# Diploma Project

**Höhere Technische Bundeslehranstalt Leonding**

**Department of Software Engineering**

# *Eclipse Live Class Diagram*

by:

Reinhard Pointner

closing date:

03.06.2007

supervised and graded by:

Gerhard Gehrer

# Contents

# Acknowledgments

I would like to thank my supervisor Gerhard Gehrer for his support and advice on how to approach various problems and difficulties I have come across and of course for pressuring me to work hard.

I would also like to thank FabIT for giving me the idea to get involved in eclipse plug-in development and the opportunity to do this as a diploma project. In addition I would like to thank Christian Kastner, my FabIT contact, for his thoughts on the matter and his support and for giving me a rather free hand in realizing my ideas.

Finally I would like to thank my family and friends for their aid and support, especially my sister for proofreading this paper.

# Abstract

These days there are tons of UML Editors. They are designed as stand-alone applications or as plug-ins for existing Integrated Development Environments (IDE). Although these UML Editors usually provide import and export functionality to generate diagrams from source code and vice versa, diagrams and source code will drift apart in time, due to changes in the code design (adding of new classes, methods, fields, etc.). Design and code will diverge, ultimately making the original designs obsolete. Developers tend to live with this problem because the only solution would be permanent Round-Tripping, which is very troublesome and time-consuming.

The ultimate solution can be achieved by synchronizing the various views (source code, class diagram, flowchart, etc.) of the program with each other. Changes in one view will result in changes in all the other affected views. As a result, the developer can choose freely on which view he or she will work on, while not having to worry about inconsistencies.

# Kurzbeschreibung in Deutsch

Mittlerweile gibt es eine Vielzahl an UML Klassendiagramm-Editoren – einerseits als stand-alone Applikationen, andererseits direkt in eine IDE integriert. Obwohl diese in der Regel Import- und Exportfunktionen für die Generierung von Quellcode oder Diagrammen bieten, unterscheiden sich nach einiger Zeit Quellcode und Diagramm erheblich. Grund dafür ist, dass oft während der Implementierung noch Änderungen am Design des Codes (neue Klassen, Methoden, Felder, usw.) vorgenommen werden. Dadurch entfernt sich die geplante Realisierung (ursprüngliches Klassendiagramm) immer mehr von der tatsächlichen Umsetzung (Code). Dieses Problem könnte man durch ein ständiges Round-Tripping vermeiden. Diese Methode ist allerdings gerade bei vielen kleinen Änderungen sehr störend und zeitaufwendig.

Eine optimale Lösung wäre es, wenn die verschiedenen Sichten (Quellcode, Klassendiagramm, Ablaufdiagramm, ...) auf ein Programm miteinander synchronisiert wären. Änderungen an einer Sicht sollen sich vollautomatisch auch auf die anderen Sichten auswirken. Das bedeutet, dass sich der Entwickler die freie Wahl hat, über welche Sicht er den Programmcode bearbeiten will.

Eclipse Live CLD nimmt sich diese Idee zum Vorbild und stellt ein Klassendiagramm bereit, dass sich jedes mal entsprechend aktualisiert, wenn der Sourcecode geändert wird. Zusätzlich werden Element wie Klassen und Member die im Diagramm gelöscht werden auch im Quellcode gelöscht.

# Introduction

## *Motivation*

Most of the UML Editors nowadays are able to reverse engineer a diagram from the source code as well as generating the source code for a certain diagram. The Problem is, you cannot edit the sources and the diagram all at once.

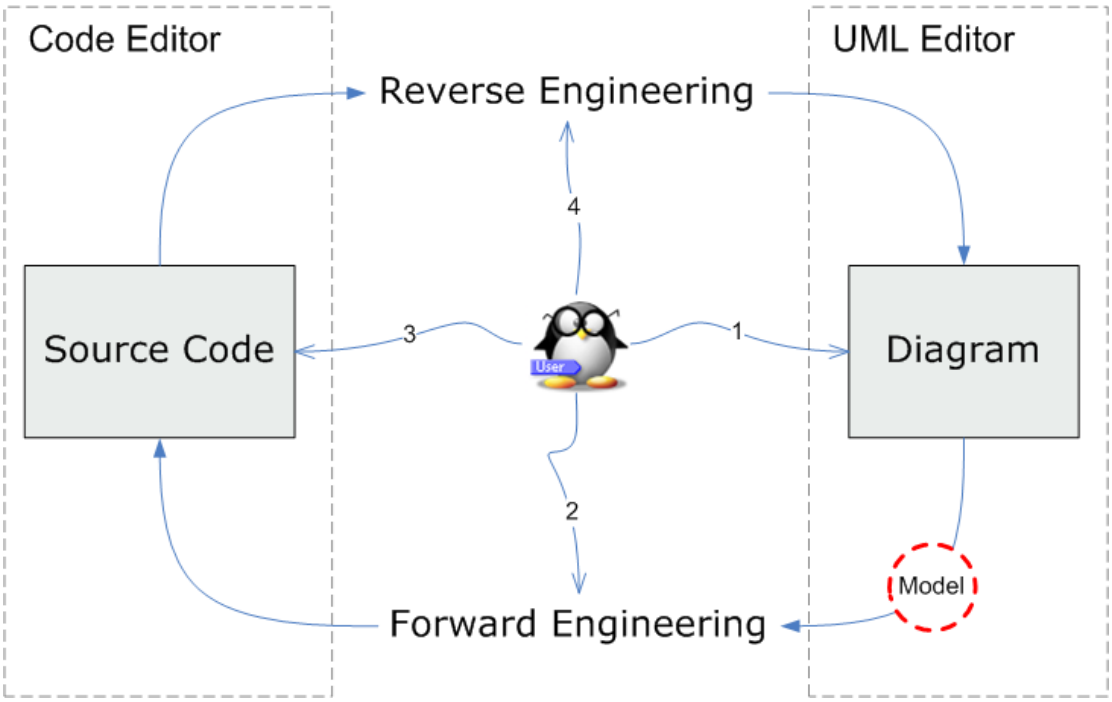



*Figure 1: Round-Tripping*

First you have to draw your class diagram, then generate the source files. While implementing the classes you most likely will find the previously designed class model insufficient and so you will begin to alter the classes to fit

your needs. The class diagram will become more and more outdated and eventually useless. To prevent this from happening the developer would have to keep source and diagram synchronized manually.

The circle in the figure above labeled "Model" is the abstract form of your program. It is neither source code nor diagram or any other kind of view. It's passed around between source code and the diagram. Ideally there's only one model which means source code and diagram describe the same model.

To keep source code and diagram synchronized at all times, the developer has to bother with forward/reverse engineering constantly to keep source code and the diagram consistent.

This is time-consuming and troublesome process, and so, in reality, the diagram is drawn once and updated never (or at least not frequently).

## *Environment*

Although there are many UML Editors, both stand-alone Applications like ArgoUML, as well as plug-ins for IDEs like AmaterasUML for eclipse. None of them can stay consistent with the corresponding source-code automatically.

One Exception is the eclipse plug-in green which able to keep imported classes and the corresponding source file synchronized but you still have to import all the classes you need into the diagram and you also have to layout them yourself.

# Task Definition

The goal is to implement an eclipse view that displays a live class diagram. "live" means that changes in the source code will immediately result in changes in the diagram and vice versa.

A new method for instance can either be added in the source code (thereby also manipulating the class diagram) or in the class diagram (resulting in corresponding updates of the source code).

Jumping from an diagram element to the according source code fragment must be possible to allow for code browsing. Refactoring abilities like renaming or deleting of elements like types, members and fields is also a vital feature.

You should also be able to export the diagram as Portable Network Graphics (PNG) or Scalable Vector Graphics (SVG). All the SVG elements, that correspond to a Java element should link to the documentation of these Java elements.

# Conceptual Design

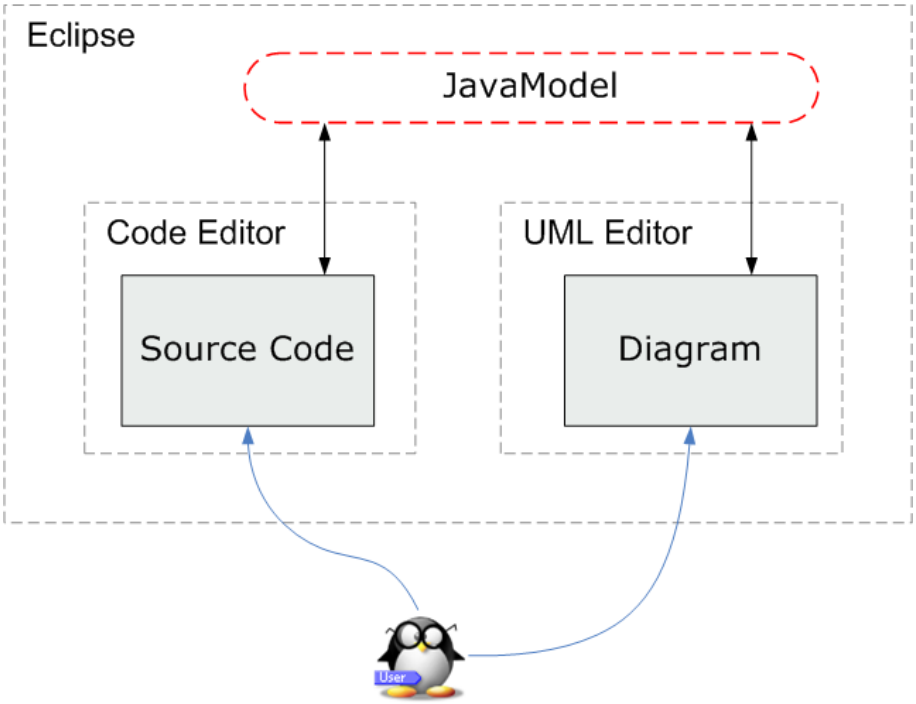



*Figure 2: Synchronization between source code and diagram*

Eclipse, or rather the Java Development Tools (JDT), use an object-oriented tree-like representation of all the projects, packages, classes, members and other Java elements, the so called Java model. Whenever the user changes the source code, the Java model will be updated, and vice versa. By changing the Java model you can change the actual sources.

You could say, your source code is merely a view of this model. A UML class diagram could simply be another view of the same model.

To provide a consistent view the diagram has to listen to changes in the model and has to be updated accordingly and changes to the diagram must be conveyed to the model.

# Eclipse Plug-in Development

## *About the Eclipse Platform*

Eclipse is a platform that has been designed for application development. By default, the platform itself does not provide a great deal of end user functionality. The platform is intended to be extended by plug-ins and provides a well-designed and an extreme extensible plug-in model. The Eclipse platform can be extended with virtually any kind of functionality (support for different programming languages, code completion, graphical editors, refactoring, integrated media players, ...).

### The eclipse plug-in framework

Tools you develop can plug into the eclipse workbench using well defined hooks called extension points.

The platform itself is built in layers of plug-ins, each one hooking to the

extension points of lower-level plug-ins, and in turn defining their own extension points for further customization.

You can also define dependencies for your plug-in to other plug-ins allowing you to access existing functionality.

Each plug-in contains one plug-in runtime class. This class must extend from Plugin (or a specialization thereof like AbstractUIPlugin) and is loaded on plug-in activation by the eclipse platform. These plug-in runtime classes are usually Singletons.

### plugin.xml

All the information the eclipse platform needs to load an hook up a plug-in is stored in a file called plugin.xml. Every plug-in has its own folder or jar-Archive containing the plugin.xml file, all the class files need for execution and other plug-in specific files.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="org.eclipse.ui.editors">
      <editor
            name="MyFile Editor"
            extensions="myfile"
            class="myfile.MyFileEditor"
            id="myfile.MyFileEditor">
      </editor>
   </extension>
</plugin>
```

*Code Snippet 1: plugin.xml for a simple editor plug-in*

## Plug-in Activation

Plug-ins are loaded on a as-needed basis. On startup, only the plugin.xml is read, so the platform knows how the plug-in is intended to be integrated into the workbench. The editor in the code example above for instance is only loaded and activated when the user opens a file of the type "myfile".

If a plug-in needs to be activated on startup, the org.eclipse.ui.startup extension must be defined.

```
<extension
        point="org.eclipse.ui.startup">
</extension>
```

*Code Snippet 2: Plug-in activation on startup*

## Interactions between the Eclipse Platform an the Plug-in

When the eclipse platform is launched, the plugin.xml files of all the plug-ins in the plugins folder are read but no Java class is actually loaded. With the information from the plugin.xml the platform knows exactly when to load which part of the plug-in (a plug-in can implement multiple extension points and can exist of multiple views, editors, etc.) and what kind preparations have to be made.

For example, if a new view is defined, then the workbench has to provide an additional menu item in the "Show View" dialog before any Java class of your plug-in is loaded, so you can activate this view.

The plug-in runtime class will only be loaded when any part of the plug-in is loaded. It will always be loaded before any other class from the plug-in and will only be loaded once. After that the part that has been activated will be integrated into the workbench.

All extension points require certain types or interfaces that must be extended from or implemented. For example, to implement the editor extension point, the class specified in the plugin.xml must extend from EditorPart. This way the platform can easily communicate with any kind of editor since they all have the same superclass and the defined set of (probably overridden) methods of EditorPart.

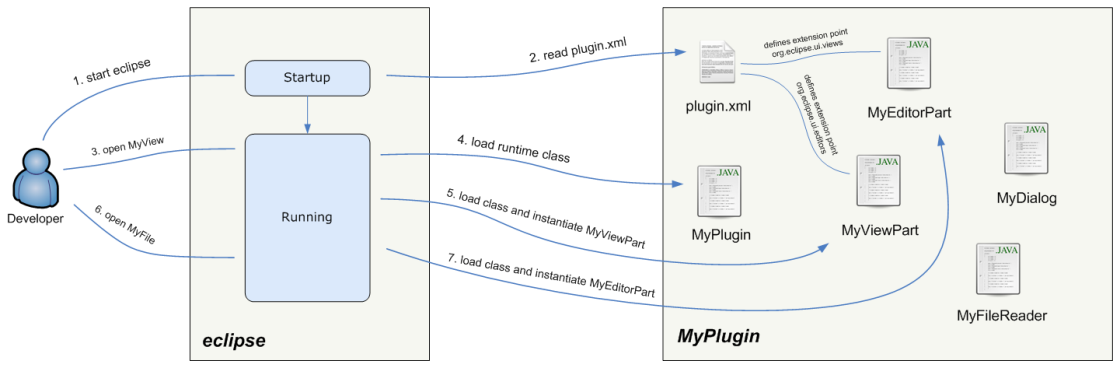



*Figure 3: Interactions between the Eclipse Platform an the Plug-in*

## *JDT*

The Java Development Tools (JDT) are a set of plug-ins that add Java specific behavior to the eclipse platform and contribute Java specific views, editors, and actions to the workbench. JDT itself is can be used as the foundation for plug-ins that need do one or more of the following things:

- Manipulate Java resources, such as creating projects, generating and refactoring of Java source code.
- Detect problems in code.
- Direct the Java IDE, for example, opening editors and launching wizards
- Add new functions and extensions to the Java IDE itself.

### JavaModel

The JavaModel can be used to get any Java element within the workspace (projects, classes, methods, fields, etc.) and provides methods for performing copy, move, rename, and delete operations on them.

```
IWorkspace workspace = ResourcesPlugin.getWorkspace();
IJavaModel javamodel = JavaCore.create(workspace.getRoot());
```

*Code Snippet 3: Getting a reference to the Java model root element*

You can use classes defined in other plug-ins like any kind of API but you have to add these plug-ins to your dependencies first. The class ResourcesPlugin for example is in the plug-in org.eclipse.core.resources, whereas JavaCore is in org.eclipse.jdt.core.

## Java Elements

The Java model is a set of hierarchically ordered Java elements. One well-known view, displaying most of these Java elements, is the Package View in eclipse.

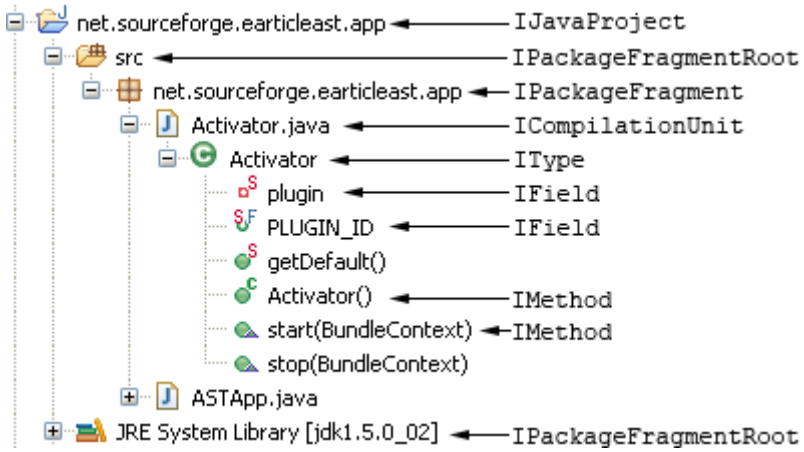



*Figure 4: Java Model Overview*

***Table of Java Elements***

| Element | Description |
|---|---|
| IJavaModel | Represents the root Java element, corresponding to the workspace. The parent of all Java projects. |
| IJavaProject | Represents a Java project in the workspace. |
| IPackageFragmentRoot | Represents a set of packages, and maps them to an underlying resource which is either a folder, JAR, or ZIP file. |
| IPackageFragment | Represents an entire package. |
| ICompilationUnit | Represents a Java source (.java) file. |
| IType | Represents either a source type inside a compilation unit, or a binary type inside a class file. |
| IField | Represents a field inside a type. |
| IMethod | Represents a method or constructor inside a type. |

## JavaUI

This class provides static methods for:

- creating selection dialogs for various kinds of Java elements.
- opening a Java editor

```
//element implements IJavaElement
IEditorPart editor = JavaUI.openInEditor(element);
JavaUI.revealInEditor(editor, element);
```

*Code Snippet 4: Opening an editor and revealing a Java element*

In order to use the class JavaUI you have to add the plug-in org.eclipse.jdt.ui to your dependencies.

### JavaCore, IElementChangedListener and IJavaElementDelta

You can listen to changes in the Java Model by simply adding a new IElementChangeListener to the JavaCore. On any kind of change in the Java model your listener will be called and given a ElementChangedEvent containing a IJavaElementDelta.

A IJavaElementDelta describes changes of an Java element between two discrete points in time. Given a delta, you can access the element that has changed, and any children or parents of that element that are affected as well.

```
JavaCore.addElementChangedListener(new IElementChangedListener() {
        public void elementChanged(ElementChangedEvent evt) {
                IJavaElementDelta delta = evt.getDelta();
                //do something with delta
        }
});
```

*Code Snippet 5: Listening to the Java model*

## *Draw2d*

The Draw2d plug-in provides the toolkit for displaying graphics and is based on the Standard Widget Toolkit (SWT). It focuses on efficient painting and layout of figures. In comparison to Graphics2D you operate on a much higher lever of abstraction. Among the main features of Draw2d are automatic connection routing and scaling.

### Figures

Figures are the building blocks of Draw2d and can be nested. Draw2d provides a lot of figure types like geometrical shapes, polylines, labels or scroll-panes and you can use LayoutManagers the layout these figures.

The Draw2d user interface is composed of a tree of figures. Each figure can paint itself and tell its children to do the same. Figures cannot paint outside of their parents bounds.

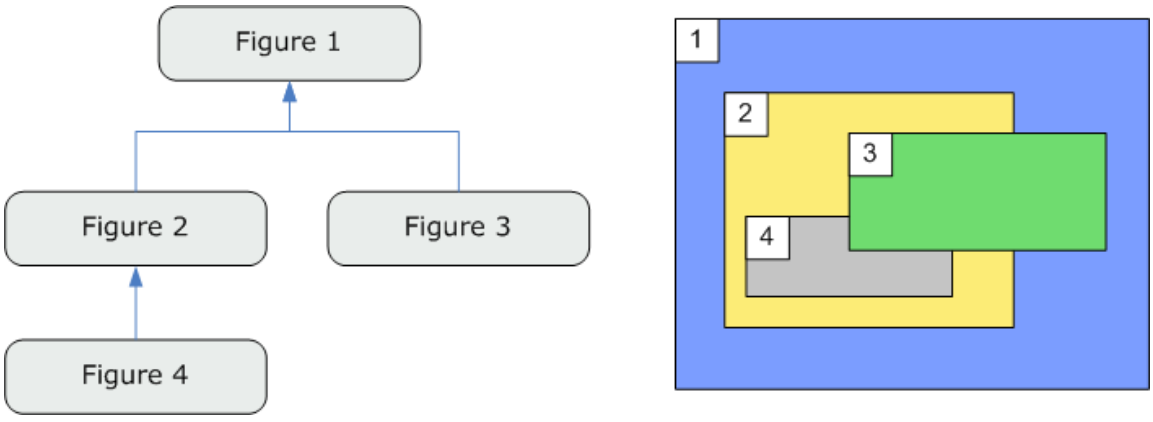



*Figure 5: Tree of figures and its graphical representation*

Draw2d itself is not bound to eclipse or GEF and can also be used in stand-alone SWT applications as well.

## *GEF*

The Graphical Editing Framework (GEF) allows you to create graphical editors for eclipse. Draw2d is used for displaying graphics. GEF provides you with every thing you need to integrate your editor into the eclipse workbench. You can also take the advantage of the many common operations (like drag and drop, snapping, aligning, etc.) or extend them for your specific needs.

GEF uses a Model-View-Controller architecture which already provides most of the functionality you need to translates user actions to changes in the model and/or view.

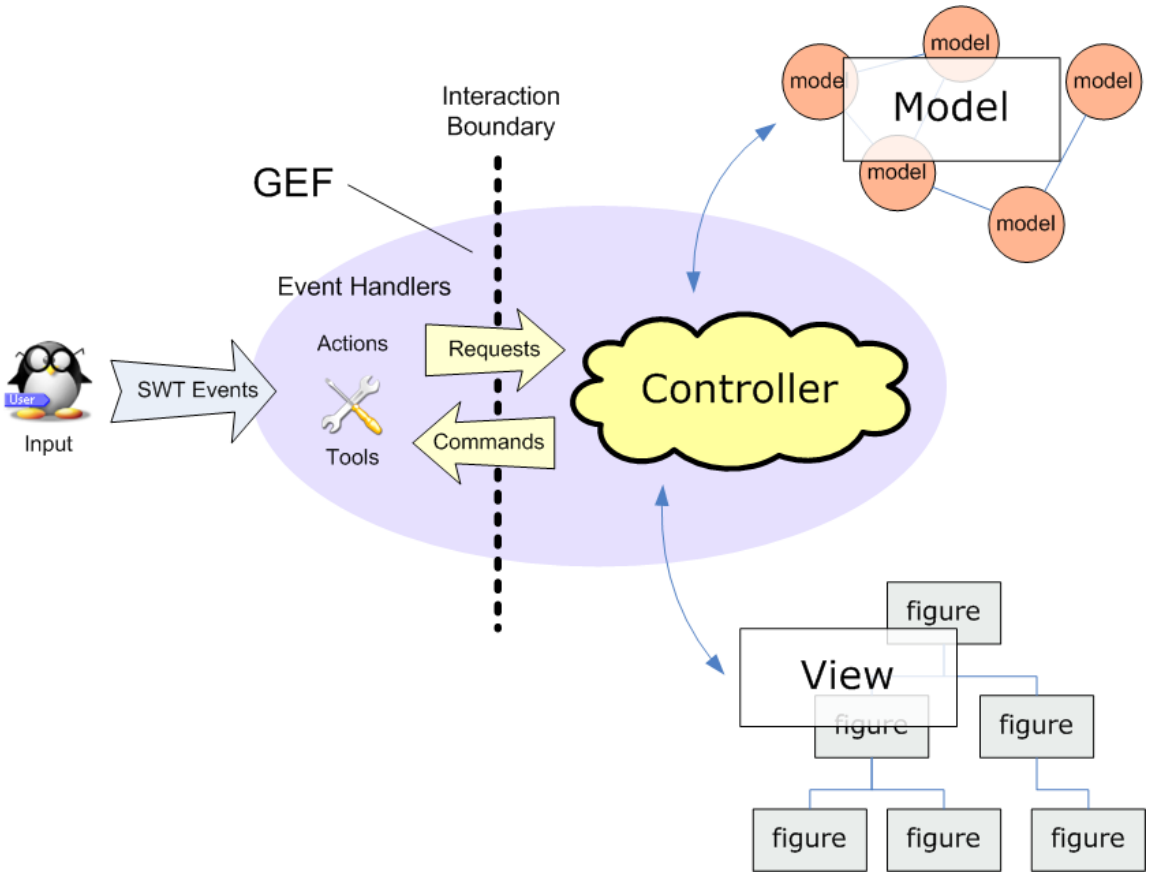



*Figure 6: GEF Overview*

The Graphical Editing Framework provides the link between the model and the view. It also provides input handlers, such as tools and actions, that turn events into requests.

The model is changed via commands. Redo and undo support is already built into the framework and works out of the box, provided the implementation of your commands support these operations.

## Model-View-Controller

### *Model*

The model is some kind of data. Although any type of model can be used with GEF, a tree-like model is often chosen because the controller design of GEF requires a tree-like structure of all controller elements.

The model elements must have some sort of notification mechanism, so the controller can listen to changes in the model.

### *View (Figures)*

The view is anything that is visible to the user. Both Figures and TreeItems can be used as view elements.

### *Controller (EditParts)*

There is usually one controller for each visual model element. The controller is called an EditPart. Each EditPart is responsible for creating and maintaining its views according to its model. EditParts are the link between the model and the view and are organized in a tree structure. Each EditPart has a parent EditPart and can have multiple child EditParts.

EditParts are also responsible for editing. EditParts contain helpers called EditPolicies, which handle most of the editing tasks and visual feedbacks and can be reused throughout your application.

## EditPartViewer

An EditPartViewer is an SWT Control (like lists, trees, tables, etc.) that manages a set of EditParts and displays their respective view. There are two types of viewers provided in GEF. A graphical viewer hosts figures, mostly used to display the diagram, while a tree viewer displays a tree of TreeItems, usually used to display the outline of a graphical view.

Each EditPartViewer contains one RootEditPart, an EditPart which has no model associated with it. In addition every EditPartViewer is responsible for maintaining a list of selected EditParts.

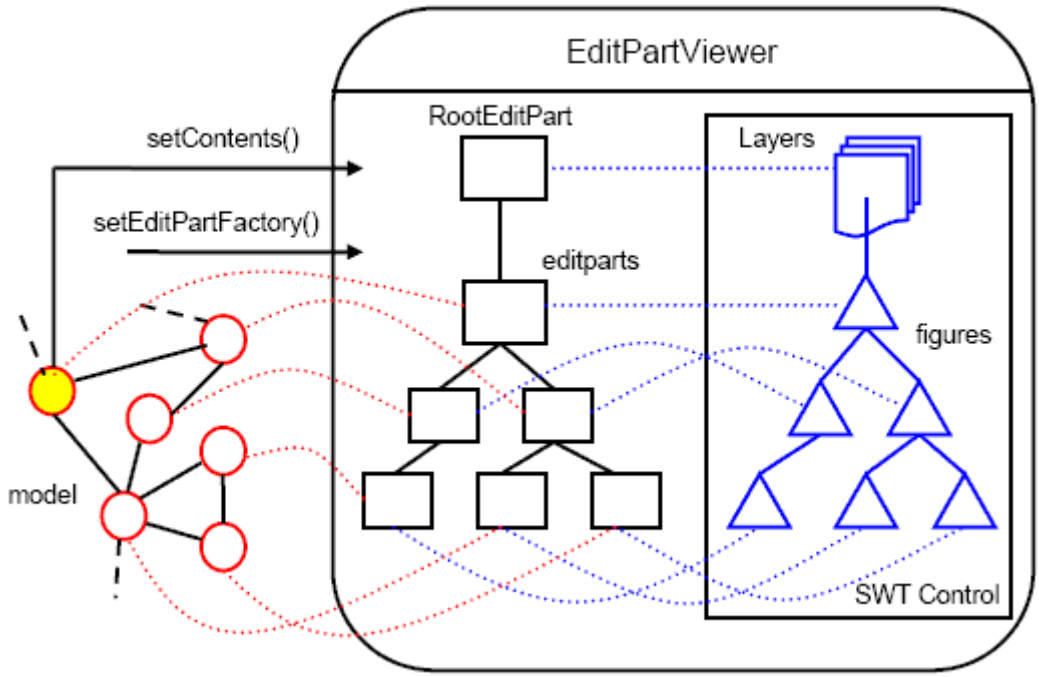



*Figure 7: Outline of an EditPartViewer*

### EditPartFactory

In the process of building or refreshing the view, GEF needs to create the the proper EditParts to represent some model elements. This process is encapsulated in an EditPartFactory.

Whenever GEF needs an EditPart for a model element, it asks the EditPartFactory to create one. You have to implement an EditPartFactory yourself for each EditPartViewer.

```
public class ShapesEditPartFactory implements EditPartFactory {

public EditPart createEditPart(EditPart context, Object modelElement) {
        EditPart part = null;

        if (modelElement instanceof ShapesDiagram) {
                part = new DiagramEditPart();
        }
        else if (modelElement instanceof Rectangle) {
                part = new RectangleEditPart();
        }
        else if (modelElement instanceof Circle) {
                part = new CircleEditPart();
        } else {
                throw new RuntimeException("Can't create part for " +
                                "model element " + modelElement);
        }

        part.setModel(modelElement);
        return part;
}
```

*Code Snippet 6: EditPartFactory*

# Implementation

## *Overview*

This chapter describes all the major challenges I came across and how to solve them.

For this project Eclipse 3.2 and GEF 3.2 were used as a foundation. You also have to keep in mind, that eclipse uses SWT and JFace as a User Interface Toolkit and not AWT/Swing.

## *Adding a new View to the Workbench*

After creating a plug-in project you only have the plugin.xml and the plug-in runtime class. In order to add a new view to your eclipse environment you have to implement the org.eclipse.ui.views extension point.

First you have to create a new class that extends from ViewPart and than you have to create a new entry in your plugin.xml.

```
<view
        name="Live Class Diagram"
        icon="hierarchy.gif"
        category="Live Class Diagram"
        class="livecld.HierarchyViewPart"
        id="livecld.HierarchyViewPart">
</view>
```

*Code Snippet 7: Segment from plugin.xml for a view extension*

### *Using a graphical EditPartViewer in a ViewPart*

GEF can be used in editors and views alike. But the main difference is, that, for GEF based editors there is already a class called GraphicalEditor which serves as a quick starting point for anyone who is new to GEF.

It will create an Editor containing a GraphicalViewer (subclass of EditPartViewer) which will be hooked up with an EditDomain automatically.

When using GEF inside a view you have to put all these parts together yourself. Adding a GraphicalViewer to your view and showing diagrams is simple, provided you already have your model and an according EditPartFactory creating your controller and view parts.

In order to react to user actions you have to hook up the viewer with an EditDomain which is responsible for translation user action to GEF requests.

```
@Override
public void createPartControl(Composite parent) {
        this.editDomain = new ViewPartEditDomain(this);
        this.viewer = new ScrollingGraphicalViewer()
        this.viewer.createControl(parent);

        this.viewer.setRootEditPart(new ScalableFreeformRootEditPart());
        this.viewer.setEditPartFactory(new ClassDiagrammPartFactory());
        this.viewer.setContents(this.modelRoot);

        this.editDomain.addViewer(this.viewer);
        ...
}
```

*Code Snippet 8: Setting up a GraphicalViewer within a ViewPart*

## *Extracting the Type Hierarchy from the JavaModel*

Eclipse already provides a way of getting the type hierarchy for given type. Such a type hierarchy provides navigation between a type and its resolved supertypes and subtypes.

```
//type implements IType
ITypeHierarchy hier = type.newSupertypeHierarchy(new NullProgressMonitor());
IType superTypes[] = hier.getAllSuperclasses(type);
```

*Code Snippet 9: Getting the all super types for a given type*

If you want to display the complete type hierarchy of all your classes (excluding API classes), you can't just start at Object and then resolve all subtypes because you would end up not only with your own classes but also with the thousands of classes of the Java API.

In order to only display the type hierarchy of your classes you have to extract all the classes you need from the Java Model.

```
//javaProject implements IJavaProject
for (IJavaProject javaProject : javaModel.getJavaProjects()) {
   for (IPackageFragmentRoot pkgRt : javaProject.getPackageFragmentRoots()) {
      //do not include APIs (jars)
      if (!pkgRt.isArchive()) {
         for (IJavaElement element : pkgRt.getChildren()) {
            IPackageFragment pkg = (IPackageFragment) element;
            for (ICompilationUnit unit : pkg.getCompilationUnits()) {
               for (IType type : unit.getAllTypes()) {
                  //do something with type
               }
            }
         }
      }
   }
}
```

*Code Snippet 10: Getting all non-API types from the Java model*

After you have collected all the desired types you have to resolve the supertype hierarchy for each of these types and finally combine your results to one type hierarchy.

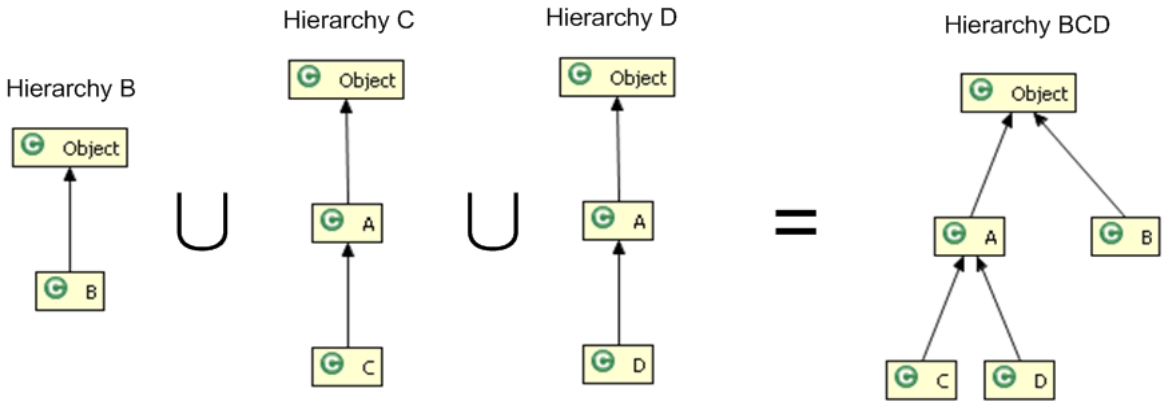



*Figure 8: Joining type hierarchies*

## *Model-View-Controller*

### Model

The root element of the model is the DiagrammRootModel. It contains one HierarchyNodeModel, the highest level node in the type hierarchy (Object). Each HierarchyNodeModel has a list of lower-level HierarchyNodeModels thereby creating a HierarchyNodeModel tree. Each HierarchyNodeModel also contains a IType which represents the actual type from the Java model.

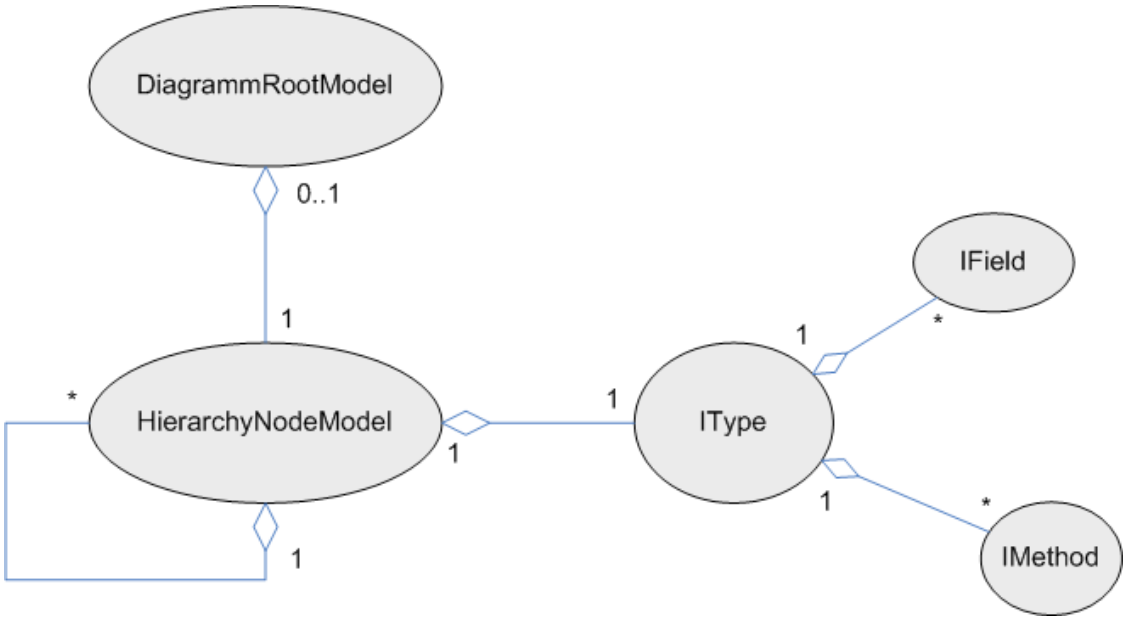



*Figure 9: Model aggregation diagram*

The following object diagram shows the model of the hierarchy shown in Figure 7.

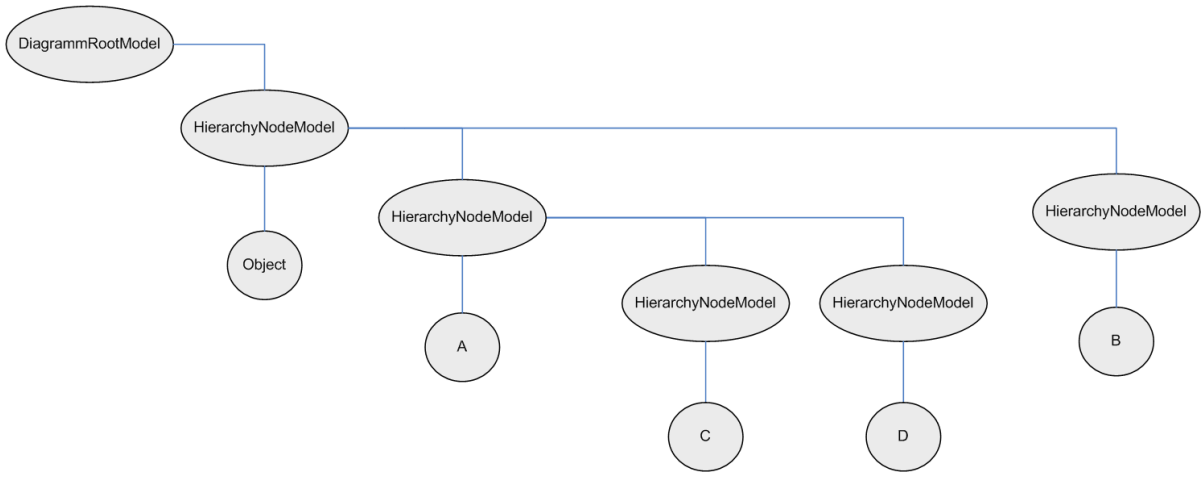



*Figure 10: Model object diagram*

## Editparts

The structure is very similar to that of the model. The EditPart tree is created directly from the model using the ClassDiagrammPartFactory.

### *Simulating multiple content panes*

Each EditPart has one content pane which is part of its view. When child EditParts are added, their visual representation is added to the content pane of their parent EditPart.

But in this case, each HierarchyNodeEditPart is in need of 2 content panes. One content pane for its child HierarchyNodeEditParts and one for its child TypeEditPart.

This problem is solved by adding two EditParts that have no real model. These two EditParts will be added to the content pane of the HierarchyNodeEditPart. One will contain for the TypeEditPart and the other one will contain for the lower-level HierarchyNodeEditParts.

For this to work, the automatic EditPart generation using the EditPartFactory has to be overridden.

```
private Object dummyTypeModel = new Object();
private Object dummyChildrenModel = new Object();

@Override
protected EditPart createChild(Object model) {
        HierarchyCompartmentEditPart part =
                new HierarchyCompartmentEditPart();
        List compartmentChildren = null;

        if (model == dummyTypeModel) {
                compartmentChildren = new ArrayList(1);
                compartmentChildren.add(getCastedModel().getType());
        } else if (model == dummyChildrenModel) {
                compartmentChildren = getCastedModel().getChildren();
        }

        part.setCompartmentModelChildren(compartmentChildren);
        part.setModel(model);
        return part;
}
```

*Code Snippet 11: Simulating multiple content panes in a AbstractGraphicalEditPart*

## Figures

The figures are created and added to the appropriate layers by the EditParts. The figures representing the view of EditParts and the connections between them are added to a separate layers. The figure below shows not only the figures representing the Java Model but also the figures that are used for layouting purposes and are normally not visible.

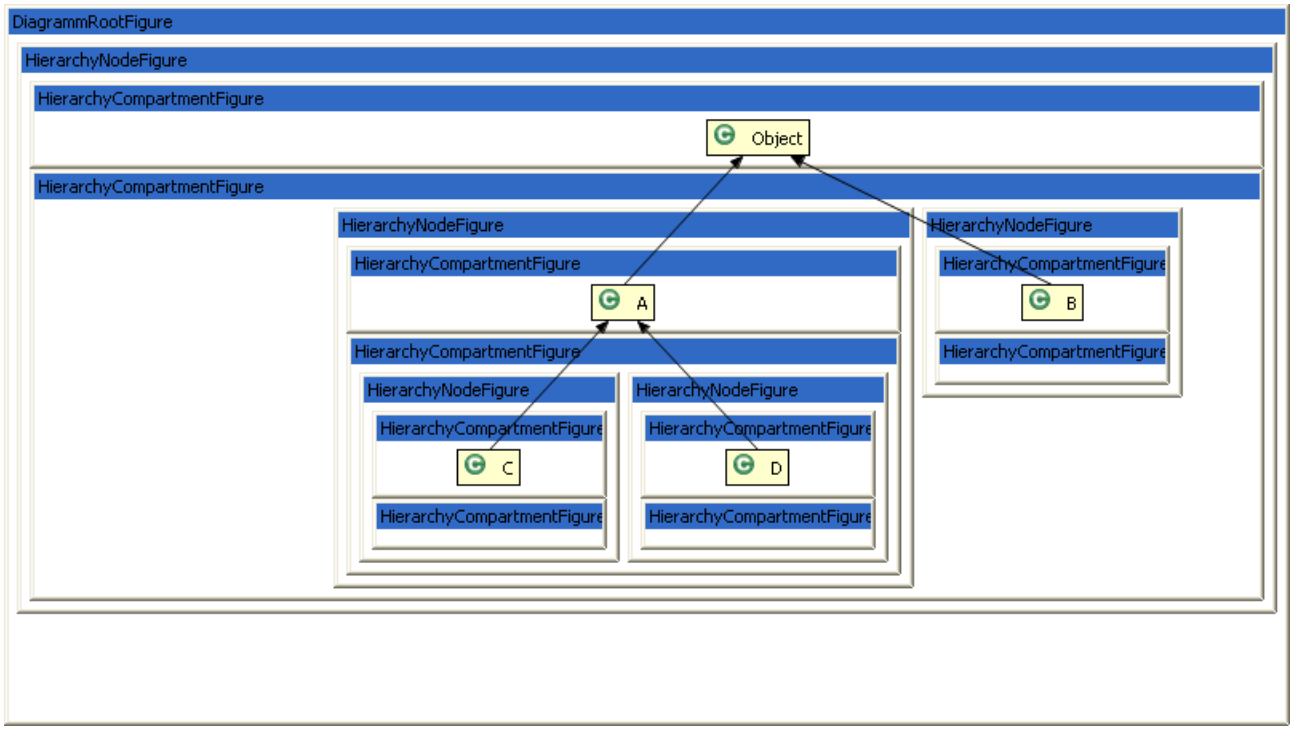



*Figure 11: Hierarchy layout using nested figures*

## *Using the icons and decorations of JDT*

The JDT provides you with icons for all sorts of Java elements as well as decorations for indicating visibility, modifiers or warnings and so on.

```
JavaUI.getSharedImages().getImageDescriptor(ISharedImages.IMG_OBJS_CLASS);
```

*Code Snippet 12: Getting the Java class icon*

```
new JavaElementImageDescriptor(imageDescriptor,
        JavaElementImageDescriptor.STATIC, SIZE);
```

*Code Snippet 13: Getting an icon with static decoration*

### *Java Element Images*

| Icon | Name | Description |
|---|---|---|
|  | ISharedImages.*IMG_OBJS_CLASS* | public class |
|  | IsharedImages.*IMG_OBJS_INNER_CLASS_PRIVATE* | protected inner class |
|  | ISharedImages.*IMG_OBJS_INNER_CLASS_PROTECTED* | protected inner class |
|  | IsharedImages.*IMG_OBJS_INNER_CLASS_PUBLIC* | public inner class |
|  | IsharedImages.*IMG_OBJS_INNER_CLASS_DEFAULT* | default inner class |
|  | IsharedImages.*IMG_OBJS_INTERFACE* | public interface |
|  | ISharedImages.*IMG_OBJS_INNER_INTERFACE_PRIVATE* | private inner interface |
|  | ISharedImages.*IMG_OBJS_INNER_INTERFACE_PROTECTED* | protected inner interface |
|  | ISharedImages.*IMG_OBJS_INNER_INTERFACE_PUBLIC* | public inner interface |

| | | |
|---|---|---|
|  | IsharedImages.*IMG_OBJS_INNER_INTERFACE_DEFAULT* | default inner interface |
|  | IsharedImages.*IMG_OBJS_ENUM* | public enum |
|  | IsharedImages.*IMG_OBJS_ENUM_PRIVATE* | private enum |
|  | *ISharedImages.IMG_OBJS_ENUM_PROTECTED* | protected enum |
|  | ISharedImages.*IMG_OBJS_ENUM_DEFAULT* | default enum |
|  | ISharedImages.*IMG_OBJS_PRIVATE* | private method |
|  | IsharedImages.*IMG_OBJS_PROTECTED* | protected method |
|  | ISharedImages.*IMG_OBJS_PUBLIC* | public method |
|  | ISharedImages.*IMG_OBJS_DEFAULT* | default method |
|  | ISharedImages.*IMG_FIELD_PRIVATE* | private field |
|  | ISharedImages.*IMG_FIELD_PROTECTED* | protected field |
|  | IsharedImages.*IMG_FIELD_PUBLIC* | public field |
|  | ISharedImages.*IMG_FIELD_DEFAULT* | default field |

## *Listening to workbench selections*

Each workbench window has its own selection service. The service keeps track of the selection and propagates selection changes to all registered listeners.

When a Java element has been selected, the selection listener will be called and given IStructuredSelection containing all selected elements. If a compilation unit was selected in the package view, the selection would contain the corresponding ICompilationUnit Java model element.

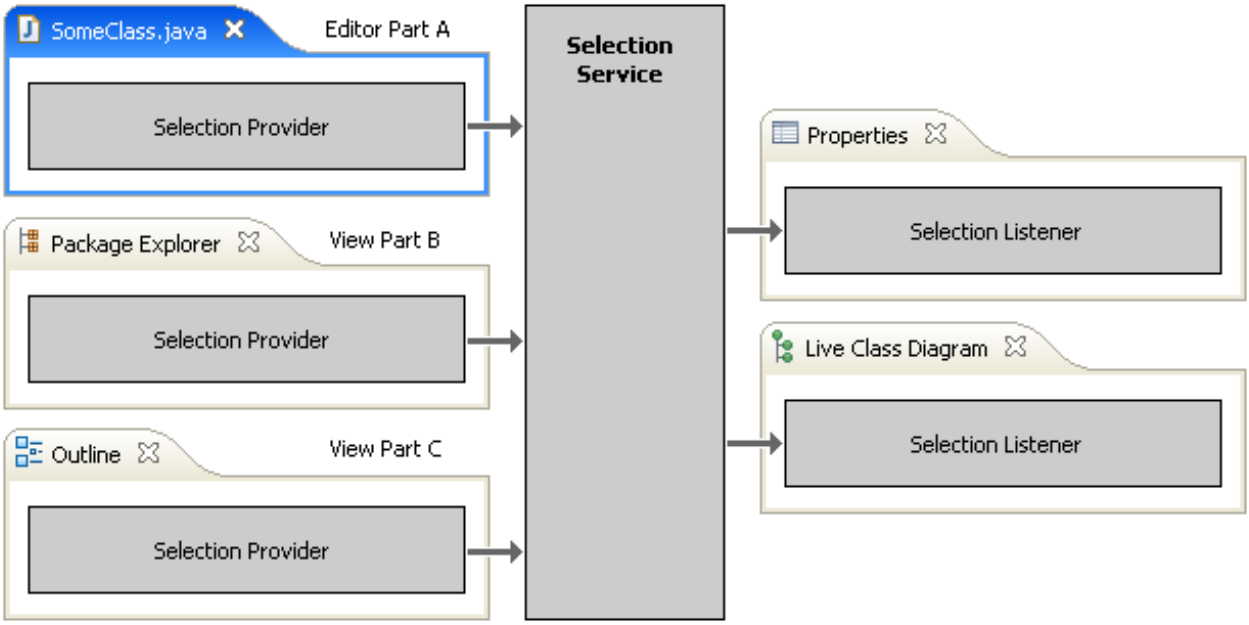



*Figure 12: Selection Service overview*

```
viewPart.getSite().getWorkbenchWindow().getSelectionService()
                            .addSelectionListener(selectionListener);
```

*Code Snippet 14: Registering a selection listener*

## *Listening to changes in the Java model*

Although it is very easy to add a listener to the JavaCore, it is not as trivial to extract useful information from the ElementChangedEvents and corresponding IJavaElementDelta tree. The delta will not only contain the changed element, but also all the parent Java models of the changed Java element because from there perspective on of their children has changed. A delta can contain many different types of changes all at once.

### JavaModelListener

The JavaModelListener is a singleton that adds its own IElementChangedListener to the JavaCore. It traverses the IJavaElementDelta tree and propagates new ElementChangedEvents with current delta to registered listener if they are interested in the Java element of the current delta. The idea behind this is to simplify listening to changes of specific Java elements.

## *Updating the Diagram*

EditParts provide a refresh() method that updates the view and child EditParts. This usually works because in almost all GEF applications refresh() is called from a user interface thread, but in this case refresh() is called within some kind of Java model listener, because when the Java model changes all IElementChangedListeners are called by the JavaReconciliationThread which has no access to the user interface and therefore refresh() has no effects.

## EditPartRefreshJob

To ensure refresh() is called from the user interface thread a new eclipse Job extending from UIJob is necessary.

```
public class EditPartRefreshJob extends UIJob {
            private EditPart refreshRootPart = null;
            private boolean refreshChildren = false;

            public EditPartRefreshJob(EditPart editPart) {
                        this(editPart, false);
            }

            public EditPartRefreshJob(EditPart editPart, boolean refreshChildren)
            {
                        super(Display.getDefault(), "Refresh Diagram");

                        this.refreshRootPart = editPart;
                        this.refreshChildren = refreshChildren;

                        //set high priority
                        this.setPriority(INTERACTIVE);
            }

            @Override
            public IStatus runInUIThread(IProgressMonitor monitor) {
                        if (refreshRootPart == null)
                                    return Status.CANCEL_STATUS;

                        refreshRootPart.refresh();

                        if (refreshChildren)
                                    refreshChildren(refreshRootPart);

                        return Status.OK_STATUS;
            }

            private void refreshChildren(EditPart part) {
                        for (Object child : part.getChildren()) {
                                    EditPart childPart = (EditPart) child;
                                    childPart.refresh();

                                    refreshChildren(childPart);
                        }
            }
}
```

*Code Snippet 15: EditPartRefreshJob*

### Changes of modifiers

If a modifier of a type, a field or a method is changed (removal of a static modifier, changes in visibility, etc.) a new EditPartRefreshJob refreshing the corresponding EditPart is scheduled. When a modifier is changed, only the icon or the label text has to be changed.

### Renaming of members

The Java Model doesn't really support renaming of Java elements. Instead of renaming a element from A to B, A is deleted and B is added.

### Adding and removing of Members

Whenever the model of a FieldEditPart oder a MethodEditPart is deleted, it will schedule a new EditPartRefreshJob refreshing the higher level TypeEditPart, which will in turn, refresh its children and thereby add/remove the FieldEditParts and MethodEditParts that do not have a model anymore. Therefore many different types of changes in fields and methods can be updated all at once.

### Adding and removing of Classes

Due to the nested layout and the tree of HierarchyNodeModels, you can only add a class to the hierarchy or remove a class from the hierarchy by rebuilding the whole hierarchy from scratch.

### *Changing the Java model*

When the user changes certain Java elements using the actions provided in the context menu like *Rename* and *Delete*. The corresponding EditPart for this model element is not notified directly.

Instead the Java model listeners are notified by the subsequent changes in the Java model. It is of no consequence whether the changes are initiated by the user or by a plug-in. The JavaModelListener will notify all affected EditParts that need to update their visuals.

In principal, the controller will first change the element, then the Java model will automatically update the source code accordingly and due to the subsequent Java model change events, the controller will update its visuals.

```
//element implements ISourceManipulation (ICompilationUnit, IMember, etc.)
element.delete(false, new NullProgressMonitor());
```

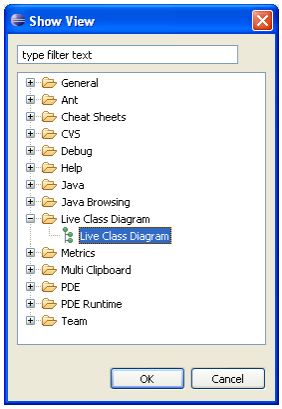
*Code Snippet 16: Deleting a Java element*

# User Guide

## *Installation*

Copy the Live CLD jar or the Live CLD plug-in folder into the /plugins folder in your eclipse installation.

## *Running Eclipse Live CLD*




After installing the plug-in, start or restart eclipse and activate the Live CLD View (Window -> Show View -> Other...). Now the Live CLD view will appear in your workbench.

You just have to click on a project, package, compilation unit or some other Java element and an according class diagram will appear.

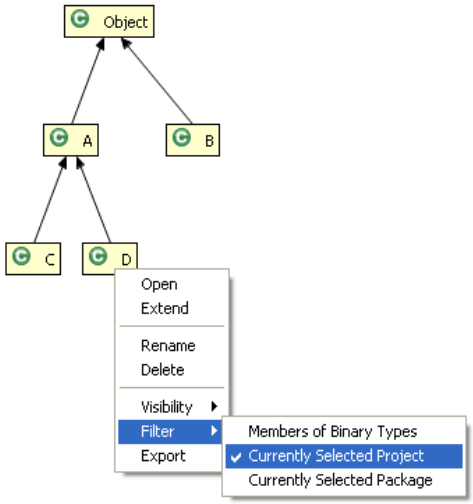
## *Tasks*

By default, the project filter is activated, that means, that the Live CLD view will always show you the type hierarchy of the currently selected project. Click on another project an the type hierarchy will update accordingly.

When selecting an element in the package view or in the outline view that is also shown in the Live CLD view, the view will try to scroll to this element.

## Open

*Open* will open the editor containing the corresponding element and then select the element in this editor.




## Extend

*Extend* will open a new class dialog, the super type will be set to the clicked class an the location an containing package will also be set, according to the super type.

## Rename

*Rename* will open a rename dialog for the selected element.

## Delete

You can delete types/fields/methods directly via the diagram. If the type is the primary type of a compilation unit, this compilation unit will be deleted.

## Visibility

By changing the diagram visibility, you can choose which visibility you want to concentrate on. For example, if protected is selected only classes/methods/fields will be shown, that have a equal or higher visibility compared to protected (default and public), all private classes/methods/fields will be hidden.

### Members of Binary Types

By default fields and methods of API classes are hidden but you can change this setting, if you want to see all the methods and fields of these API classes.

### Currently Selected Project

This filter will always show you a class diagram of the project of the last selected Java element.

### Currently Selected Package

This filter will always show you a class diagram of the package of the last selected Java element. When a project is selected, this filter will work exactly like the project filter.

### Export

You can export the current diagram as JPEG oder BMP.

When trying to export very big images you might get an OutOfMemoryException. You can solve this problem by increasing the the maximum memory for this virtual machine (change the -Xmx256m argument in eclipse.ini to -Xmx1024m or higher and restart eclipse).

# Future Prospects

Although the main features of Live CLD have been implemented, there is still much room for improvement. For example you could also indicate errors and warnings in the diagram.

As for the basic idea of a dynamically changing UML diagram, there are many possibilities. On of my ideas in early development was displaying a flow chart of the currently selected method that will update according to any changes in the source code of this method. This could be done using the Abstract Syntax Tree (AST) API also included in JDT.

I have also been offered an internship at Fabasoft for this summer and I have been asked to further develop my plug-in and combine the advantages of a live class diagram with the advantages of a traditional class diagram.

# Conclusion

Writing your first plug-in is actually rather simple because eclipse already provides wizards and templates you can use as a quick starting point. Creating a view with a text field inside, for example, is a matter of minutes.

But when I started to delve into GEF I was overwhelmed with its complexity at first. Although there are many tutorials, it takes many hours to get a basic understanding of how GEF works and how all its parts fit together. I had many problems setting up the initial Live CLD view because I did not have a profound enough understanding of GEF and its components.

In retrospect I have to say GEF is very well designed and displaying elaborate, interactive and feature-rich diagrams is in fact rather easy in comparison to the complexity of this task.

JDT also posed a problem because much of the functionality I needed was not addressed in the JDT developer guide and there are very few tutorials and articles on the more sophisticated subjects. I often had to resort the the source code of JDT to find out how certain tasks were handled there.

Personally I liked working with JDT and GEF because of the high level abstraction, at least for as long as everything worked as I expected it to work. But when there is a problem it can get frustrating very quickly because if you do not understand the underlying resources it is nearly impossible to find the source of the problem and solve it.

# Sources

[1] http://help.eclipse.org/help32/index.jsp, 6. Mai 2007

[2] http://www.eclipse.org/articles, 19. Mai 2007

[3] http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html, 19. Mai 2007

[4] http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html, 19. Mai 2007

[5] http://eclipsewiki.editme.com/GefDescription, 12. Mai

[6] http://wiki.eclipse.org/index.php/GEF_Description, 12. Mai

# List of Abbreviations

| Abbreviation | Description |
| --- | --- |
| IDE | Integrated Development Environment<br>Software that helps software developers in developing software. |
| UML | Unified Modeling Language<br>Describes a set of diagrams that visualize certain aspects of a program. |
| JDT | Java Development Tools<br>A set of plug-ins that add Java specific behavior to the eclipse platform. |
| GEF | Graphical Editing Framework<br>Framework that simplifies the creation of extensive graphical editors for eclipse. |
| AWT / Swing | Abstract Window Toolkit / Swing<br>Standard Java GUI Toolkit maintained by Sun. |
| SWT | Standard Widget Toolkit<br>Alternative GUI Toolkit to AWT / Swing maintained by the Eclipse Foundation. |
| SVG | Scalable Vector Graphics<br>XML-based file format for vector graphics |
| PNG | Portable Network Graphics<br>File format for raster graphics |

# Figures

# Code Snippets