    }
}

public List<String> generateCode(String parentId)
{
    List<String> list = new ArrayList<String>();

    StringBuffer listData = new StringBuffer();
    listData.append("new String[]{");
    String[] contents = getItems().split(
        EnvironmentUtils.getLineSeparator());

    for (int i = 0, n = contents.length; i < n; i++)
    {
        listData.append(StringUtils.doubleQuoted(contents[i]));
        if (i < contents.length - 1)
        {
            listData.append(",");
        }
    }
    listData.append("}");

    StringBuffer line1 = new StringBuffer();
    line1.append("JList ").append(getId()).append("= new JList(").append(
        listData).append(");");
    list.add(line1.toString());

    StringBuffer line2 = new StringBuffer();
    line2.append(getId()).append(".setBounds(").append(
        CodeGenUtils.translateBounds(getBounds())).append(");");
    list.add(line2.toString());

    StringBuffer line3 = new StringBuffer();
    line3.append(parentId).append(".").append(getId()).append(");");
    list.add(line3.toString());

    return list;
}

```

```
}
```

模型类中 getPropertyDescriptors、getPropertyValue、setPropertyvalue 以及 generateCode 方法的实现和其他组件的实现是非常类似的，唯一需要注意的就是 Items 的属性描述器，因为 Eclipse 中没有提供 Items 这样多行文本的属性描述器，因此定义了多行文本属性描述器 MultiLineStringPropertyDescriptor。

【代码 6-49】 多行文本属性描述器：

```
package com.cownew.Eclipse.properties;

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.jface.viewers.CellEditor;
import org.eclipse.jface.viewers.DialogCellEditor;
import org.eclipse.jface.viewers.ICellEditorValidator;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.views.properties.PropertyDescriptor;

public class MultiLineStringPropertyDescriptor extends PropertyDescriptor
{

    public MultiLineStringPropertyDescriptor(Object id, String displayName)
    {
        super(id, displayName);
    }

    public CellEditor createPropertyEditor(Composite parent)
    {
        CellEditor editor = new MultiLineStringCellEditor(parent);
        if (getValidator() != null)
        {
            editor.setValidator(getValidator());
        }
        return editor;
    }

    protected ICellEditorValidator getValidator()
    {
        return new ICellEditorValidator() {

            public String isValid(Object value)
            {
                if (value == null)
                {
                    return "Cannot be null";
                }
                return null;
            }
        };
    }
}
```

MultiLineStringCellEditor 是从 DialogCellEditor 派生的多行文本单元格编辑器，用来提供对话框形式的多行文本编辑功能。

【代码 6-50】多行文本单元格编辑器:

```
class MultiLineStringCellEditor extends DialogCellEditor
{
    public MultiLineStringCellEditor(Composite parent)
    {
        super(parent);
    }

    protected Object openDialogBox(Control ctrl)
    {
        String oldValue = (String) getValue();
        if (oldValue == null)
        {
            oldValue = "";
        }
        MultiLineStringDialog dlg =
            new MultiLineStringDialog(ctrl.getShell(), oldValue);
        if (dlg.open() == MultiLineStringDialog.OK)
        {
            return dlg.getValue();
        }
        return oldValue;
    }
}
```

当用户单击编辑器中的浏览按钮的时候, openDialogBox 方法就会被调用, 以弹出对话框, 在 openDialogBox 方法中弹出多行文本对话框 MultiLineStringDialog, 并且将对话框中的编辑值作为返回值填充到编辑器中。多行文本对话框 MultiLineStringDialog 的实现如下。

【代码 6-51】多行文本对话框:

```
class MultiLineStringDialog extends Dialog
{
    private Text text;
    private String value;
    private String initialValue;

    protected MultiLineStringDialog(Shell shell, String initialValue)
    {
        super(shell);
        if (initialValue == null)
        {
            this.initialValue = "";
        } else
        {
            this.initialValue = initialValue;
        }
    }

    protected Control createDialogArea(Composite parent)
    {
        Composite composite = (Composite) super.createDialogArea(parent);

        FillLayout layout = new FillLayout();
        composite.setLayout(layout);
        text = new Text(composite, SWT.MULTI | SWT.V_SCROLL | SWT.H_SCROLL);
        text.setText(initialValue);
```

```

        return composite;
    }

    protected Point getInitialSize()
    {
        return new Point(300, 200);
    }

    public boolean close()
    {
        value = text.getText();
        return super.close();
    }

    public String getValue()
    {
        return value;
    }
}

```

Draw2d 中的 Label 可以支持多行文本，当其中的文本中含有换行符的时候 Label 会自动实现视图中的文本换行，利用这个特性可以非常容易地实现列表框的视图。下面来看一下列表框视图的实现。

【代码 6-52】列表框视图：

```

package com.cownew.uidesigner.components.listbox;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Graphics;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.SchemeBorder;
import org.eclipse.swt.graphics.Color;
public class ListBoxFigure extends Figure
{
    private String items;

    private Label innerList = null;

    protected static final SchemeBorder.Scheme SCHEME_FRAME =
        new SchemeBorder.Scheme(
            new Color[] {ColorConstants.lightGreen,
                        ColorConstants.lightGreen,
                        ColorConstants.lightGreen },
            new Color[] {ColorConstants.lightGreen,
                        ColorConstants.lightGreen,
                        ColorConstants.lightGreen });

    public ListBoxFigure()
    {
        super();
        setBorder(new SchemeBorder(SCHEME_FRAME));
        innerList = new Label();
        innerList.setLabelAlignment(Label.LEFT);
        innerList.setTextAlignment(Label.TOP);
        add(innerList);
    }
}

```

```

    {
        this.items = items;
        this.repaint();
    }

    protected void paintFigure(Graphics graphics)
    {
        super.paintFigure(graphics);
        innerList.setBounds(getBounds());
        //Label 可以放置带换行符的多行文字，所以可以自动实现多行效果
        innerList.setText(items);
    }
}

```

最后需要实现的就是组件的控制器。

【代码 6-53】列表框控制器：

```

package com.cownew.uidesigner.components.listbox;

import org.eclipse.draw2d.IFigure;
import com.cownew.uidesigner.model.Component;
import com.cownew.uidesigner.parts.ComponentPart;
public class ListBoxPart extends ComponentPart
{
    protected IFigure createFigure()
    {
        return new ListBoxFigure();
    }

    protected void doRefreshFigure(IFigure figure, Component component)
    {
        ListBoxFigure bf = (ListBoxFigure)figure;
        ListBox btn = (ListBox)component;
        bf.setItems(btn.getItems());
    }
}

```

6.10 使用演示

至此，一个简单的界面设计工具就完成了，下面来看一看运行效果。

(1) 首先创建一个 Java 项目，并创建包 com.cownew.demo，在包 com.cownew.demo 上右击，在弹出的快捷菜单中选择【新建】|【其他】命令，接着会弹出如图 6.1 所示的对话框(【选择向导】界面)。



图 6.1 【新建】对话框

(2) 在对话框中选择【UI 编辑器】选项，单击【下一步】按钮，进入如图 6.2 所示的界面。



图 6.2 设置文件名

(3) 在【文件名】文本框中填入“MyApp.ui”，单击【完成】按钮，Eclipse 就会打开新建的界面文件，如图 6.3 所示。

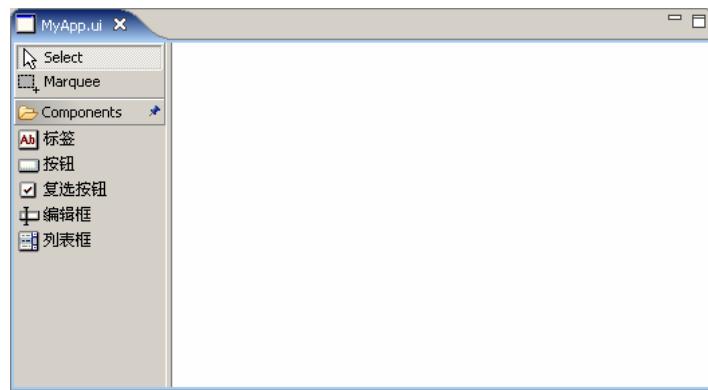


图 6.3 界面设计器

(4) 编辑器左侧是组件面板，从组件面板中拖放需要的控件到图形编辑区，并且修改相应的属性。设计好的界面如图 6.4 所示。

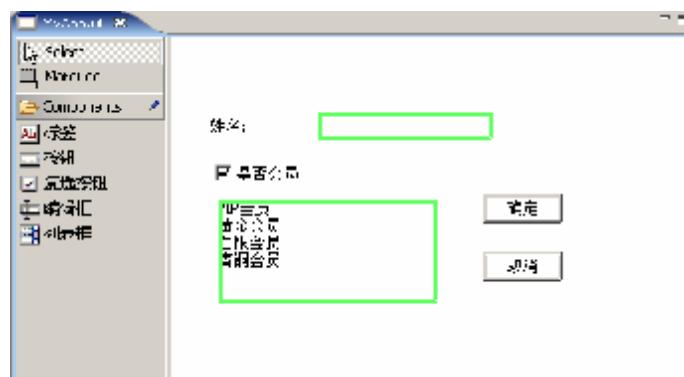


图 6.4 设计好的界面

在界面文件的目录下可以看到自动生成的 MyApp.java 文件，代码如下。

【代码 6-54】 生成的界面代码：

```
package com.cownew.demo;

import javax.swing.*;
import java.awt.*;
public class MyApp extends JFrame
{
    private static final long serialVersionUID = 1L;

    private JPanel jContentPane = null;

    public static void main(String[] args)
    {
        MyApp frame = new MyApp();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public MyApp()
    {
        super("MyApp");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new BorderLayout());
        this.jContentPane = new JPanel();
        this.jContentPane.setLayout(new GridLayout(1, 1));
        this.jContentPane.add(new JTextField("姓名"));
        this.jContentPane.add(new JCheckBox("单行文本"));
        this.jContentPane.add(new JComboBox(new String[] {"IP三只", "龙之女", "林心如", "周润发"}));
        this.jContentPane.add(new JButton("确定"));
        this.jContentPane.add(new JButton("取消"));
        this.setContentPane(this.jContentPane);
    }
}
```

```

        initialize();
    }

    private void initialize()
    {
        this.setSize(800, 600);
        this.setContentPane(getJContentPane());
        this.setTitle("JFrame");
    }

    private JPanel getJContentPane()
    {
        if (jContentPane != null)
        {
            return jContentPane;
        }
        jContentPane = new JPanel();
        jContentPane.setLayout(null);
        JLabel labelName= new JLabel("姓名:");
        labelName.setBounds(new Rectangle(21,27,60,22));
        jContentPane.add(labelName);
        JTextField edtName= new JTextField();
        edtName.setBounds(new Rectangle(93,26,91,22));
        jContentPane.add(edtName);
        JCheckBox checkbox1= new JCheckBox("是否会员");
        checkbox1.setBounds(new Rectangle(23,59,117,22));
        checkbox1.setSelected(true);
        jContentPane.add(checkbox1);
        JList listbox1= new JList(new String[]{"VIP 会员", "黄金会员",
                                              "白银会员", "青铜会员"});
        listbox1.setBounds(new Rectangle(25,83,141,64));
        jContentPane.add(listbox1);
        JButton button1= new JButton("确定");
        button1.setBounds(new Rectangle(191,82,60,22));
        jContentPane.add(button1);
        JButton button2= new JButton("取消");
        button2.setBounds(new Rectangle(192,121,60,22));
        jContentPane.add(button2);
        return jContentPane;
    }
}

```

运行此 Java 文件，效果如图 6.5 所示。



图 6.5 运行效果

