

GEF 开发指南

org.eclipse.gef

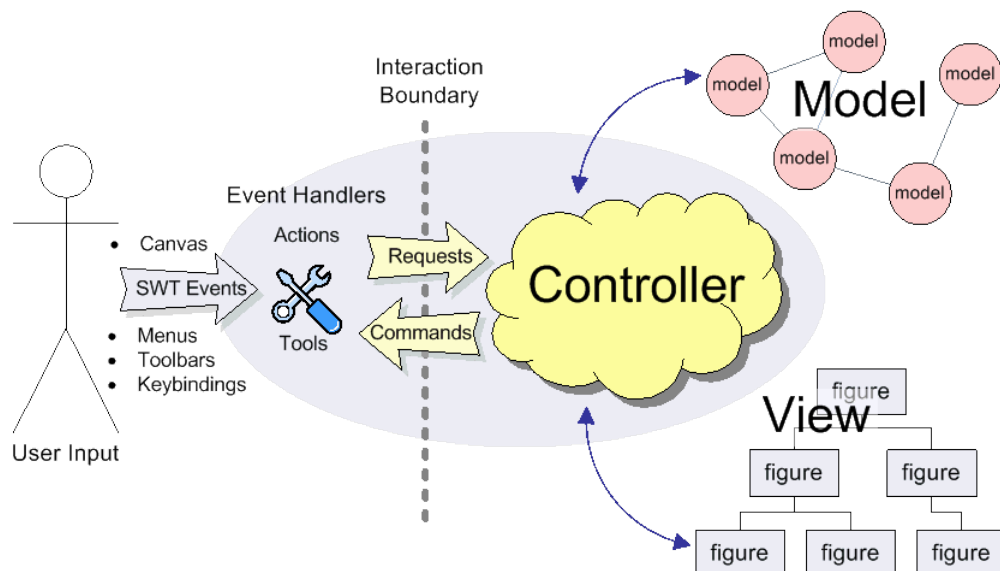
- [概述](#) - "big picture" 的介绍
- [何时使用GEF](#) – GEF和Eclipse平台的使用
- [EditParts](#) – GEF的主要构成部分
- [图形界面](#) – 如何为你的模型创建图形界面
- [编辑和编辑策略](#) – 给图形界面添加编辑操作支持
- [EditPart 生命周期](#) – 了解一些有趣的事件
- [工具和绘图板](#)
- [交互](#) – GEF支持的用户交互和响应

概述

Draw2D 侧重于高效的绘制和布局图形，而 GEF 插件则在 Draw2D 之上添加了编辑操作。它的目的在于：

1. 使用 draw2d 的图形使图形化的显示模型更加便利
2. 支持与鼠标、键盘或者工作台的交互
3. 提供与上面相关的常用的组件

下图显示了 GEF 的高层结构。GEF 可以被大概定义为图中中间的区域，它提供了应用模型与视图之间的桥梁，同时还包括一些能将事件(Event)转换为请求(Request)的工具(Tool)和操作(Action)。请求(Request)和命令(Command)用于进行交互指令封装，它们对模型有效。



在 MVC(M: 模型 V: 视图 C: 控制器)三层架构设计中，控制器(Controller)往往是视图(View)和模型(Model)之间唯一的中介，它负责控制视图显示，处理 UI 事件并将它们作用于模型。在 GEF 中它们扮演的角色如下：

模型(Model)

模型可以是任何持久化的数据,所有的模型都可以用在 GEF 中,但是它们必须有某种通知机制。尽管技术上不要求这样,但是模型和命令(Command)经常密切相关。命令完成对模型的修改,同时它还可以支持用户的重做(Redo)和撤销(Undo)操作。一般来说,命令只作用于模型本身。

视图(View)Figures/Treeltems

视图对用户来说是可视化的,在 GEF 中图形(Figure)和树节点(TreeItem)都可以作为视图元素。

控制器(Controller)EditPart

通常会为每一个可视化的模型对象分配一个控制器。在 GEF 中它们称之为“EditPart”。EditPart 是模型和视图之间的桥梁,它也负责响应用户编辑操作,一种叫做 EditPolicy 的助手将辅助它完成绝大部分的编辑任务。

Viewers

EditPart 在 EditPartViewer 上显示视图。GEF 中有两种视图,一种视图使用 GraphicalViewer 显示模型图形而另一种则以 TreeViewer 来显示树形节点。GEF 的 Viewer 与 JFace 的类似,它们使用 SWT 控件。Viewer 提供选择功能,在 Viewer 中选中的就是 EditPart。

何时使用 GEF?

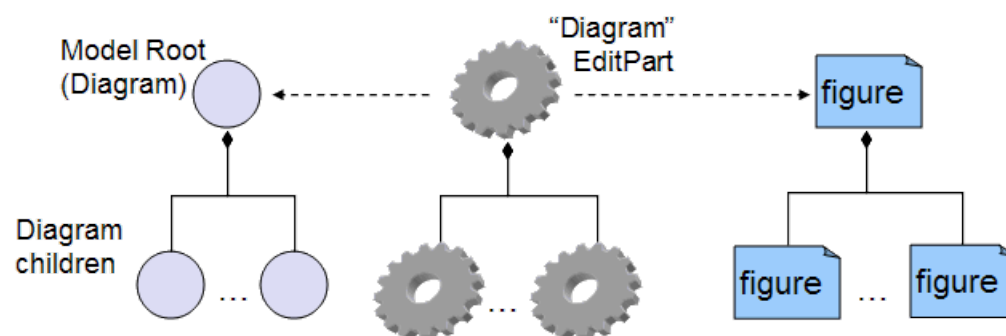
在 Workbench 中,任何可以使用 SWT 控件的地方都可以使用 GEF。它可以用在编辑器、视图、向导中等。最常用的是在 EditorPart 中,有时候也用在编辑器的大纲视图中。

GEF 需要 Eclipse 富客户端平台(RCP)支持,同时需要 org.eclipse.ui.views 插件来提供属性页功能。

EditPart 介绍

EditPart 关联着视图和模型,同时它也有自己的组织结构——父子关系。EditPart 可以有子 EditPart,它们之间的关系与对应模型之间的关系一样。比如某个示例图模型包含一些节点模型,对应的就有示例图 EditPart 就包含一些节点 EditPart,作为它的子 EditPart。

EditPart 之间的父子关系会持续到图形(Figure)中,父图形将包含子图形。在某些情况下,父 EditPart 对应的视图由几个的 Figure 组成,这些 Figure 都是容器面板,每个面板都有自己子 Figure。这样你就会得到一个具有并行结构图形了。



连接(Connection)不同于简单的树形图形结构，它们代表了两个对象之间的关联关系。Draw2D 的连接图形用在 view 中。连接存放在模型中，但是对应的 EditPart 则由源端点和目的端点模型对应的 EditPart 管理。连接的图形也做了特殊处理，它们被放置在一个特殊的 Layer——ConnectionLayer 上，ConnectionLayer 位于主 Layer(PrimaryLayer)之上，以保证连接与端点图形之间的逻辑性正确，以及相互之间不干扰。

GEf 提供 2 种 EditPart 的实现：GraphicalEditPart 和 TreeEditPart。GraphicalEditPart 使用 Figure 作为视图，它同样支持连接。TreeEditPart 使用 SWT 的 TreeItem 作为视图，以显示树形结构的大纲(Outline)。

EditPart 的作用如下：

- 创建和维护视图(Figure 或者 TreeItem)
- 创建和维护子 EditPart
- 创建和维护连接 EditPart
- 支持模型编辑

维护视图和其他的 EditPart 意味着 EditPart 必须知道模型的变更信息以便做出相应的操作，通常的做法是将 EditPart 作为一个监听器，监听关联模型对象。当它收到更新通知时，EditPart 按照变更的内容更新视图或者自身结构。

EditPart 这个名称也意味着它必须支持模型的编辑。这里我们首先关注的是构建一个可视化的显示模型的程序的步骤。

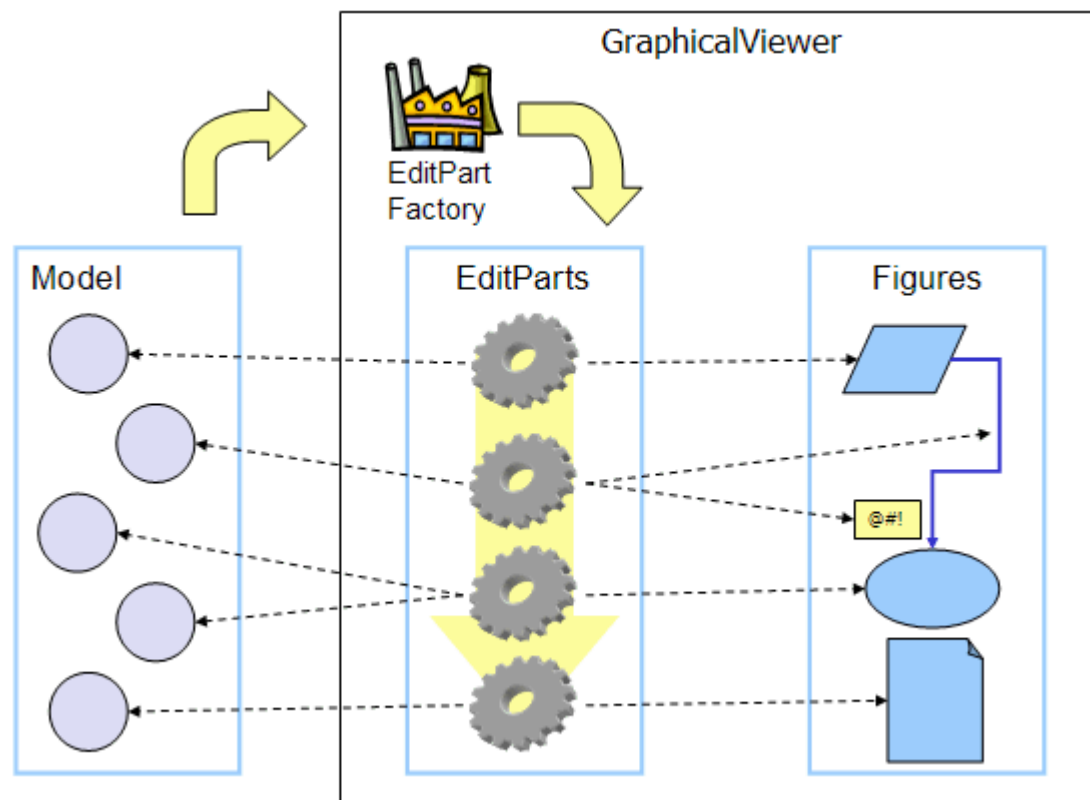
为模型创建图形界面(GraphicalViewer)

一旦你拥有了模型(Model)以及对应的显示图形(Figure)，接下来就是将这些组件组合起来。这意味着我们需要创建一个 EditPart，然后用它关联模型和视图图形。GEF 自带的 EditPart 是抽象类，在你的应用程序中必须要扩展它们。首先我们创建应用程序的基础。

GEF 提供类 ScrollingGraphicalViewer，它使用 Draw2D 的 FigureCanvas。大部分的程序将使用这个类，除非因为我们不需要滚动条。下一步我们将决定根(Root)EditPart。每一个 EditPart Viewer 都需要一个 Root EditPart，它不对应任何已有的模型。Root EditPart 为 Viewer 中的其他 EditPart 提供一个统一的上下文环境。GEF 提供了两种可选的 Root EditPart：

- a) **ScalableRootEditPart** – 提供标准的 Layer 集合，并支持缩放功能。
- b) **ScalableFreeformRootEditPart** – 与 ScalableRootEditPart 相似，区别是它的 Layer 集合中的每一个 Layer 都符合 Freeform 接口，意味着它们可以向负方向扩展，比如向左或向上。它非常灵活，也是最常用的 Root EditPart。

现在 Viewer 和 Root EditPart 已经准备好了，接下来我们需要为 Viewer 提供填充模型(Content)。填充模型指的是 Viewer 显示的模型，一般的我们先初始化它为所有模型的父模型，然后再依次添加子模型。这时候 EditPartFactory 出现了，它负责将模型和对应的 EditPart 联系起来。EditPartFactory 需要在 Root EditPart 里设置。当模型创建以后，EditPartFactory 负责为其创建对应的 EditPart。EditPart 的父子关系在这个时候也就产生了，随着 EditPartFactory 的不断被调用，所有的 EditPart 将被创建完毕。



实现填充模型(Content)的 EditPart

下面我们尝试实现填充模型的 **EditPart**, 同时记得要在 **EditPartFactory** 里将模型和它的 **EditPart** 对应起来。**EditPart** 的 **Figure** 作为其他图形的背景, 通常情况下它甚至不用绘制, 但记得给它指定一个布局管理器(**LayoutManager**)。Figure 的初始化在 **EditPart** 的 **createFigure()** 方法中, 你需要重写这个方法。如果使用 **FreeForm** 格式的 **Root EditPart**, 那么 **Figure** 就可以简单的指定为 **FreeformLayer**, 同时布局管理器也相应位 **FreeformLayout**。下面是一个可选的 **createFigure()**:

```
protected IFigure createFigure() {
    Figure f = new FreeformLayer();
    f.setBorder(new MarginBorder(3));
    f.setLayoutManager(new FreeformLayout());
    return f;
}
```

在初始化的过程中, **Root EditPart** 在自身创建完毕以后, 将开始创建子 **EditPart**。这时候会调用 **getModelChildren()** 方法来获得子模型, 所以这个方法需要重写, 并返回模型的子模型列表。然后根据设置的 **EditPartFactory**, 模型和对应的 **EditPart** 就都被创建出来了。

实现子模型的 EditPart

子模型也可以称之为节点(**node**), 通常它们需要显示一些信息。它们对应的 **Figure** 就比填充模型对应的 **Figure** 复杂多了, 有时候 **Draw2D** 默认提供一些 **Figure**(基本形状、**Label** 等)就可以

满足要求，但更多的时候你需要自己实现 **Figure**。在 **Viewer** 初始化的时候，每一个 **EditPart** 都会调用自身的 **refreshVisuals()** 方法。这个方法负责处理模型属性到图形界面显示的映射，**EditPart** 一般都需要重写这个方法，在某些复杂的场景下，这个函数有可能被拆分为几个子函数，分别实现一些功能。在我们开始对模型进行监听的时候，这些函数可能会被再次调用。

任何时候，父模型都记得重写它的 **EditPart** 中的 **getModelChildren()** 方法。

添加连接(Connection) EditPart

Connection 用于连接不同的对象，那么 **ConnectionEditPart** 则负责连接两个 **EditPart**。在 **GEF** 中如果一个 **EditPart** 对应的模型可以作为连接的源端点或者目的端点，那么它就属于 **NodeEditPart**。**ConnectionEditPart** 在创建完毕以后，将委托给它的 **Source** 和 **Target EditPart** 维护，此时 **Source** 和 **Target EditPart** 都必须重写 **getModelSourceConnections()** 和 **getModelTargetConnections()** 方法，以便 **GEF** 知道模型之间的连接关系。在 **EditPart** 创建完毕后，**GEF** 将检测它是否存在连接，若存在，通过 **EditPartFactory** 来创建对应的 **ConnectionEditPart**，然后 **ConnectionEditPart** 的 **createFigure()** 负责将 **Connection** 的 **Figure** 显示在界面上。

Connection 的 **Figure** 必须为 **Draw2D** 的 **org.eclipse.draw2d.Connection**，**ConnectionEditPart** 则需要为 **Figure** 指定 **Source** 和 **Target** 锚点(**Anchor**)，方法 **getSourceConnectionAnchor()** 等负责设定锚点。锚点必须为 **NodeEditPart**。

ConnectionEditPart 也属于 **EditPart**，它无非是为了提供连接功能而添加一些新的方法、属性，本质上它还是一个 **EditPart**，对应的模型也可以有属性，也可以在属性页里显示。同时它自身还可以作为连接的端点活着拥有子模型。

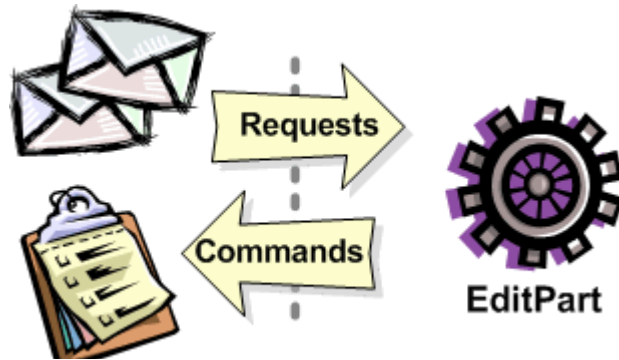
总结

至此我们都在讨论如何图形化地显示模型。**GEF** 的 **AbstractGraphicalEditPart** 类是非常重要的，通常用户只需要扩展它，根据模型重写以下几个方法：

- a) **createFigure()** – 负责创建视图(**Figure**)，注意它并不对模型的状态进行映射，映射操作发生在 **refreshVisuals()**。
- b) **refreshVisuals()** – 映射模型与视图，比如模型里设置的位置信息，需要通知视图显示出来，可以通过重写这个方法实现。负责的模型属性映射可能需要将此方法拆分为多个子方法共同完成。
- c) **getModelChildren()** – 指定模型的子模型，必须重写，否则子模型无法初始化。
- d) **getModelSource/TargetConnections()** – 指定模型对应的连接，注意此处连接指的是 **ConnectionEditPart**。。。

编辑和策略

现在你已经知道如何进行图形化的显示，接下来就开始尝试图形编辑吧。图形编辑是 **EditPart** 提供的最复杂的功能，它不但要求对模型进行修改，同时需要与视图进行交互在界面上反馈修改的结果。**GEF** 使用请求(**Request**)对交互的源进行抽象，工具和 **UI** 解释器创建请求并调用 **EditPart** 提供的 **API** 方法来完成交互。下面列出的是一些 **EditPart** 提供的 **API** 方法。



EditPart 中使用请求(Request)的方法为:

1. `EditPart getTargetEditPart(Request)`
`boolean understandsRequest(Request)`
2. `void showSourceFeedback(Request)`
`void eraseSourceFeedback(Request)`
`void showTargetFeedback(Request)`
`void eraseTargetFeedback(Request)`
3. `Command getCommand(Request)`
4. `void performRequest(Request)`

1 编辑的第一步是确定关联的 EditPart，通常情况下，它们会包括 Viewer 中选中的 EditPart 和当前鼠标位置处的 EditPart。被选中的 EditPart 也会根据它们是否响应请求而进行调整，不响应请求的 EditPart 将被忽略掉。鼠标选中的 EditPart 称为目标(Target)，目标可以由 Viewer 的助手或者 `getTargetEditPart(Request)` 方法获取。注意并非所有的交互都有目标。

2 交互中，尤其是在鼠标动作响应的交互中，EditPart 需要根据自己的角色(Role)做出特定的反馈(Feedback)。正在被操作的 EditPart 称之为源(Source)。例如，当我们在图形中拖动一个节点时，这个节点就是源，而图形就是目标。节点需要根据拖动的结果做出源反馈(Source Feedback)，比如移动到另一个的位置、修改大小等。图形也要做出相应的目标反馈(Target Feedback)。另一种情况，当我们重置连接的端点时，节点可能就变成目标，这时候它要做出目标反馈，比如删除或者新建连接。注意一些交互操作可能只对源有影响，这时候只需要源作为反馈。

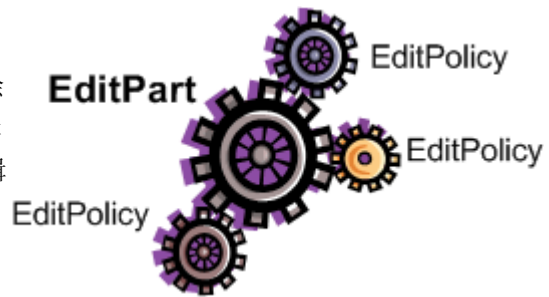
3 最终修改模型的是命令(Command)，对于指定的请求 EditPart 需要产生命令。命令同时也会用来辅助确定是否响应某个操作，如果没有对应的命令或者命令不能执行(Not Executable)，UI 界面将提示无法响应这个操作。EditPart 通过返回一个不可执行的命令来阻止某种操作，而不是将命令设置为 NULL，后者不会阻止交互的发生。当然如果所有的 EditPart 都没有对应的命令，操作一样不会响应。

4 最后，GEF 提供了一些常见的 API 方法使 EditPart 响应一些操作，这些操作不会立刻影响到对象模型。比如打开一个对话框、启用直接编辑(Direct-edit)功能。

编辑策略(EditPolicies)

EditPart 不会直接处理编辑操作，它们使用编辑策略来响应编辑操作。每一个编辑策略(**EditPolicy**)只负责一种编辑工作或者一组相关联的工作。**EditPolicy** 被设计为可以在不同的 **EditPart** 上重用，另外，它们还可以动态的改变对编辑操作的处理动作，比如布局或者连接路径变化的时候，

当上面列举的任何一个编辑方法被调用的时候(除了 `performRequest()`)，**EditPart** 将委托它的编辑策略去处理请求。**EditPart** 可能会只让第一个编辑策略处理请求，也可能让所有的编辑策略都处理请求，这视方法的具体实现而定，详细参考方法的 API 说明。



在 **EditPart** 的创建过程中，`createEditPolicies()`用来安装适当的编辑策略。编辑策略的安装需要指定角色(**Role**)，角色在 GEF 中无非就是一个身份标识而已。GEF 内置了几种常用的角色，在 **EditPart** 需要移除或者调整编辑策略时，角色将变得非常重要。针对这些角色，GEF 也提供了几种常用的编辑策略，当然大部分编辑策略都是抽象类，需要用户自己实现某些方法。编辑策略会在交互(**Interactions**)一节详细讨论。

命令(Commands)

命令在编辑的处理过程中被分发的各处，它们负责封装对应用模型的修改。

每个应用程序都由一个命令栈(**Command Stack**)，命令最好不要直接执行而是放在命令栈中再执行。命令栈将维护已经执行过的命令的重做(**Redo**)和撤销(**Undo**)等操作，直接执行的话你将无法使用这些特性。

EditPart 生命周期

当我们提到生命周期，**EditPart** 最典型的的就是激活(**activation**)和停用(**deactivation**)操作，激活过程中它将添加模型属性监听器，而停用时它会移除监听器。但是我们仍然有必要了解一下 **EditPart** 完整的生命周期。

1) 创建

最先是创建过程，绝大多数的 **EditPart** 会在 **Viewer** 的 **EditPartFactory** 中被创建出来。在创建以后，将执行以下方法：

setModel() – 如果 **EditPart** 的构造方法没有将模型对象作为参数传递进来，那么 **setModel()**将会被执行，以确保 **EditPart** 知道它维护的模型。

2) 添加到图形

setParent() – 设置 **EditPart** 的父 **EditPart**，它提供了一个用于索引它的路径，这个路径可能会用来进行 **EditPart** 的注册和访问，通过 **EditPart Registry**。

createFigure() – 创建视图(**Figure**)，注意此时父 **EditPart** 和模型都已经设置完毕，可以使用这些信息。

addNotify() -- 通知子 **EditPart** 添加完毕，这时子 **EditPart** 将进行以下操作：

1. 在 **Viewer** 的 **Registry** 中注册自己
2. 创建编辑策略
3. 刷新显示自己，然后初始化它自己的子节点或者连接(如果存在的话)

activate() – 激活 **EditPart**，表明它已经做好了编辑的准备。父 **EditPart** 在自己被激活后再激活它的子 **EditPart**，**Root EditPart** 在 **Viewer** 被创建后激活。在激活的过程中：

1. 开始监听模型。子类应当重写这个方法，以添加必要的监听器
2. 激活所有的编辑策略
3. 激活所有的子 **EditPart** 和源连接的 **ConnectionEditPart**

3) 正常使用

这时候 **EditPart** 处于正常使用状态，它开始进行一系列的处理，直到它被从 **Viewer** 中移除或者 **Viewer** 关闭。

4) 停用

deactivate() – 停用 **EditPart**，与 **activate()**相反。记得移除激活时添加的模型监听器。



在 **EditPart** 被移除以后还会有一些操作继续进行，但是如果是 **Viewer** 被关闭(正常或者非正常)，那么只保证执行 **deactivate()**。因此，**activate()**和 **deactivate()**通常需要重写而其他的方法可以忽略。

removeNotify() –通知此 **EditPart** 将要被移除，在此期间，**EditPart** 仍然可以访问上下文环境，它会进行以下操作：

1. 确定 **EditPart** 没有被选中或使用
2. 调用子 **EditPart** 的 **removeNotify()**方法
3. 从 **Viewer** 的 **Registry** 中注销自己.
4. 移除任何以自己为源或者目的的连接，一般地，连接只有在源和目的都被删除以后才会自动删除，所以此处要显示的移除相关连接，以免出现只有一个端点的无效连接

setParent(null) – 移除操作的最后一步，将父 **EditPart** 设置为 **null**。



EditPart 不会死而复生，在撤销删除时，将会产生新的 **EditPart**。因此命令最好不要引用 **EditPart**，同时 **EditPart** 不要用来存储一些在撤销时需要使用的信息。

工具(**Tools**)和绘图板(**Palette**)

GEF 的工具处理几乎可以所有的事件，在 GEF 中，**EditDomain** 用来记录当前有效地工具。应用程序可能会选择使用绘图板(**PaletteViewer**)来显示多种工具，它允许用户在不同的工具集之间切换。

工具如何工作

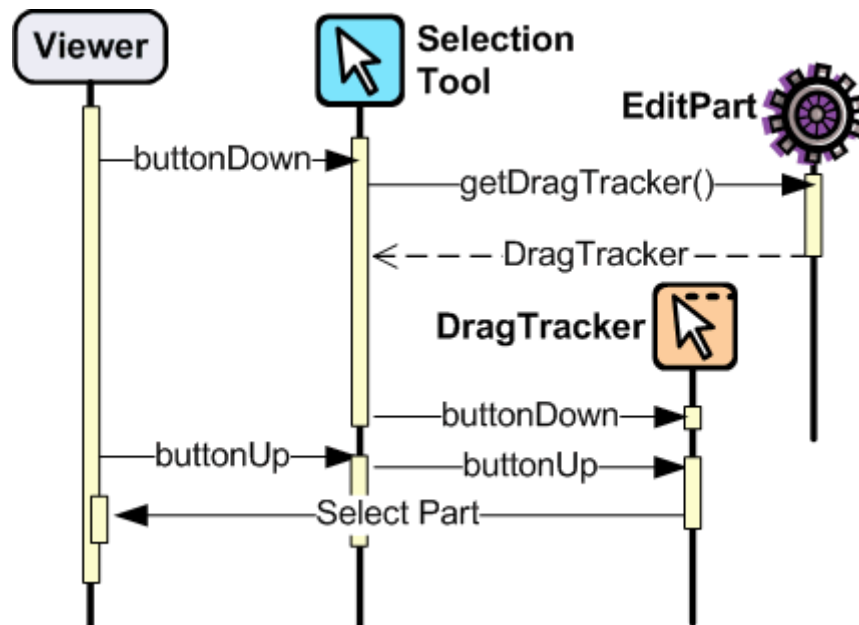
工具的实现有点像状态机，SWT 的事件(Event)驱动状态机工作。根据事件和当前状态，工具将执行特定的处理，这些处理包括：

1. 请示 **EditPart** 是否显示反馈
2. 从 **EditPart** 中得到命令
3. 在命令栈(Command Stack)中执行命令
4. 更新鼠标光标，比如有时候有手型，有时候则为指针型

工具的激活需要通过 **EditDomain**，在一个 **EditDomain** 中的所有 **Viewer** 只能拥有同一个工具。如果使用了绘图板，从绘图板中选择某个工具也会激活它。

选择工具(The Selection Tool)

选择工具是 GEF 中最基本的工具，一般也是应用程序一开始默认的工具。我们可以委托选择工具来处理任何 **EditPart**。它会从 **EditPart** 获得一种叫做拖拽跟踪者(DragTacker)的辅助工具，也会在鼠标进行拖拽时响应。拖拽(Drag)是鼠标在按下与松开之间进行的任何操作，这期间会产生很多事件。事件被转发给代理，然后根据拖拽发生的时间和位置来进行不同的处理。例如，点击一个 **Handle**(选中图形时图形边框上出现的柄状节点，一般在四个角和边框的中点位置)可能会带来形状大小的修改，或者连接端点的移动。



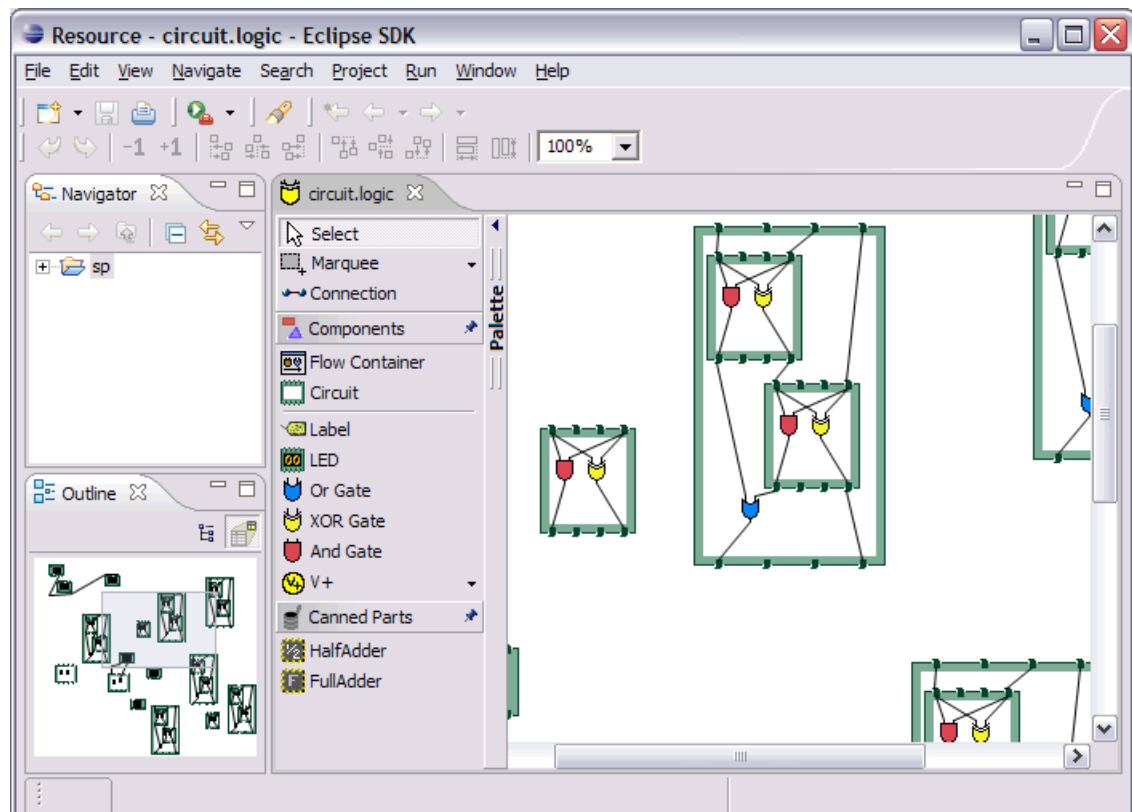
值得一提的是，选择工具并不选中 **EditPart**，所有的鼠标点击都作为拖拽来处理。当选择工具接收到一个针对 **EditPart** 的鼠标按下事件时，它会请求获得一个拖拽跟踪(Tracker)。EditPart 使用 **SelectEditPartTracker** 来获取跟踪，它不但跟踪鼠标按下事件，而且还包括鼠标松开之前的所有事件。Tracker 甚至还处理鼠标的双击事件。

更多选择工具和 Tracker 的信息请参考选择交互一节。

绘图板(Palette)

GEF 的 `PaletteViewer` 使用的是 SWT 控件，它提供了一系列的工具可供用户选择。同时 `PaletteViewer` 也可以作为拖拽的源，允许用户直接把对象从绘图板中拖到图形中。绘图板虽然不是必需的，但是使用非常普遍。

我们可以指定绘图板的位置，甚至可以放在编辑器里。GEF 提供了一个 `WorkbenchView` 来放置绘图板，同时一个自定义的面板——`FlyoutPaletteComposite` 将负责管理 `PaletteViewer` 的位置，一般地 `PaletteViewer` 位于主控件的旁边。在 `Logic` 示例中，这个面板就是编辑器的主控件，见下图：



`PaletteViewer` 显示绘图模型，一般以根模型(`PaletteRoot`)开始。`PaletteRoot` 使用可以展开合拢的目录抽屉(`Drawer`)或者组(`Group`)来组织各种绘图模型。每个组成员都可以包含绘图元素(`Palette Entry`)。绘图元素或者是一个工具，或者是一个模板(`Template`)。模板将在下面的创建一节中讲述。

绘图板提供几种显示模式，比如只显示图标。你也可以提供一个自定义的模式，允许用户自己修改或创建绘图板内容。

GEF 中的交互类型

这一节我们将讲述 GEF 框架中包含的各种交互操作和每个交互操作相关联的组件。交互可以是任何影响模型或者 UI 界面状态的行为。许多交互是可视化的，也有一些不能图形化地显示。交互可能包含一下内容：

1. 菜单操作(通常在工具栏、菜单栏和弹出菜单中)
2. 点击

- 3. 点击并拖拽.
- 4. 将鼠标停留在某处
- 5. 鼠标拖放
- 6. 按键.

同时我们还将讲述每个交互的参与者和它们的处理，包括：

- 1. 处理输入的工具 ；
- 2. 被选择的菜单操作 ；
- 3. ID 以及 **EditPart** 的工具或菜单操作处理的请求的实例。ID 由 **RequestConstants** 类定义；
- 4. 处理请求的角色。它们是 **EditPolicy** 接口定义的常量；
- 5. 用来处理交互的编辑策略；

选择(Selection)

Tools	Request	Edit Policies and Roles	Actions
SelectionTool	SelectionRequest	SelectionEditPolicy	SelectAllAction
MarqueeTool	DirectEditRequest	DirectEditPolicy	
SelectEditPartTracker	REQ_SELECTION_HANDLER	SELECTION_FEEDBACK_ROLE	
*GraphicalViewerKeyHandler	REQ_OPEN		
	REQ_DIRECT_EDIT		

在 **Viewer** 中选择是最基本和普遍的交互操作，这里讨论的大部分的交互操作都基于选中的对象。而选择本身也是一个复杂的过程。上面的表单总结了选择工具以及它涉及到的请求、编辑策略和菜单操作。

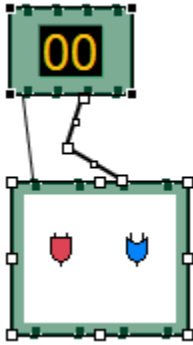
首先定义一下选中对象(Selection)，选中的是 **EditPartViewer** 里的一系列 **EditPart**。选中对象的修改是由 **Viewer** 提供的方法控制的，而不是直接操作对象本身。选中对象不会为空，即使所有的选中的对象都被清空了，**Viewer** 的根模型对应的 **EditPart** 将默认被选中。最新一次选中的对象将作为首选对象。

与选中密切相关的是聚焦(Focus)。聚焦发生在一个单独的 **EditPart** 上，用于通过键盘操作选中的对象。通过改变聚焦点，用户可以在不修改选中对象的前提下由一个 **EditPart** 切换到另一个 **EditPart**。如果没有显示的聚焦点，那么首选对象默认为聚焦对象。

Viewer 通知 **EditPart** 何时被选中，而 **EditPart** 负责显示选中和聚焦的状态。典型的，**EditPolicy** 会给图形添加一些 **Handle** 表明它被选中。在 **Logic** 示例中，使用 **ResizableEditPolicy** 给 LED 和电路部分加上了 **Handle**。黑色的部分标识首选对象。

选中对象的 **Handle** 关系到它的拖拽和大小，进一步关系到图形的布局管理器，所以通常是父 **EditPart** 安装相应的 **EditPolicy**，使得子图形显示恰当的 **Handle**。比如，安装了 **XYLayoutPolicy** 的 **EditPart** 将自动给它的子 **EditPart** 安装 **ResizableEditPolicy** 编辑策略。

连接可能通过修改连接图形的线条宽度来提示选中，比如 **Logic** 示例中的 **WireEditPart**。连接的 **Handle** 是由 **EndpointEditPart** 和 **BendpointEditPolicy** 共同负责的。



Selection Handles

选中对象的目标定位(Targeting)和反馈

选择对象的时候，**Selection Tool** 首先使用 **SelectionRequest** 来定位目标 **EditPart**。在极少数情况下，目标 **EditPart** 不可选中，这时候目标定位失败。在不断的鼠标定位的过程中，**Selection Tool** 将对当前的目标 **EditPart** 调用 **showFeedback()**方法，向它传递一个类型为 **REQ_SELECTION** 的 **SelectionRequest**。值得推荐的是，许多应用程序忽略了这个请求，因为当鼠标在图形上移动时，不断的刷新反馈会使用户崩溃。因此，当用户的鼠标停留在某个图形上时，会额外发送一个类型为 **REQ_SELECTION_HOVER** 的反馈请求。很多时候，**EditPart** 会这么处理这个请求——在图形上弹出一个对话框用来显示信息，就像工具提示条(**Tooltip**)一样。**SELECTION_FEEDBACK_ROLE** 角色就用来处理这种反馈请求，如果你需要类似的反馈处理，记得在安装编辑策略的时候使用它。

使用这些反馈请求的好处是 **Selection Tool** 不用频繁的提示这些反馈信息。比如用户开始拖拽一个图形了，他可能不希望拖拽的时候一直有一个弹出的消息框，提示你一些反馈信息。还有一点需要提示，在其他工具激活的时候，不会再有选择反馈。

使用 **DragTracker** 进行选择

一旦用户点击鼠标，一个 **DragTracker** 的实例将被创建，用于进行选择目标定位。它将返回一个 **SelectEditPartTracker** 对象或者是 **DragEditPartTracker** 对象，来进行选择，具体返回哪一种类型的对象取决于是否允许拖拽。这些跟踪者(**Tracker**)将在恰当的时候修改选中对象，同时它还会根据 **SHIFT** 或者 **CTRL** 按键是否按下来来处理选中。这意味着一次可以选择多个对象。

填充模型的 **EditPart** 不会被 **Tracker** 选中，因为它不会出现在一个多重选中的情况下。这时候会产生 **DeselectAddTracker** 或者 **MarqueeDragTracker**。记住在没有选中任何对象时，填充模型将被选中以保证选中对象不为空。

其它的选择请求

EditPart 可能会处理两个与选中相关的请求，这些请求与交互过程中鼠标点击相关。第一个是双击，在 **GEF** 称为展开(**RES_OPEN**)，这个请求用于处理打开、展开或者显示对话框等操作。也就是说，如果你想在图形上捕获双击事件，处理这个请求即可，不需要在图形上建立鼠标事件监听。另一个请求是直接编辑(**REQ_DIRECT_EDIT**)。举一个直接编辑的示例，对于一个文本框或者标签，用户可能想直接在界面上修改它的文字描述。首先选中图形，隔一段时间后再再次单击这个图形，就可以产生这个请求了。如果注册了快捷键(一般为 **F2**)，选中后点击 **F2** 可以获得同样的效果。注意需要隔一段时间再次点击，这是为了与双击请求进行区别。

菜单操作 **Selection Actions**

GEF 提供了用于全选的操作——`SelectAllAction`，它会将当前 `Viewer` 下所有的 `EditPart` 都选中。

使用键盘选择 **Selection using the Keyboard**

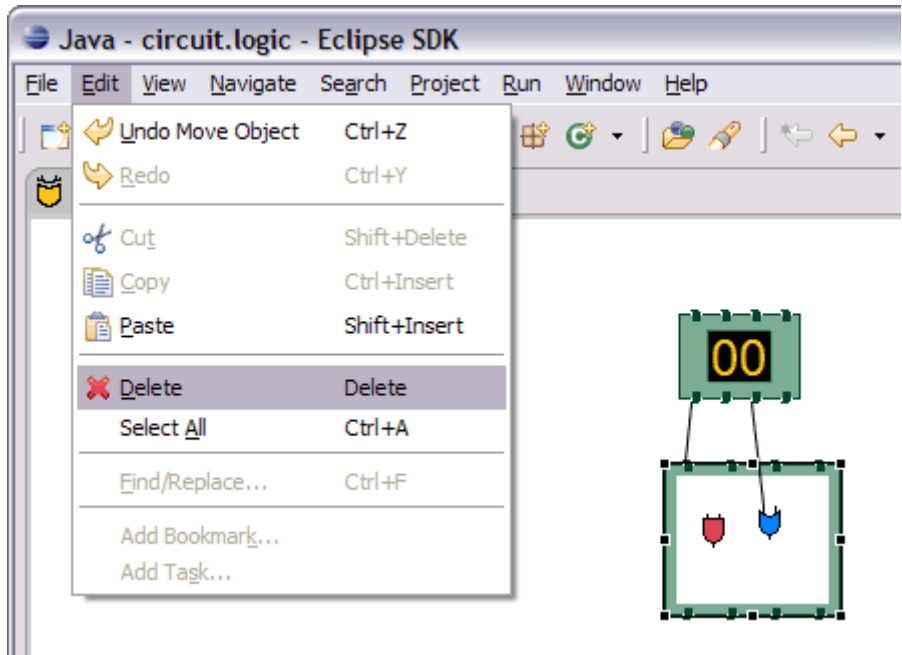
如果在图形界面上安装了 `GraphicalViewKeyHandler`，那么键盘选择就可以启用了。它负责接收当前工具传递来的按键事件，如果事件涉及到选择操作，`Selection Tool` 将负责分发这些事件。

注意 `DragTracker` 不必用于 GEF 的 `TreeViewer`，后者使用其他的方法处理选择、拖拽等操作。

基本模型操作

删除(Delete)

Tools	Requests	Edit Policies and Roles	Actions
	REQ_DELETE	COMPONENT_ROLE CONNECTION_ROLE RootComponentEditPolicy	DeleteAction



删除是所有 GEF 程序都应该支持的通用操作。工作台在编辑菜单上内置了一个全局的删除菜单操作，应用程序只用注册一下 `DeleteAction`，就可以使用删除操作了。`DeleteAction` 向当前选中的对象发送一个 `REQ_DELETE` 类型的请求，所有的 `EditPart` 都应该安装一个提示用户是否支持删除操作的编辑策略。

`EditPart` 分为基本组件(Component)和连接两种，组件是构成 `EditPart` 结构的基本元素，它们都是 `RootEditPart` 的子 `EditPart`。而连接稍有不同，它们属于源端点对象和目的端点对象拥有的。

`COMPONENT_ROLE` 关键字用于安装组件编辑策略。用户可以用过扩展 `ComponentEditPartPolicy` 并重写与删除命令相关的方法。`RootComponentEditPolicy` 用于填充模型的 `EditPart`，它的作用在于避免 `Root Figure` 被删除。**注意这里有几个概念需要区别一下：填充模型对应的 `EditPart` 为 `content EditPart`，与 `GraphicalViewer` 的 `RootEditPart` 是不一样的。**

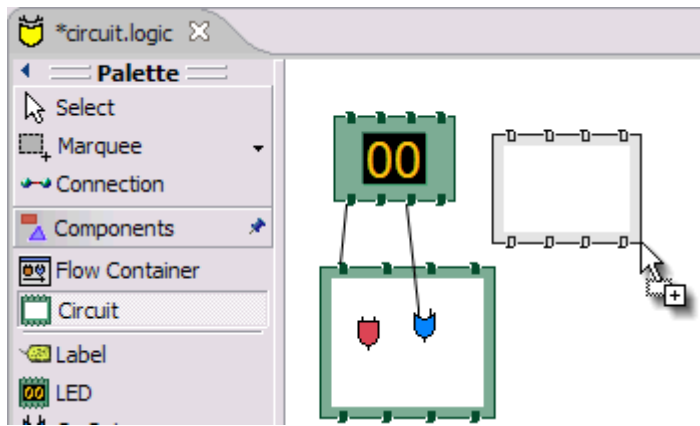
COMPONENT_ROLE 关键字用于安装连接编辑策略，用户可以通过扩展 ConnectionEditPolicy 并重写与删除命令相关的方法来实现连接的删除操作。



实现删除操作的命令往往比较困难，尤其在有连接情况下。该命令必须判断被删除的对象是否拥有连接，或者它的子节点是否拥有连接，如果存在，这些连接都必须删除。注意在删除连接的过程中可能出现重复删除，比如同时选中了连接的源端点和目的端点。

创建(Creation)

Tools	Requests	Edit Policies and Roles	Actions
CreationTool	REQ_CREATE Create	CONTAINER_ROLE LAYOUT_ROLE	CopyTemplateAction PasteTemplateAction
TemplateTransferDropTargetListener		TREE_CONTAINER_ROLE	
TemplateTransferDragSourceListener		ContainerEditPolicy LayoutEditPolicy	



CreateRequest 用来给 EditPart 创建子节点，它以 REQ_CREATE 为标识。创建过程可能发生在以下三种情况下：点击、拖拽或粘贴。该请求提供位置、对象和对象类型信息，对象和对象类型信息都由 CreateFactory 提供。创建请求隐藏了 factory，直接与创建的对象打交道。在一些时候，它还会包括尺寸信息。

产生 CreateRequest Producing CreateRequests

创建工具提供了加载光标模式来记录新创建的对象的位置。 当点击并拖动鼠标时，创建工具将跟踪用户定义的矩形的大小。 创建工具位于绘图板里，由 CreationToolEntry 实现。 松开鼠标时，创建工具可以保留设置并用于下次创建，也可以恢复为默认。

拖放也可以实现创建，拖拽的源可以是任何事物，一般来说是 PaletteViewer。拖放需要使用类型为 Template 的绘图元素，而 TemplateTransfer 负责根据 Template 拖拽源转换为拖拽目的。同时 PaletteViewer 要注册 TemplateTransferDragSourceListener 和 TemplateTransferDropTargetListener 监听器。根据创建的模型的不同，这两个监听器都要重写，并获得与模型对应的请求。

CombinedTemplateCreationEntry 支持以上两种创建方式。

处理创建请求


目标 **EditPart** 负责显示反馈和返回用于创建模型的命令。**GEF** 提供了两种处理创建操作的编辑策略。一种策略是为视图定制的,包括图形化的视图和树形结构的视图,对应的编辑策略分别为:**LAYOUT_ROLE** 和 **TREE_CONTAINER_ROLE**。

另一种策略只针对模型,应用程序可以借此来区分通过图形界面创建和非图形界面创建公用的部分。在大多数场景下,逻辑是在命令中实现的,因此这种编辑策略其实没有多大的用处。

LayoutEditPart 根据容器的布局管理器来完成创建过程,比如,如果容器使用了 **XYLayout** 布局,那么相应的创建命令中就必须使用(x, y, w, h)四个元素来对创建的子节点进行约束。不使用约束(**Constraint**)的布局可能会要求得到根据拖放位置计算出的位置索引(**Index**),并以此来确定创建的子节点的位置。针对基本的布局类型,**GEF** 内置了几个抽象的 **EditPolicy**,用户可以扩展它们来实现自己的编辑策略。

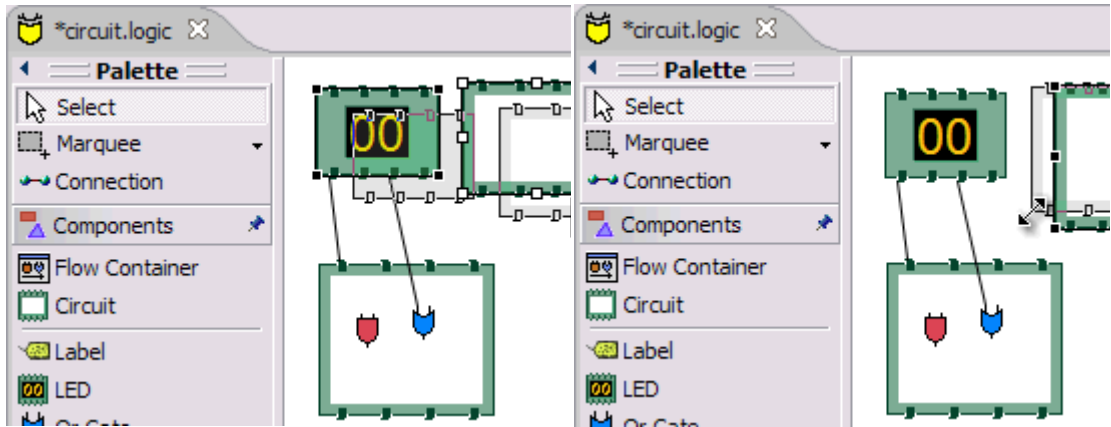
TreeContainerEditPolicy 用于支持树形结构的 **Viewer** 的子节点创建,这时候它需要确定用于节点创建和反馈的索引位置。

PasteTemplateAction 可以用于在不使用鼠标的情况下创建对象,这使得创建更方便了。同时 **CopyTemplateAction** 会被添加到绘图板中。复制时,内部机制完成数据的转换和复制,紧接着在粘贴时, **PasteTemplateAction** 会恢复转换的数据,产生 **CreateRequest** 并发送给 **EditPart**。这种交互操作中没有用到鼠标,因此新建节点的位置不确定,这也限制了在复制的时候只允许选中一个对象。

 当创建命令被重做时,它必须保证恢复到第一次创建子节点时的状态。如果某个命令创建了子节点,接下来的命令又对这个节点进行了修改,那么此时的重做操作很可能失败。

移动和调整大小

Tools	Requests	Edit Policies and Roles	Actions
DragEditPartsTracker	ChangeBoundsRequest	LayoutEditPolicy	AlignmentAction
ResizeTracker	AlignmentRequest	ResizableEditPolicy	MatchSizeAction
	REQ_MOVE REQ_CLONE REQ_ADD REQ_ALIGN REQ_ORPHAN REQ_RESIZE	ContainerEditPolicy	



移动操作

调整大小操作

DragEditPartsTracker 对基本的选择行为进行了扩展，它允许在图形编辑器中拖动选中的对象。这会带来三个潜在的交互操作：移动，重置父节点和复制。这三种操作都使用 **ChangeBoundRequest**。**ChangeBoundRequest** 继承自 **GroupRequest**，它包含大小的增量、位置的偏移和鼠标的最终位置。

拖动选中对象时，如果拖动完毕后的对象仍处在它的父节点范围中，那么将产生 **RES_MOVE** 请求。如果越出范围，那么将产生重置父节点的请求——**REQ_ORPHAN**，这个请求将被发送至原来的父节点，同时新的父节点会产生一个新的请求——**RES_ADD**。当拖动鼠标的同时按下了 **CTRL**(**MAC** 系统中为 **ALT**)，操作往往被视作复制，将产生 **RES_CLONE** 的请求，并发送给目标 **EditPart**。

以上的请求都要求目标对象处理矩形信息和鼠标位置信息，然后 **LayoutEditPolicy** 接手负责处理请求，处理的过程依请求类型而定。对于使用四元约束式的布局管理器，原始的位置和大小信息都将废弃，更新为请求记录下来的新的位置和大小信息。对于用索引来布局的管理器，鼠标的位置仍然作为生成新的布局索引的依据。

可以选择用 **ContainerEditPolicy** 来处理在 **ADD**、**ORPHAN**、**CLONE** 请求中与布局无关的命令。

调整大小 Resizing

在同一个范围内修改图形的约束将带来大小调整。注意约束(**Constraint**)的左侧或者上侧改变时，节点的位置一般也随之更改了。大小调整只对支持约束的布局管理器有效，比如 **XYLayout**。**ResizableEditPolicy** 使用了 8 个用于调整大小的 **Handle**，当对象被选中时，**ResizeTracker** 负责响应大小调整。**SHIFT** 和 **CTRL** 按键可以用来辅助完成大小调整操作。

显示在图形上 **Handle** 的数量和位置由图形所在的布局管理器决定，比如在表格中，会出现插入调整、依附、列间距和其他属性的 **Handle**。有一些布局不要 **Handle**，但是在图形的四个边角依然会出现 **Handle**，以提示用户选中。拖拽这些 **Handle** 的效果与拖动图形一样。

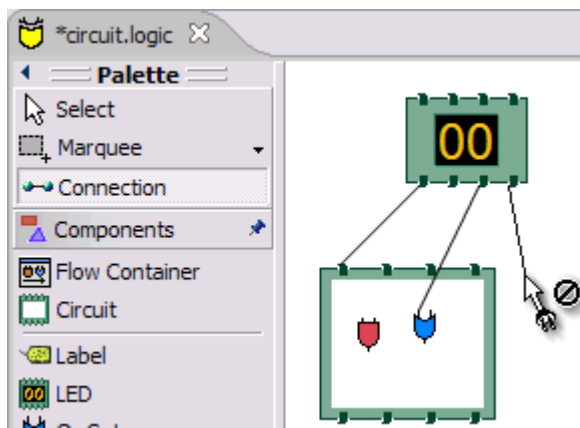
Handle 和布局的关系如此紧密，因此强烈建议使用父节点的 **LayoutEditPolicy** 来安装 **PRIMARY_DRAG_ROLE** 编辑策略角色。如果在编辑中，容器的布局管理器发生了变化了，这时候布局相关的编辑策略也会与新的布局绑定。同时所有子节点的编辑策略也将更新。

MatchSizeAction 可以根据首选对象的大小来匹配所有选中的对象。它的实现与手动调整单个对象的大小一样，也使用了同样的请求。

AlignmenAction 用于对齐多个对象的位置，它使用了 **AlignmentRequest** 请求。**AlignmentRequest** 继承自 **ChangeBoundsRequest**。在大多数情况下，**AlignmentAction** 的行为与移动操作一样，不同的是，它可以根据首选对象和对齐的方向来对齐所有选中的对象。

连接创建 Connection Creation

Tools	Requests	Edit Policies and Roles	Action s
ConnectionCreationTool	CreateConnectionRequest	GraphicalNodeEditPoli	
ConnectionDragCreationTool	REQ_CONNECTION_START	cy	
	REQ_CONNECTION_END	NODE_ROLE	



ConnectionCreationTool 用于在节点之间创建连接。创建操作需要用户选中相应的工具，然后选择连接的两个端点。点击 ESC 将撤销连接的创建。 ConnectionDragCreationTool 与前者类似，区别是它是一次鼠标的拖拽。 ConnectionDragCreationTool is similar, but the interaction is a single mouse drag. This tool can be returned as the drag tracker from a handle or even an editpart in some cases.

创建过程分为两部分，第一步是定义连接的源节点(Source)。源节点不仅仅包含节点信息，还可能还包括节点上某个特定的端口(Port，作为连接的锚点)。REQ_CONNECTION_START 类型的 CreateConnectionRequest 请求这时候会确定源端点对应的 EditPart，并且产生创建命令。这时候命令只保存了连接的源节点信息，还没有完全实现，GEF 是不会尝试执行它，甚至都不会理会它是否可以执行。

第二步是定义连接的目的节点(Target)。GEF 同样使用 CreateConnectionRequest 请求，但这次它的是 REQ_CONNECTION_END 类型的。在第一步中我们得到了一个创建的命令，保存在请求中。命令还没有完全实现，至少它要知道连接的另一个端点——目的节点的信息，这时候我们取出这个命令，将目的节点的信息保存到命令中。至此，才形成一个完整的连接创建命令。当然 GEF 也需要进行命令可执行性的判断，如果允许执行，那么将创建出一条新的连接。

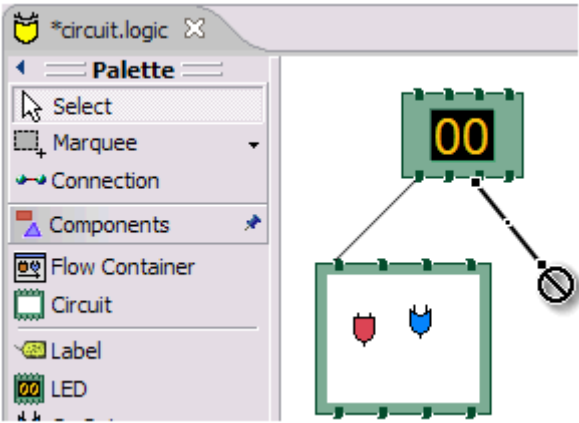
上面的过程中，source 和 target 对应的 EditPart 需要在连接创建的过程中显示一些反馈信息，比如图形可能会被一些高亮显示的依附点包围，提示它将是一个连接的端点。同时连接过程中会出现一条连接线(ConnectionFigure)，然后确定连接线的锚点。GraphicalNodeEditPolicy 负责显示连接线，而 NodeEditPart 则负责为连接提供锚点。

连接的创建需要多个组件协作完成，同时也需要它们做出图形化的反馈显示，这些反馈包括被选中的节点周围出现的依附点(锚点)和新出现的连接线。

编辑连接

Tools	Requests	Edit Policies and Roles	Actions
ConnectionEndpointTracker	ReconnectRequest REQ_RECONNECT_SOURCE REQ_RECONNECT_TARGET ET	ConnectionEndpointEditPolicy ENDPOINT_ROLE GraphicalNodeEditPolicy NODE_ROLE	

拖动连接的端点的操作称为重置连接(Reconnect)，在端点上移动连接点的位置也属于重置连接。



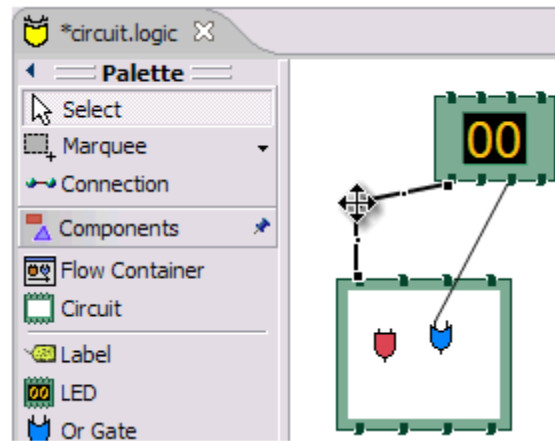
GEF 使用 `ConnectionEndpointEditPolicy` 在连接的两个端点加上一些 **Handle**，这个策略的角色为 `ENDPOINT_ROLE`。每个 **Handle** 都会返回一个重置连接的 **Tacker**，它们记录连接的端点位置。重置连接的命令由 `GraphicalNodeEditPolicy` 提供，命令的执行与连接的新端点相关。

在拖动连接端点的过程中，**Tracker** 发送 `ReconnectRequest` 请求给新的端点对象，这个对象上安装的 `GraphicalNodeEditPolicy` 生成重置连接命令。重置连接命令需要知道连接的新旧端点，然后判断连接的重置是否有效，若有效将执行重置处理。

折线连接(Bending Connections)

Tools	Requests	Edit Policies and Roles	Action s
ConnectionBendpointTracker	BendpointRequest REQ_MOVE_BENDPOINT REQ_CREATE_BENDPOINT	BendpointEditPolicy CONNECTION_BENDPOINTS_ROLE	

连接的路径有多种，有些连接使用拐点(Bendpoint)来定义连接的路径。我们可以通过安装 `BendpointEditPolicy(CONNECTION_BENDPOINTS)` 来编辑连接的拐点，而折点的修改将改变连接线的路径。这个编辑策略会在连接线上显示一系列的拐点，在用户选择修改连线路径的位置会出现一个代表拐点的 **Handle**。



每个 Handle 都有一个 `ConnectionEndpointTracker`，在用户尝试弯曲连接的时候，它会向连接对应的 `EditPart` 发送 `BendpointRequest` 请求。对于已有的拐点，请求有两种类型：`REQ_MOVE_BENDPOINT` 和 `REQ_CREATE_BENDPOINT`。当然将拐点拉回原来的位置，可以处理为移动拐点，也可以处理为删除拐点，视编辑策略的实现而定。

示例图显示了 Logic 中的连线。`ShortestPathConnectionRouter` 在连接遇到阻碍时会计算出最短的路径并绕过阻碍，这就形成了一些拐点。拐点保存在连接路由的拐点列表汇总。

