Quick Orientation Viewer

Yiming Xia
Abstract

Nowadays, it is still difficult to keep the orientation where one is in the class structure of a software project one is programming in. In this thesis, we will introduce an Eclipse plug-in to see the class diagram of the current project with a UML class diagram displayed in a viewer.

The Eclipse platform provides a kernel program and lots of plug-in and it is feasible and convenient for us to design our own plug-in. So, we will create a plug-in that can be seamlessly integrated in the Eclipse and to show the graph in memory as UML class diagram with the help of third party tools. Some of them would be JDT (Java Development Tools) and GEF (Graphic Editing Framework).
# Table of content

1. **Introduction.................................................................................................1**
   1.1. Background for the thesis........................................................................1
   1.2. Problem Description..............................................................................1
   1.3. Motivation..............................................................................................1
   1.4. Goals and Criteria..............................................................................1
   1.5. Outline.................................................................................................3

2. **Technology Background ........................................................................4**
   2.1. UML 2.0 ..............................................................................................4
       2.1.1. Language......................................................................................4
       2.1.2. Model..........................................................................................4
       2.1.3. UML Diagrams...........................................................................4
   2.2. Eclipse Plug-in.....................................................................................6
   2.3. Draw2d and GEF ...............................................................................7

3. **Requirements .........................................................................................8**
   3.1. Users.....................................................................................................8
   3.2. Features................................................................................................8
   3.3. Use-Cases ..........................................................................................9
       3.3.1. Use-Case Model..........................................................................9
       3.3.2. Use-Cases...................................................................................9
   3.4. Functional requirements......................................................................11

4. **Architecture.............................................................................................14**
   4.1. The Eclipse Architecture......................................................................14
       4.1.1. What is Eclipse?.........................................................................14
       4.1.2. Open Architecture.....................................................................14
       4.1.3. Eclipse Platform.......................................................................15
   4.2. Draw2D and GEF framework...............................................................16
       4.2.1. Draw2D....................................................................................16
       4.2.2. GEF.........................................................................................18
   4.3. Java Development Tools (JDT)............................................................21
   4.4. Architecture of our product.................................................................21

5. **Design and implementation .................................................................23**
   5.1. Eclipse plug-in..................................................................................23
   5.2. GEF.................................................................................................23
       5.2.1. Models (Model).........................................................................23
       5.2.2. EditParts (Controller)...............................................................25
       5.2.3. Figures (View)..........................................................................26
       5.2.4. PartFactory...............................................................................28
5.2.5. GraphicalViewer ................................................................. 29
5.2.6. Request, Command and Policy ...................................... 29
5.2.7. Action & ActionGroup ..................................................... 30
5.2.8. Layout .............................................................................. 30
5.2.9. Visual Effect Improvement ............................................. 32
5.3. JDT ....................................................................................... 32
5.4. ViewPart ............................................................................. 33

6. Conclusion and Future work ............................................ 35
6.1. Conclusions ................................................................. 35
6.2. Future work ................................................................. 35

Reference .................................................................................. 37

Appendices ................................................................................ 38

Appendix A Installation .............................................................. 38
Appendix B Detailed Referenced Introduction ........................ 40
  Appendix B-1 Eclipse Components ........................................ 40
  Appendix B-2 Lightweight System ........................................... 40
Appendix C Related Works ......................................................... 41
List of Figures

Figure 2.1 UML Diagram Hierarchy Tree ................................................................. 5
Figure 2.2 Elements of Class Diagram ................................................................. 5
Figure 2.3 Package .............................................................................................. 6
Figure 2.4 Class Diagram Relations ................................................................. 6
Figure 3.1 Use-Case Model ............................................................................... 9
Figure 4.1 Eclipse Platform ............................................................................. 14
Figure 4.2 Eclipse Plug-in Platform ................................................................. 15
Figure 4.3 Draw2D framework ..................................................................... 16
Figure 4.4 Draw2D application example ..................................................... 17
Figure 4.5 Draw2D Border Examples ............................................................ 17
Figure 4.6 Draw2D Connection Examples .................................................. 18
Figure 4.7 Draw2D ChopboxAnchor ............................................................... 18
Figure 4.8 GEF framework OverView .......................................................... 19
Figure 4.9 GEF MVC Diagram 1 ................................................................. 19
Figure 4.10 GEF Connection ....................................................................... 19
Figure 4.11 GEF MVC Diagram 2 ............................................................... 20
Figure 4.12 GEF Policies and Commands ................................................... 21
Figure 4.13 Architecture of the Project ....................................................... 22
Figure 5.1 Model Class Diagram ................................................................. 23
Figure 5.2 Controller Class Diagram ........................................................... 25
Figure 5.3 View Class Diagram ................................................................. 26
Figure 5.4 some icons example ................................................................. 27
Figure 5.5 list example ............................................................................... 28
Figure 5.6 class diagram example ............................................................ 28
Figure 5.7 createPartControl example ..................................................... 29
Figure 5.8 Processing Actions Diagram ...................................................... 29
Figure 5.9 Action and Action Group Diagram .......................................... 30
Figure 5.10 Layout Solution 1 ...................................................................... 31
Figure 5.11 Layout Solution 2 ...................................................................... 31
Figure 5.12 Final results Diagram .............................................................. 34
List of Tables

Table 3-1 Requirement Feature Table .................................................................9
Table 3-2 Requirement Use-Case Table ............................................................11
Table 3-3 Function Requirement Diagram .......................................................13
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Window Toolkit</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Model Framework</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphic User Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Development Kit</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tools</td>
</tr>
<tr>
<td>LWS</td>
<td>Lightweight System</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>PDE</td>
<td>Plug-in Development Environment</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SWT</td>
<td>Standard Widgets Toolkit</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
1. Introduction

This chapter is a problem description about what will be solved in this thesis and also the motivation for doing it. Then goals and criteria about how to design and implement our program will be explained. At last, it provides an outline of the thesis.

1.1. Background for the thesis

Nowadays, there still not existing a good solution to the orientation of a project. And there are usually two ways to do it up till now. One is to do it manually, which means to draw it on the paper, and this way is not very convenient. The other ways is to use a traditional UML designer. It is a good idea to use software, but in some situation, the project may not be that huge so that we must invest so much time on it. So we came up with the idea of developing a useful Eclipse plug-in for the orientation. The Eclipse is an expandable platform itself. We will extend the viewer of the platform to achieve our purpose.

1.2. Problem Description

The problem addressed by the thesis is to design a plug-in which can display the class diagram of the currently being edited project in Eclipse and show them in a viewer on the right side of the platform. This may involve a lot of hard technique to solve the problem.

The most difficult problem is to understand the original project both from the theory and from the code. The old project has integrated and hidden this technique in the code, so we have to find every key point one by one.

Then, the next problem will be moving the necessary parts of the code into our project. It is based on a good understanding of the framework of Eclipse plug-in and GEF. Because we need to rebuild a plug-in and add some function such as to read the runtime information from the kernel of Eclipse using JDT API.

The last problem will be something like refactoring (or adaption). Although we already have some good functions in original project, we cannot just reuse it. One example is we choose “ViewPart” instead of “EditPart” to show the diagram. So some character must be rewrite because of that. To solve this problem, we can start with imitating the example in the text books. [10, 11]

1.3. Motivation

There is some difficulty nowadays to keep the orientation where one is in the class structure. The traditional way to solve this problem involves third party tools generating UML class diagrams through reverse engineering. This process is yet too far away from the development and programming work, and interrupts the workflow of a programmer, limiting his efficiency, since he needs to leave his development environment to perform the necessary actions, printing diagrams, and so on. On the other way the manual approach, sitting with pen and paper next to the computer is not convenient and flexible. So there is a need for us to provide a quick way to give the orientation in a project.

1.4. Goals and Criteria

This section describes the goals pursued by the thesis in order to solve the problem and the criteria used for validating the goals:
The first goal, which is also a basic goal, is to get familiar with UML 2.0 class diagrams including the class property and its relationships. Actually, we do not need to learn it proficiently, for there existing implemented code there, so we just need to know several important points. This goal is reached if we can list the elements and relations used in UML 2.0 class diagrams.

The second goal is to understand the old project. We need to learn both the theory of Eclipse Plug-in and GEF. Then one or more examples may be necessary. After that, we will come back to the old project code. We need to read the code back and forth, that is to say, to figure out what is each package responsible for, and how the classes cooperate with each other to serve different purpose. It may be a little bit difficult at first, because it is a project written by somebody else. In the process of understanding, we may come up to some feature that we are not interested in, but it would be better if we try to understand all of them because it may help us to remove the code we do not want correctly.

The third goal will be try to run the old project, to load a gml file and then to show the diagram in the editor. The outcome diagram should be a UML Class Diagram like boxes with a name for the class, names for methods and fields.

The forth goal is adding a simple layout for the final output diagram. We do not need a very perfect algorithm for two reasons: the first one is that it is too complex to design a good algorithm, and that is far away from our original aim, the second is that we can call a very good layout function from another project later. But at least we should be able to display the class being currently edited in the Java editor with all its methods and fields in the center of the viewer, and the next classes being part of the project (excl. library classes) which are referenced by the current class are displayed on a circle around the current class in the center.

The fifth goal is to collect the runtime information and to save it to a data structure. This goal would be reached if data can be retrieved using JDT and then use some API provided by VizzAnalyzer to convert the JDT data into another form so that it can be identified by the old project. Afterwards, the data will be shown by GEF in a viewer. Additionally, the view content should automatically updates, once a new class is active in the editor.

The sixth will be refactor. We find out that the old project use “EditPart” mode to show a diagram while in our case we prefer “ViewPart” mode. There are also some reasons list below:

- “EditPart” will lead to open a new page in the editor part while “ViewPart” will allow us to see the diagram during coding.
- “EditPart” may occasionally lead to some extra warning in the project.

The last goal is to improve the project and this is an optional work. There are mainly three points we should take into consideration.

- The speed of the execution. Updates should not take more than 1 second.
- The simple layout should be replaceable by a sophisticated layout algorithm.
- The output of the diagram would be great if we add zoom in and zoom out function, and we also need to stress the center class.

Here are some other optional features:

- double clicking a class in the viewer opens the according file in the editor
- the current method is highlighted in the editor
1.5. Outline
The structure of this paper thesis is described as follows. Chapter 2 introduces some general idea about Eclipse plug-in and GEF. Chapter 3 provides the requirements for the project. Chapter 4 describes some more detailed knowledge especially Eclipse Plug-in and GEF framework. Chapter 5 is the implementation of the theory for our project. At last, Chapter 6 is some conclusion and also some questions.
2. Technology Background

This section gives a detailed description of the UML 2.0 class diagrams, and some brief introduce of the Eclipse plug-in architecture, the Draw 2D and GEF frameworks. These technologies will be referenced throughout this thesis.

2.1. UML 2.0

The UML Unified Modeling Language (Unified Modeling Language) is designed to provide a standard language for designing and structuring applications [1]. And it has become the standard for building Object-Oriented software [2]. Let us look at the OMG specification states:

“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things as well as concrete things.” [3]

2.1.1. Language

UML is one language, not a method. We can use lots of element defined in UML to design our project, and write it down in the document. But it is not a method or procedure. There is not any process defined in UML.

2.1.2. Model

It is very important to realize the relationship between the UML model and the set of diagrams. A diagram is a graphical representation of the model and generally, it represents three different views of a model: [2]

- Functional requirements: The functional requirements of the system from the user’s point of view.
- Static structural: The static structure of the system using objects, attributes, operations, and relationships, including the class diagrams which we will use in our project.
- Dynamic behavior view: The dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects.

2.1.3. UML Diagrams

UML 2.0 has 13 types of diagrams [2], which can be categorized hierarchically in Figure 2.1 at next page.

Considering the topic of this thesis, we choose to just look into the class diagram. A class diagram is one type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.
Elements of a Class Diagram

A class diagram is primarily composed of the following elements that represent the system’s specific entities:

- **Class**: The class is mainly formed by two parts: functionality and property. The functionality is exposed to other classes as *methods*. Apart from functionality, a class also has properties. The properties of a class are called *attributes*.

  A class is represented by a rectangle. Figure 2.2 shows a typical class in a class diagram:

  ![Class Diagram Example](image)

  **Figure 2.2 Elements of Class Diagram**

  - The class name typically has the first alphabet capitalized. If your class has more than one word, capitalize the first alphabet of both words and join the two.
  - A list of attributes of your class goes in here. The syntax is `attribute: Type = "default value (if any)"
  - A list of your methods goes in here. The syntax is `Method name (List of parameters (if any)): Return type (if any)`

- **Interface**: An interface is a variation of a class. As we saw from the previous point, a class provides an implementation of certain functionality of a system. An interface, on the other hand, provides only a definition of functionality of a system. And later a separate class will implement the actual functionality.

  An interface shares the same features as a class; in other words, it also contains attributes and methods. The only difference is that that the methods are only declared in the interface and will be implemented by another class implementing the interface. [4]

- **Package**: A package provides the ability to group together classes and interfaces that are related. Grouping these design elements in a package provides better readability. [4]

  From the Figure 2.3, we can see a package is represented as a tabbed folder. A package
can also have relationships with other packages.

![Figure 2.3 Package](image)

### Relationships Between Classes

In a class diagram, obviously we cannot have classes just floating around, so we need to see the relationship between them. Figure 2.4 shows the kinds of relationships between classes [4].

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Relation</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Association</td>
<td><img src="image" alt="Diagram" /></td>
<td>When two classes are connected to each other in any way, an association relation is established. For example: A “student studies in a college” association can be shown as:</td>
</tr>
<tr>
<td>1 a.</td>
<td>Multiplicity</td>
<td><img src="image" alt="Diagram" /></td>
<td>An example of this kind of association is many students belonging to the same college. Hence, the relation shows a star sign near the student class (one to many, many to many and so forth kind of relations).</td>
</tr>
<tr>
<td>1 b.</td>
<td>Directed Association</td>
<td><img src="image" alt="Diagram" /></td>
<td>Association between classes is bi-directional by default. You can define the flow of the association by using a directed association. The arrowhead identifies the container-contained relationship.</td>
</tr>
<tr>
<td>1 c.</td>
<td>Reflexive Association</td>
<td><img src="image" alt="Diagram" /></td>
<td>An example of this kind of relation is when a class has a variety of responsibilities. For example, an employee of a college can be a professor, a housekeeper, or an administrative assistant.</td>
</tr>
<tr>
<td>2</td>
<td>Aggregation</td>
<td><img src="image" alt="Diagram" /></td>
<td>When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It is also called a “has a” relationship.</td>
</tr>
<tr>
<td>2 a.</td>
<td>Composition</td>
<td><img src="image" alt="Diagram" /></td>
<td>Composition is a variation of the aggregation relationship. Composition connotes that a strong life cycle is associated between the class.</td>
</tr>
<tr>
<td>3</td>
<td>Inheritance/Generalization</td>
<td><img src="image" alt="Diagram" /></td>
<td>Also called an “is a” relationship, because the child class is a type of the parent class. Generalization is the basic type of relationship used to define reusable elements in the class diagram. Literally, the child classes “inherit” the common functionality defined in the parent class.</td>
</tr>
<tr>
<td>4</td>
<td>Realization</td>
<td><img src="image" alt="Diagram" /></td>
<td>In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) “realizes” the contract by implementing the functionality defined in the contract.</td>
</tr>
</tbody>
</table>

![Figure 2.4 Class Diagram Relations](image)

### 2.2. Eclipse Plug-in

The Eclipse platform is structured as a core runtime program and a set of plug-ins that provide additional features [5]. Plug-ins contribute functionality to the platform by extending pre-defined extension points. When starting up the workbench, it is not like starting up a single Java program. Actually, it is activating a platform runtime, which can dynamically find plug-ins and start them whenever needed.
When we want to design our plug-in project, we can do this by defining extensions in the “plugin.xml” file. The platform has lots of extension points, but usually, we just need to implement some of them which are necessary for our system’s behavior. From the platform’s perspective, the plug-in created by us is not different from basic plug-ins which have already exist in the platform, such as resource management system.

More details about the Eclipse Plug-in will be described in Chapter 4.1.

2.3. Draw2d and GEF
The Graphical Editing Framework (GEF) allows us to create a graphical editor from an existing application model [5]. GEF consists of two plug-ins. The org.eclipse.draw2d plug-in provides a layout and toolkit for displaying graphics. Then, we can make use of the many common operations provided in org.eclipse.gef and extend them for the specific domain. Moreover, GEF employs MVC (model-view-controller) architecture.

Then let us have a look at these plug-ins: [6]
- org.eclipse.draw2d – The Draw2d plug-in
- org.eclipse.gef – The GEF plug-in (requires Draw2d)

More details about the Draw2d and GEF will be described in Chapter 4.2.
3. Requirements

This section is talking about the requirements specification for the Quick Orientation Viewer. The project is to design one Eclipse plug-in program. The plug-in will extend a viewer on side of the Eclipse workplace, and in the viewer, we should be able to see the class diagram. This requirement will involve two other components. One is to find the correct class information in the memory, the second is to use GEF to build up a corresponding model and then try to show it up. Now we will go into details and start by describing the users of the system, then we discuss the features of the system and the Use-Cases.

3.1. Users

There are mainly two groups of users who will use our project, one are programmers, and the other one are system analyzers. It is very common that one may get lost in the classes, and it will be great if there is a class diagram to help him orientate himself. As to the system analyzer, the benefit of the program is even more obvious, one just need to click one the class file he is interested in and then he will get a related class diagram back. And all of the process is done automatically.

3.2. Features

<table>
<thead>
<tr>
<th>Feature 1</th>
<th>A plug-in with a view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>This program should be run with Eclipse, it is a plug-in project. The only extension point for the program is a viewer. The input of the program is the request created by Eclipse according to the user’s operation and the output is related class diagram shown in the viewer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 2</th>
<th>JDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>In order to collect the information of the classes we need to use JDT (Java Development Tool), which is a fairly basic plug-in provided by Eclipse, we will use it to build a AST (Abstract Syntax Tree) parser then go through the tree and find all the classes we interested. That is what will be shown in the viewer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 3</th>
<th>GEF framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>GEF is another plug-in that we will use to build up a model and show it later. It is a MVC (Model-View-Controller) framework. First we will build a model for the class diagram, and then design the controller related to each model, the process will involves linking the model and controller, creating figures to the model and so on. It is an elegant framework, but its little complex, and we need to refer to the original work when designing the GEF part.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 4</th>
<th>Output with a layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>When we want to show up the diagram, there should be a good layout that will promise nodes will not overlap, and the arrows will never intersect. The Layout should also set the clicked class in the center and arrange the related class around it.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 5</th>
<th>Drag and Zoom actions</th>
</tr>
</thead>
</table>
**Description**
The diagram in the viewer provides the ability to be dragged by cursor, not only the nodes, but also the connections. And user can also zoom in/out to adjust to a better degree to see the picture.

**Feature 6 | Listener**
**Description**
There is a listener registered in the plug-in program. It will listen to the Eclipse platform, and control the viewer what to show. Once there is a new class clicked, the listener will respond the request by changing the content in the viewer quickly.

**Feature 7 | Color**
**Description**
In order to show the diagram vividly. We will add different color to each kind of relation. And the circle nodes are set a little transparent in order to highlight the center node. All the color are designed to make the diagram easier to be identify.

### 3.3. Use-Cases
This section shows an overview for the Use-Case Model and Use-Cases from the features described above.

#### 3.3.1. Use-Case Model

![Use-Case Model Diagram](image)

**Figure 3.1 Use-Case Model**

Figure 3.1 shows the Use-Case Model for the project, Eclipse plug-in, which will give a quick orientation view of the current code. It describes how the actor interacts with the system.

#### 3.3.2. Use-Cases

In the following figures, we will describe the details of the Use-Cases. Each Use-Case has a name, goal, pre-condition, post-condition, trigger events, description, extensions and alternatives.
<table>
<thead>
<tr>
<th>Use-Case UC1</th>
<th>Click class file</th>
<th>Feature: F01,F02,F03,F04,F05,F06,F07</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>To display the class diagram in the viewer.</td>
<td></td>
</tr>
<tr>
<td><strong>Pre-condition</strong></td>
<td>The viewer must be open before the click on the class file.</td>
<td></td>
</tr>
<tr>
<td><strong>Post-condition</strong></td>
<td>The viewer will show a class diagram.</td>
<td></td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>User</td>
<td></td>
</tr>
<tr>
<td><strong>Triggering event</strong></td>
<td>The click on the file</td>
<td></td>
</tr>
</tbody>
</table>
| **Description** | 1. The user selects the file in the Navigator Viewer  
2. The system will catch the name of the class file, and search it in the memory, then try to find the related files by the help of AST parser.  
3. The interested class files information will be picked up and then be translated into graph which will be done by VizzAnalyzer library.  
4. Afterwards, we will call a function to create models according to the content in the graph, for example, classes, attributes, methods and connections.  
5. The program displays the diagram in the viewer on the side of the workspace with a suitable layout and size.  
6. The drag and zoom actions were added to the editor so that the user can get the diagram information in more details.  |
| **Extensions** | None.  |
| **Alternatives** | 1. The user chooses the same file, and then the viewer will not be refreshed.  
2. The user chooses a file but it is not a class file, then the viewer will not work.  |

<table>
<thead>
<tr>
<th>Use-Case UC2</th>
<th>Analysis memory</th>
<th>Feature: F02</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>The system will analyze the file chosen in UC1 to prepare for the diagram displaying.</td>
<td></td>
</tr>
<tr>
<td><strong>Pre-condition</strong></td>
<td>The user must have chosen a class file.</td>
<td></td>
</tr>
<tr>
<td><strong>Post-condition</strong></td>
<td>The related class files will be analyzed to get different information to create the related models for GEF.</td>
<td></td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td><strong>Triggering event</strong></td>
<td>The system receives a request for the clicking on a file, and then the listener will respond to the request by analyzing the file content, and decide if it is necessary to show a new diagram, or just leave the old one in the viewer.</td>
<td></td>
</tr>
</tbody>
</table>
| **Description** | 1. The system will analyze the name of the current class.  
2. The JDT API will be used to pick up the related classes to the current one in the memory  
3. According to the analyzed information, the system then creates models for them.  |
<table>
<thead>
<tr>
<th>Extensions</th>
<th>None.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternatives</td>
<td>If the class has already been chosen last time, then the system will not refresh the figure again.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use-Case UC3</th>
<th>Display class diagram</th>
<th>Feature: F02,F04,F07</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>The system will display the current class diagram based on the user’s operation with the help of GEF framework.</td>
<td></td>
</tr>
<tr>
<td><strong>Pre-condition</strong></td>
<td>The models of the class diagram should have been created according to the classes in the memory.</td>
<td></td>
</tr>
<tr>
<td><strong>Post-condition</strong></td>
<td>The new class diagram will be displayed in the viewer.</td>
<td></td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td><strong>Triggering event</strong></td>
<td>The system pick up the related class from the memory, then the program will create the models in GEF, and after that, based on these models, the system can create EditParts according to the models, the EditParts draw the figure for the model. Afterwards, GEF connecting EditParts and models with PartFactory. At last, we set the factory in the GraphicalViewer. Besides, we will give a layout and some color to the class diagram.</td>
<td></td>
</tr>
</tbody>
</table>
| **Description** | 1. The system needs to get the classes information in the memory.  
2. Change the classes into a VizzAnalyzer graph form.  
3. Build up model from the new graph.  
4. Then generate EditParts and figures to the model and link them with PartFactory.  
5. Add a layout to the diagram.  
6. Set the model in GraphicalViewer as content. Set the configuration then it will have the class diagram displayed. |
| **Extensions** | None.                                      |
| **Alternatives** | If the current class has already been chosen, then the system will not analyze it again. |

Table 3-2 Requirement Use-Case Table

### 3.4. Functional requirements

This section describes the important functional requirements for the Quick Orientation. It contains the functions and tasks that the system is required to provide and perform as well as the corresponding Use-Cases.

<table>
<thead>
<tr>
<th>Requirement R01</th>
<th>Eclipse plug-in</th>
<th>Event / Use-Case: UC1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>The product is a plug-in project for Eclipse. It shall extend one viewer for Eclipse, the program should integrate with the workbench and work with other plug-ins</td>
<td></td>
</tr>
<tr>
<td><strong>Rationale:</strong></td>
<td>Eclipse is an extensible platform for tool integration through using plug-ins.</td>
<td></td>
</tr>
<tr>
<td><strong>Fit Criterion:</strong></td>
<td>The view content should automatically updates, when a new class is</td>
<td></td>
</tr>
<tr>
<td>Requirement R02</td>
<td>Analyze the memory</td>
<td>Event / Use-Case: UC 2</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The product shall be able to analyze the memory to get the current class and related classes information.</td>
<td></td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>The users’ operation may give the system one new class, then we need to analysis the class information so that we can decide if it is necessary to get more information or maybe it is even not a class file, we can just ignore the operation. Here we should use JDT API to get all necessary data, and then design the model according to the data.</td>
<td></td>
</tr>
<tr>
<td><strong>Fit Criterion:</strong></td>
<td>Collecting information about the current and referenced classes, providing the information to be displayed.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement R03</th>
<th>Use GEF framework</th>
<th>Event / Use-Case: UC3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The product shall have a UML class diagram displayed in the Eclipse viewer using GEF framework.</td>
<td></td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>The system will finally show a class diagram in the viewer. We can display the figure with the help of GEF (Graphic Editing Framework). GEF is a MVC (Model-View-Controller) framework. We, first, create different models for class diagram. Second, we will Create EditParts (Controller) relating to the models, and then use EditParts to create figures for us. These MVC are linked by Factory. Besides, we need to add a lot of policies and commands to the EditParts, which can enable it to be dragged and zoomed in/out.</td>
<td></td>
</tr>
<tr>
<td><strong>Fit Criterion:</strong></td>
<td>Using the Graphical Editor Framework being part of Eclipse to display the graphical parts.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement R04</th>
<th>Add a layout</th>
<th>Event / Use-Case: UC3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The class diagram shall have a suitable layout to avoid overlapping nodes and the transecting of connections.</td>
<td></td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>We should consider the layout to avoid the overlaps of nodes. In addition, we can cooperation with another student to add a more professional layout to this project.</td>
<td></td>
</tr>
<tr>
<td><strong>Fit Criterion:</strong></td>
<td>The viewer displays the current class in the center of the viewer and the referenced classes displayed on a circle around the current class in the center.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement R05</th>
<th>Zoom and Drag actions</th>
<th>Event / Use-Case: UC3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The class diagram in the viewer shall have drag and zoom in/out actions.</td>
<td></td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>Some times, the layout may not be very reasonable. So we should add drag action to move the nodes to a better position. In some other situation, There may be exist lots of nodes in the diagram. And we add zoom in/out actions to enable user to adjust the picture to a suitable size so that they can be seen more clearly. These actions may involves some new commands and policies in GEF.</td>
<td></td>
</tr>
</tbody>
</table>
| **Fit Criterion:** | The user can drag and zoom in/out the figures in the class diagram to
get a more clear displayed view.

<table>
<thead>
<tr>
<th>Requirement R06</th>
<th>Stress center class</th>
<th>Event / Use-Case: UC3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>The center class will be set to opaque while the circle classes will be set to semi-transparent.</td>
<td></td>
</tr>
<tr>
<td><strong>Rationale</strong></td>
<td>The class diagram should have a better output so that we can identify the most important relations between the nodes quickly.</td>
<td></td>
</tr>
<tr>
<td><strong>Fit Criterion:</strong></td>
<td>The current method is highlighted in the editor.</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-3 Function Requirement Diagram
4. Architecture

In this Chapter, we will describe the detailed architecture of Eclipse Plug-in and GEF. At the end of this Chapter, we will see an overview of our project, using all the theory introduced in this chapter. And this overview will be implemented in the next Chapter.

4.1. The Eclipse Architecture

Eclipse is an Integrated Development Environment (IDE). It is written in Java language, and is mainly designed for writing Java project, such as J2EE, J2ME and J2SE. But in our case, we will use it to develop a plug-in tool for Eclipse itself. To start with, let us have a glance at the Eclipse architecture.

4.1.1. What is Eclipse?

Eclipse is a platform that has been designed from the ground up for building web based project or other desktop applications. Eclipse is also designed for dynamic discovery, loading, and running plug-ins. Each plug-in focus on doing a specific task, and the user can add or remove the plug-in very conveniently.

4.1.2. Open Architecture

The Eclipse platform has an open architecture. It has a kernel program working in the system and also has lots of different tools plugged in it while new features can be added without impact to other tools. Moreover, we can plug our tools into the workbench using extension points.

The Eclipse SDK includes the basic platform plus two major tools that are useful for plug-in development. The Java development tools (JDT) implement a full-featured Java development environment. The Plug-in Developer Environment (PDE) adds specialized tools that streamline the development of plug-ins and extensions.

Except these two important tools, there are some other major components:

- The Platform runtime
- Workspace
- Team Support
- Workbench
- Help

![Eclipse Platform Diagram]

Figure 4.1 Eclipse Platform
These components are depicted in Figure 4.1. Some of the more detailed information about the components can be found in Appendix B-1

4.1.3. Eclipse Platform

- Runtime core
  The platform runtime is the kernel which discovers what plug-ins is installed at start-up, creating a registry of information about them. To save the time and memory cost, it just load the needed plug-in when it is actually needs.

- Workspace
  The Workspace is the plug-in responsible for managing the user’s resources includes the projects that the user create, the files and their changes in those projects and other resources. The Workspace also needs to notify other interested plug-ins about the resource changes.

- Java development tools (JDT)
  The Java development tools (JDT) is a set of plug-ins which allows the user to compile, test, debug, and edit Java programs. It adds Java specific behavior to the generic platform resource model, contributing java specific views, editor and actions to the workbench.

- Plug-in Development Environment (PDE)
  The Plug-in Development Environment (PDE) is very useful for the software developers who want to extend Eclipse since it allows them to construct the seamlessly integrated tools with the Eclipse environment. In addition, it supplies the development environment with specific functions and utilities that help you developing your own platform plug-ins in the Workbench.

![Figure 4.2 Eclipse Plug-in Platform](image-url)
4.2. **Draw2D and GEF framework**

GEF is a graphic editing framework using MVC (Model-View-Control), and we can design the Model part according to the business. The Model can inform the Control of all the change in it. The Control is implemented by “EditPart”, which is the kernel of GEF. As to the ViewPart, it is usually realized Draw2D, its aim is to show the model in a graphical way.

4.2.1. **Draw2D**

Draw2D provides the lightweight graphical system that GEF depends on for its display. It is packaged in Eclipse as a separate plug-in, *org.eclipse.draw2d*. Draw2D is hosted in a SWT canvas heavyweight control and manages the painting and mouse events that occur in the host canvas by delegating them to Draw2D figures [5]. Figures can be transparent or opaque, and can be ordered into layers, thus allowing parts of a diagram to be hidden or excluded from certain operations.

**Draw2D Framework**

A LightweightSystem associates a figure composition with an SWT Canvas. It is the kernel component of Draw2D, and is mainly composed of three parts: RootFigure is the root of all the figures; EventDispatcher receive the SWT events and then translate them into events on the appropriate figure. Next, paint events are forwarded to the UpdateManager, which coordinates painting and layout. Figures can be updated in ways which affect their appearance or size, and the UpdateManager ensures that only one layout occurs, followed by a repaint of only the region which was updated. [8]

**Lightweight System**

A lightweight system is a abstract system. We can find more detailed definition in Appendix B-2. Also, we can find the role of lightweight system in Draw 2D framework in Figure 4.3 on the top.

**Architectural overview**

Let us take a look at Figure 4.4, the basic structure of a standalone Draw2D application in the following example to improve understanding the architecture of Draw2D. [8]
As we can see, the shell serves as the SWT canvas that is needed to host Draw2D. The LWS class provides the bridge between SWT and Draw2D by routing mouse and paint events to the figures it contains. A root figure is then added to the LWS. The root figure is configured which a layout manager which controls the layout of any child figures that are subsequently added to it.

### Figures

Draw2D provides many kinds of figure that provide useful additional functionality. We describe some of these in the following sections.

- **Shapes**: some examples are the Ellipse, Polyline, Polygon, Rectangle, Rounded rectangle and so on.
- **Widgets**: these include various buttons, Checkbox, and the text entry figure, Label.
- **Layers**: They providing scaling, scrolling, and the ability to place figures into different layers.

### Features

- **Border**: The type of border in Draw2D contains GroupBoxBorder, TitleBarBorder, ImageBorder, ButtonBorder. [8] The features of Border are shown in Figure 4.5

![Figure 4.5 Draw2D Border Examples](image)

### Layouts

LayoutManagers are used to manage the position and size of a figure’s child figures.

- **FlowLayout**: Lays out its children into either rows or columns.
- **DelegatingLayout**: Delegates the layout of its child figures to the child figures’ locators.
- **XYLayout**: Places its children at the location and dimensions specified for the child.
Connection

- Routers: Connection routers are used to calculate the path that a connection takes in getting from one anchor to the other. Available connection routers can be found in Figure 4.6:

<table>
<thead>
<tr>
<th>NullConnectionRouter</th>
<th>BendpointConnectionRouter</th>
<th>ManhattanConnectionRouter</th>
</tr>
</thead>
</table>

Figure 4.6 Draw2D Connection Examples

- Anchors: An anchor is used to represent the ends of a connection by some figures.
  - ChopboxAnchor: it supposes the center point to be the connection point, and then calculate the point where the connection would intersect the figure to be the endpoint, as you can see in Figure 4.7.

![Figure 4.7 Draw2D ChopboxAnchor](image)

- Others: LabelAnchor, EllipseAnchor, and XYAnchor.
- Locators: Locator is used to position figures and its relation with connection.
  - MidpointLocator: being used to place figures at the midpoint of a connection.
  - ArrowLocator: being used to position decorations on the end of connections.
  - BendpointerLocator: being used to position bend point handles on a connection.
  - ConnectionEndpointLocator: being used to locate a figure near either the start or end of a connection.

4.2.2. GEF

Draw2d focuses on efficient painting and layout of figures. The GEF plug-in adds editing on top of Draw2d. The diagram below shows a high-level view of GEF. GEF can be loosely defined as the region in the middle. The framework provides the link between an application’s model and view. It also provides input handlers, such as tools and actions, which turn events into requests. Requests and Commands are used to encapsulate interactions and their effects on the model. We can see the GEF overview in Figure 4.8.[8, 12]

In MVC (model-view-controller) design, the controller is often the only connection between the view and the model. The controller is responsible for maintaining the view, and for interpreting UI events and turning them into operations on the model [8]. Figure 4.9 shows the MVC relations in GEF.
Model

The model in GEF only contacts with controller. It knows nothing about the view. In order to notify the changes of the model to the controller, the model must register controller as event listeners in it to catch these events. It must fire related events to the controller when it changes. And controller is responsible for notifying the view to update.[12]

Controller (EditPart)

There is usually one controller per visualized model object. The controller is called an EditPart. EditParts are the link between the model and the view. They are also responsible for editing. EditParts contain helpers called EditPolicies, which handle the much of the editing task.

EditParts also form their own structure. An EditPart maintains children. Usually this corresponds to a similar containment found in the model. The parent-child relationship of EditParts carries over into their figures. The parent’s figure will contain the children’s figures. Or, in some cases the parent part’s view is a compound view composed of several figures, one of which is designated as the content pane. The figure of each child is added to the content pane. The end result is that you have three separate hierarchical data structures which are approximately parallel to each other.

Connections are the exception to this simple tree structure. Connections may be stored in the model, but the EditParts are managed by the source and target EditPart at each end.
**View (Figures/Treeitems)**
The View is anything visible to the user. Both Figures and TreeItems can be used as view elements.

**Viewer**
An EditPartViewer is where EditParts display their view. There are two types of viewers provided in GEF. A **graphical viewer** hosts figures while a **tree viewer** displays native treeitems.

**RootEditParts**
RootEditPart is a special EditPart. It has no relationship with your model. It is used to associate EditPartViewer with contents which are the top EditPart in your application. It can be seen as a container for the contents. There is a “setRootEditPart()” method in EditPartViewer to assign the corresponding RootEditPart of the view.

There are only two implementations may probably be used — **ScalableRootEditPart** and **ScalableFreeformRootEditPart**.

**Creating a Graphical View of a Model**
The creating process is depicted in Figure 4.11.

First, we need to define a Model and some figures with which to View it, the next step is to put the pieces together. This means creating the EditParts that are going to work with each model and figure combination.

Most applications use the viewer ScrollingGraphicalViewer. The next step is to decide which RootEditPart to use. Its function is to setup the viewer and provide a uniform context for all of the application’s “real” EditParts.

Now we have a Viewer and its RootEditPart, so next we will actually set some contents into the viewer. The viewer’s EditPartFactory is then responsible for taking the contents and constructing the appropriate EditPart, which is then set on the root EditPart. Its figure gets added to the roots. At that point the contents EditPart will construct its children EditParts, reusing the viewer’s factory, until all of the EditParts and their views have been created. [9]
Requests, Policy and Command

The user’s editing operation will be converted to a series of Request. There are different kinds of Request, and we call them Role. GEF contains two main kinds of roles: graphical one and non-graphical one. EditPolicy achieves different Roles, and EditPolicy will create different Commands according to the Request. Then, the Command will change the Model. You can register some EditPolicy to one EditPart. The benefit is you can reuse the same Command. Figure 4.12 helps to understand these ideas better.

4.3. Java Development Tools (JDT)
JDT allows users to write, compile, test, debug, and edit programs written in the Java programming language.
The JDT is structured into three major components [5]:
- JDT Core – the headless infrastructure for compiling and manipulating Java code.
- JDT UI – the user interface extensions that provide the IDE
- JDT Debug – program launching and debug support specific to the Java programming language.

4.4. Architecture of our product
We have known pretty much about the theory and now let us take a preview of our product’s architecture. It is described in Figure 4.13.
First we need to build up a new plug-in in Eclipse platform. Then we need to add one extension in the “plugin.xml” so that we can have a **viewer** to show the figures. Apart from that, we also need to register a **listener** in the plug-in so that it will listen to the operation and update the current class diagram model. The related background knowledge has been introduced in Chapter 4.1

Second, we should use **GEF** to show our diagram. We start from building up the **Model**, and then create **EditPart** relating to every model. Next, we can associate the EditPart with **Figure**. At the same time, we will set a EditPart factory to create EditPart according to different model. After that, we want to manipulate the diagram. So we will add some policies and command to perform the drag and zoom in/out actions on the figure in GEF. The theory involved is in Chapter 4.2

Third, we need to get information from the memory. It can be achieved by using **JDT**. We will create Abstract Syntax Trees (AST) and analyze the classes in the memory then pick up the classes we interested and throw the others away. Then, we will create a GEF **Model** which derives from AST. Additionally, these implement will be placed in the listener. The theory involved is in Chapter 4.3.

Finally, we will set this model as the content to the **viewer** we created in Step 1. This is a general express of our architecture, and in the next Chapter, we will discuss the detailed implementation. And its theory has been included in Chapter 4.1.
5. Design and implementation

Now we will go into the design and implementation of the project applying the theory we discussed in the last chapter.

5.1. Eclipse plug-in

First of all, we should create a new plug-in project. File -> New -> Project... Choose “Plug-in Project”, then input the project name, go to next page, and then choose “Plug-in with a view” in Available Templates. Now, we get a plug-in with “view extension” already added in it. With this “view”, we can show the Class Diagram later. But before we can display the picture, we need to build the MVC framework for the diagram, which can be implemented by GEF; another very import step is to collect the Class information in the memory, and this can be achieved by JDT. [10, 11]

5.2. GEF

Now, we will go into the most important part of this project. In this part, we will build up our picture’s MVC, and then let it be shown in the “Viewer”. But actually, this GEF instance derives from another project, so we just need to make some small change according to the situation in our project.

5.2.1. Models (Model) [12]

![Model Class Diagram]

Element:
The Element is an abstract class to provide a support for the notification of the events.

It has one attribute. It is a data member of the `java.beans.PropertyChangeSupport` type,
which encapsulates the list of the listeners.

And there are three methods in Element:
- addPropertyChangeListener(PropertyChangeListener)
- remove PropertyChangeListener(PropertyChangeListener)
- firePropertyChange(String, Object, Object)

The first two methods are used to add and remove listeners in the occurrence EditPart associated with the object; and the last one is to fire changes of the model to listeners.

In GEF, the listener of the model is its EditPart. We will set it to be a listener registered in the model in the method active() in EditPart later.

A list of data members public static final String are used to define the various properties prone to notification. For example, the member public static final String PROP_LOCATION = “LOCATION” in the Node class is to define the location of the node whose location can be changed by drag action in the Viewer.

Diagram
The class Diagram that extends the Element class is used to store the whole information of all figures. It is served as a container of the models. In this case a class representing the diagram will have the list of the classes of the models as a data member.

Node
Class Node also extends the Element class. It represents an element of the diagram, which has a data member, which can store its position and size in the diagram. It also contains a name attribute to represent its name; the inputs and outputs list the incoming and outgoing connections on it.

Subject
Actually, subject is used to replace class because it is a key word in java programming language. The Subject extends Node class, so it inherits all the properties of Node. In addition, it adds its own properties: an AttributeList, which contains all the attributes of the class, and a MethodList, which contains all the methods of the class.

ListModel, AttributeList & MethodList
ListModel is the super class of AttributeList and MethodList. It contains an elementList list used to add, remove and get the elements. Afterwards, the AttributeList and MethodList extend the ListModel class.

In AttributeList, the elementList is used to deal with the attributes while the MethodList is used to handle the methods.

Attribute & Method
The Attribute class represents the attribute model. It contains these attributes:
- Id
- Identifier
- Type
- Visibility

Their names represent their meanings. And especially, the “Id” is a unique identifier of the attribute differing from others.

Similarly, the Method has the following properties:
For different relations among classes, we create one class for each kind of relation. So we have association, composition and inheritance. Every relation has a source node and a target node.

5.2.2. EditParts (Controller) [12]

EditPart are responsible for the Controller part of MVC framework, that is to say, they will bridge the model and figure. We will put them into the umlclassdiagram.parts package.

Figure 5.2 Controller Class Diagram

As we can see, from the diagram above, all EditParts extend the AbstractGraphicalEditPart class, which is particularized for the draw2d figures. There are two basic abstract methods to implement:

1. IFigure createFigure(): To return the figure associated with the model with this EditPart. The return result is an object of IFigure. Usually it is the figures in umlclassdiagram.figures package. However, here we create the Diagram Figure and Relation Figure in this method for the DiagramPart and different kinds of relation EditPart.
   - The figure corresponding to Diagram model is a figure with FreeformLayer in GEF. It is a layer that allows the user to extend it in all 4 directions.
   - For the figures of the Relations, it is a line called Polyl ineConnection in GEF used to connect two classes in default. We would add different decorations to make them look distinctive.

2. Void createEditPolicies(): Install different EditPolicies from this EditPart. If there is no need to have any actions on a model, its EditPart can have nothing in this method. In our project, we will add drag and zoom in/out actions on the figure class.

There are also some other common methods.
List getModelChildren(): This method will be called by GEF to obtain the list of the objects of the model which are regarded as the children of the model of current EditPart.

- In our project, the method in DiagramPart returns the Diagram model object, which contains a list of Subject model object.
- But in SubjectPart, it returns a list contains its attributeList and methodList.
- Similarly, in the AttributeListPart it returns the list of attributes belong to the class who have this attributeList as well as in the MethodListPart it returns the list of methods.

IFigure getContentPane(): If the parent EditPart’s figure is not the direct parent of the child EditPart’s figure, we shall override this method to get correct figure.

In our project, this will happen to SubjectPart, who has an attributeList and a methodList containing the class attributes and methods.

Void refreshVisual(): This method will be called when it is necessary to synchronize the figure and the model.

In our project, it is necessary to adjust the properties of the figure according to the model and if required to tell the parent part that the child part and its figure are to be constrained to the given constraint.

With the five methods described above, it is pretty enough to post a model without connections.

5.2.3. Figures (View) [12]

Now, we shall create figures for each model. We put the figures in umlclassdiagram.figures package.
In GEF, the code of the figures is completely independent of the model. A figure does not have any reference towards the model, which it represents. Thus the figures must save the various properties of the model, which are to be represented in data members and provide methods to make it possible to adjust them.

### Icon image

First, we will set some images for the visibility of the attributes and methods to replace the default “+” and “-” in the UML class diagram. These images will be saved in the `umlclassdiagram.figures` package. And AttributeFigure and MethodFigure will use these images.

### MethodFigure & AttributeFigure

In the Figure above, class MethodFigure and AttributeFigure extending `org.eclipse.draw2d.Label` represent the method and attribute figure.

- The method `getVisibilityIcon(int)` is to get the image for the visibility icons depending on the visibility of the attribute or method. The icon is an image as described above. Here comes an icons example in Figure 5.4.

```plaintext
-attributes :
  - attribute1: int
  - attribute2: String
  - attribute1: int

-methods :
  - method2(int, ClassA): no_type
```

Figure 5.4 some icons example

- Green: the visibility of `public`
- Red: the visibility of `private`
- Yellow: the visibility of `private protected`

- The method `setSignature(int, String, String)` in both two classes is to adjust the signature of the method or the declaration of the attribute to the properties of the Label.

- The method `getMinimumSize(int, int)` is redefined to return a new Dimension (0,0).

This label is intended to be contained in a container managed by ToolbarLayout, and this layout manager never allocates the elements with a size lower than their minimum size.

### ListFigure, AttributeListFigure & MethodListFigure

These are two figure differ only by the explanatory label. And that is why they derive from the same class: ListFigure.

Class ListFigure extends the `org.eclipse.draw2d.Figure` class.

- Layout: Its layout manager is adjusted in ToolbarLayout. ToolbarLayout is to arrange figures in a single row or column. Orientation can be set to either a row or column layout. It aligns the children in the parent’s client area by compressing the children by some amount, but not smaller than their minimum size.
- `getMinimumSize`: The method `getMinimumSize` is redefined, just like the one in AttributeFigure.
- ListFigureBorder: The ListFigure provides an edge extending the LabeledBorder class so as to be able to write “attributes:” or “methods:” according to the situation. The
output of the border just looks like the one in Figure 5.5.

We create the ListFigureBorder class extends AbstractLabeledBorder to define the edge of the ListFigure’s border. And the final effect can be seen in the left picture.

The method `paint(IFigure, Graphics, Insets)` is to introduce graphics, while method `calculateInsets(IFigure)` is to define the place taken by the edge so that the figure host can adjust his client area correctly and not to exceed on the edge while being aligned and while aligning his children.

SubjectFigure
We create the SubjectFigure class to represent the class corresponding to class model. The subjectFigure contains a `label` to represent its class name and a `contentPane` to add the lists of attributes and methods of the class. And the contentPane is accessible by the method `getContentPane()` in SubjectFigure.

- **Layout & ContentPane**
  We create a ContentPane class in the same file to produce a contentPane. And ToolbarLayout, who manages the layout of SubjectFigure, also governs ContentPane.

- **getPreferredSize**
  The method `getPreferredSize(int, int)` is redefined so that the preferred width never exceeds the width of the SubjectFigure.

- **ContentFigureBorder**

We also create a ContentFigureBorder class extends org.eclipse.draw2d.AbstractBorder in the same file as ContentPane to set its border. This border simply paints a line 1 pixel in thickness across its top, to serve as the separator between the label, which represents the class name, and the contentPane. We can see the example of the result in Figure 5.6.

5.2.4. PartFactory[12]
With the help of PartFactory, we can associate the model and its controller. There are only two steps to implement the link.

a) Create the right controller according to the given model.
b) Use `setModel()` to connect the model and its controller.

### 5.2.5 GraphicalViewer

Now, it is time to create a view of the class diagram. Usually, the view is displayed in “EditPart”, but in our project, we need to show it in “ViewPart”. The difference between EditPart and ViewPart will be covered later.

Because we choose ViewPart as our diagram panel, most of the configuration will be done in the method of `void createPartControl(Composite parent)`.

```java
// Viewpart
graphicalViewer = new ScrollingGraphicalViewer();
canvas = (FigureCanvas) graphicalViewer.createControl(parent);
ScalableRootEditPart root = new ScalableRootEditPart();
graphicalViewer.setRootEditPart(root);
graphicalViewer.setEditDomain(new EditDomain());
graphicalViewer.setEditPartFactory(new PartFactory());
```

**Figure 5.7 createPartControl example**

As we can see from the code, Configure the GraphicalViewer includes choosing the suitable RootEditPart and EditPartFactory. ScalableRootEditPart enable us to zoom and scroll.

Afterwards, we need to used `setContents()` method in `showDiagram()` method. Of course, they are both created in the ViewPart class. In our project, the content comes from the memory, which will be mentioned soon.

The last important thing is to set an `EditDomain` to store the Command Stack in GEF, which is also used to notify the SWT events generated in GraphicalViewer.

### 5.2.6 Request, Command and Policy[12]

Now we will add the drag action for the class figure of a class diagram. This will refer to the Request, Command and Policy in GEF. When a model is modified, it will generate the Requests (org.eclipse.gef.Request). Then the Request will be sent to the EditPart corresponding to the model. The related Editing Policy should have been installed in the EditPart. The Editing Policy is used to manage some Command, which will perform these requests.

**Figure 5.8 Processing Actions Diagram**

- First, we create a CustomXYLayoutEditPolicy in `umlclassdiagram.policies` package extending org.eclipse.gef.editpolicies.XYLayoutEditPolicy.
- Next, we will install the policy in DiagramPart. This can be achieved in the `createEditPolicies()` method in DiagramPart.
Third, we need to create Command. We will create a ChangeConstraintCommand class in `umlclassdiagram.commands` package to change the constraint.

Afterwards, this command will be called by Editing Policy using `createChangeConstraintCommand(EditPart child, Object constraint)` method.

At last, we need to listen to the model changes and notify to refresh the view. So we add the `firePropertyChange()` method in the `setSize()` and `setLocation()` method in Node model to fire the changes of the Subject model.

### 5.2.7. Action & ActionGroup

After the drag action, we will add **zoom in/out** action. This will involve designing an ActionGroup extending `org.eclipse.ui.actions.ActionGroup`.

![Figure 5.9 Action and Action Group Diagram](image)

First, create the Constructor, and send in the necessary parameter.

Then, set the ToolbarManager in method `fillActionBars(IActionBars actionBars)`. Add three actions and one zoom combo box in the toolbar manager.

At last, add some code in `createPartControl(Composite parent)` to use ActionGroup and ToolbarManager.

### 5.2.8. Layout

Now we will design a easy layout for the Model in GEF. As we have already known, there are two parameters in abstract class Node: **position** and **size**. There are in turn responsible for the location and dimension in the model. We will design a layout by mainly changing these two valuables.
Situation 1: When there are no more than nine nodes in the Model, we can use the layout solution 1.

The squares with numbers in the center present the position of the classes. There numbers mean the sequence in which the class node will be set up. And especially, the node in the center will be the class which is currently being activated.

Situation 2: When there are more than nine nodes in the model, we will apply the second solution.

First we will count the number of the nodes, then minus it by eight. Next, we will divide it by two. Finally, we will decompose the answer into form of \((x+1)/x\) by \(x\), while \((x+1)/x + \(x\) = \(ans\).

For example, if we received 11 nodes, let us calculate according to the process introduced above. We first get 3 by minus it by eight, then we get 1 by divide it by 2. At last, we will receive the answer: 1 by 0. So the final size would be 4 by 3 (plus both numbers by three). The output would look like Figure 29(a).

Let us take “13” as another example: we can obtain these answers by the process: 13 => 5 => 2 => 1 by 1 => 4 by 4. The output would look like Figure 29(b).

And the numbers in the squares also presents the sequence of the nodes, while the first node is the currently being activated class.

The biggest problem of this algorithm is that it does not consider connection. So as you...
can see from Figure 30, the connections intersect each other.

But considering the time, we have to save our efforts on the layout algorithm, but at the end of the project, we can call a layout algorithm function designed in another project, and it would be much better.

5.2.9. Visual Effect Improvement
Because we may come into some relative complex diagram sometimes, so it is very necessary for us to improve the visual effort of the diagram. There are three tasks that can lead to a good output.

- **Connection**
  As we have defined in Model, there are several kinds of connection, such as composition, association, or inherency. And it would be more obvious if we add color to different type of connections. This task will be implemented by adding some codes in the Figure part of GEF:
  After creating these lines representing connection, we will set color to these lines.
  
  It would be better that if we can solve the problem of crossing lines.

- **Transparency**
  In the layout designing part, we have set the most important class in the center of diagram. In order to intensify the effect, we would also set the class nodes around the center one into semi-transparent.

  Because some nodes would be opaque while others being transparent and we do not want to involve in extra models, we would change the constructor of the Figures in GEF and add a judgment that if the class node is in the first position of the list. If it is, we do no change anything, which will keep it opaque. Otherwise, we will extend “Label”, and create “MyLabel” to set the nodes’ titles into transparent. The method about the transparency is `setAlpha()`. Some more detailed information about transparency can be found in Appendix C.

- **Center Class**
  We have been always talking about set the first class in the center of the diagram. But actually, we may run into the situation that the first class in the list is not the one being currently activated after the list is created. So we need to add some patch to fix it.

  We can iterate the list and change the position of the clicked class when we find it by comparing the node’s names to the one which is the activated. Afterwards, we can do the following operation to the list.

5.3. JDT
Up to now, we have already built up a whole model for the Class Diagram. Now, let us discuss how JDT get the information to be displayed.

We will use JDT to analyze the memory in method `showSelection(IWorkbenchPart sourcepart, ISelection selection)`.

First, we will identify if the “sourcepart” is an instance of editor, that is to say, if we are clicking on an editor or a viewer.

In eclipse, a workbench part is a visual component within a workbench page, and there are two subtypes: **view** and **editor**.

- **View**: A view is typically used to navigate a hierarchy of information, open an editor or display properties for the active editor. As in our program, we use view to show the Class Diagram.
Editor: An editor is typically used to edit or browse a document or input object.

If it is, we will collect the code in this editor using `getEditorInputJavaElement` and save it in `JavaElement`. Then, we will compare and decide if its kind is Java compilation unit.

If it is again, we create an ASTParser, it is a Java language parser for creating abstract syntax trees. We set the `JavaElement` as its content, and start to parse using `createAST(null)`.

After the parsing, we get a compilation unit AST result, now, we use a visitor to walk the AST result, and save the list of current and closely related compilations units.

Next, we will use some function to change the compilation units into a new form: `GraphInterface`, which can be identified by `VizzAnalyzer`. After that, we will try to get the graph information saved in this parameter and build up a GEF model so that we can display it in the viewer.

5.4. ViewPart

As introduced before, a View is a visual component within a workbench page. In our project, we use view to show the picture.

createPartControl

The most important method in the ViewPart Class is `createPartControl(Composite parent)`. We write all the initialization and configuration here.

In our case, we add a listener to monitor the user’s action on the workbench. Once there are some action on the editor, the listener will be triggered, and send current Class Diagram to the Viewer.

Next, we setup the GraphicViewer, to make it work in the Viewer.

At last, we add three different zoom actions and one zoom combo box in the toolbar of the viewer.

Listener

We will add a Listener to the viewer so that once there is some action on the workspace, the viewer will be informed and try to scan if it is the active of a new class. Especially, all the scan and refresh operations will be set here.

Dispose

It is also very important for us to unregister our listener when the view is closed. So we need to use `removeSelectionListener(listener)` method.
As we can see from the figure above, this is the final result of our work. The diagram viewer is on the right side while the editor is in the center of the Eclipse IDE. Once we click on the new class file in the Package Viewer on the left side, the program will exam the memory and refresh the viewer panel to show a new class diagram.
6. Conclusion and Future work

In this chapter, we will summarize the thesis. It shows how the problem has been solved and proposes some future work.

6.1. Conclusions

The problem to be solved by this thesis was to implement a viewer plug-in for the Eclipse IDE resembling a UML class diagram based on the current position in code. The solution for the problem is based on the existing Eclipse plug-in: JDT and GEF. They provide the API for extracting necessary information from memory and viewing the class information in the memory as UML class diagram by the MVC architecture of the GEF framework.

We have described the goals and criteria for the problem in section 1.4. And we described final results in section 5.4.

The first goal was to have a general idea of UML class diagram. We discussed the related theory in detail in Chapter 2.1.

The second goal was to figure out how to receive the data in the memory and return a model back. Additionally, we understood the GEF framework, and how the MVC was built up. We documented our findings in Chapter 5.2.

The third goal was to improve the output by adding a simple layout so that we can see the outcome more clearly. This goal depended on the success of the second goal, and we implemented the simple layout as described in Chapter 5.2.8.

The fourth goal was to pick up live program information from the memory. We solved this problem with the help of JDT. We saved the memory information into one graph parameter which can be identified by VizzAnalyzer and then call a function which already exist in the old project and then create the GEF model according to the graph, as described in Chapter 5.3.

The fifth goal was to refactor the project to use ViewPart instead of EditPart. Because we want to see the diagram while we are editing code, so it is reasonable to move the viewer on the right side of the editor. It may cause some little difference from the original project, so we needed to do some adaption work such as adjust the zoom in/out action into a action group and reset the tool bar into the viewer. We discussed our efforts in Chapter 5.2.5 and Chapter 5.4.

The last goal was improvement. In order to make the final output more clearly, we changed the color of the connections by their different kinds. Then we set the class node in the center opaque and the nodes around it semi-transparent, as described in Section 5.2.9.

6.2. Future work

The Quick Orientation View has almost finished its basic purpose. But we can keep on working on the project and make some improvement to make it better.

The layout is very simple in the project. The best way to solve this problem is to cooperate with another student. His project is to design a layout for the VizzAnalyzer graph. Actually, we once tried to combine these two projects, but it seems not very successful. The key problem is that we have different VizzAnalyzer graph file as input. The information in the file is different, so we cannot just call the function with out any modification. And considering the time, we have to compromise and cheat the program by changing the file manually so that the system will work. At least, we have proved, once the file content is identical, these two projects can cooperate pretty
Another work would be some further improvement on the diagram. The size of node should be able to be adjusted automatically according to the length in the content. Additionally, we can add some commands and policies to respond the double-clicking on the node and open the relating files in the editor.

Apart from that, we also need to solve the redundancy connection problem in the memory. The information we pick up from the system may be a little bit different from what it is supposed to have. We may find an extra relationship between two classes and that may lead to some misunderstanding. This may be caused by the function we used in the old project.
Reference

4. The UML Class Diagram http://www.developer.com/design/article.php/2206791 (062308)
6. GEF Project Overview http://www.eclipse.org/gef/overview.html (062308)
7. Eclipse overview http://www.jdg2e.com/ch08.architecture/doc/ (062308)
12. Yun Liu, UML 2.0 with VizzAnalyzer – Visualization of class diagrams, MSI Växjö University, Sep, 2007
Appendices

Appendix A Installation

In this appendix, we will discuss the installation of our plug-in project.

Because Eclipse will scan its file directory in the initialization, we need to export our project as “jar”. Then copy it to the place where saves all the plug-ins. So our project will be loaded then.

➢ Choose: File => Export…

![Figure Appendix A 1 Installation Instruction 1]

➢ Choose “JAR file”, then click on “Next”

![Figure Appendix A 2 Installation Instruction 2]

➢ Input the file path and file name, then choose “Finish”.

38
At last, copy the exported jar file into eclipse’s home directory: *eclipse\plugins*.

After all the steps done, please do not forget to restart Eclipse platform in order to load in our project during its startup.
Appendix B Detailed Referenced Introduction

In this chapter, we will list some description quoted from other document, in order to improve the comprehension of the thesis.

Appendix B-1 Eclipse Components
This table comes from [5].

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform runtime</strong></td>
<td>Defines the extension point and plug-in model. It dynamically discovers plug-ins and maintains information about the plug-ins and their extension points in a platform registry. Plug-ins are started up when required according to user operation of the platform. The runtime is implemented using the OSGi framework.</td>
</tr>
<tr>
<td><strong>Resource management (workspace)</strong></td>
<td>Defines API for creating and managing resources (projects, files, and folders) that are produced by tools and kept in the file system.</td>
</tr>
<tr>
<td><strong>Workbench UI</strong></td>
<td>Implements the user cockpit for navigating the platform. It defines extension points for adding UI components such as views or menu actions. It supplies additional toolkits (JFace and SWT) for building user interfaces. The UI services are structured so that a subset of the UI plug-ins can be used to build rich client applications that are independent of the resource management and workspace model. IDE-centric plug-ins define additional function for navigating and manipulating resources.</td>
</tr>
<tr>
<td><strong>Help system</strong></td>
<td>Defines extension points for plug-ins to provide help or other documentation as browsable books.</td>
</tr>
<tr>
<td><strong>Team support</strong></td>
<td>Defines a team programming model for managing and versioning resources.</td>
</tr>
<tr>
<td><strong>Debug support</strong></td>
<td>Defines a language independent debug model and UI classes for building debuggers and launchers.</td>
</tr>
<tr>
<td><strong>Other utilities</strong></td>
<td>Other utility plug-ins supply function such as searching and comparing resources, performing builds using XML configuration files, and dynamically updating the platform from a server.</td>
</tr>
</tbody>
</table>

Table Appendix B 1 Eclipse Platform

Appendix B-2 Lightweight System
A lightweight system is a graphics system that is hosted inside a single heavyweight control. The advantage of lightweight systems is that they are much more flexible than the native windowing system, which is generally composed of rectangular components. They allow you to create and manipulate arbitrarily shaped graphics objects. Because they simulate a heavyweight graphics system within a single heavyweight window, they allow you to create a graphically complex display without consuming a lot of system resources.[8]
Appendix C Related Works

In this chapter, we will have an overview of the Rich Client Platform (RCP).

Eclipse RCP helps us developing desktop application efficiently. The developing process of RCP is almost the same as Eclipse plug-in. The only little difference would be the way we extend the extension points. And the application has a similar GUI with Eclipse platform, besides it may not have as many toolbars or menus as Eclipse.

The Figure below shows our testing RCP project, which is also a diagram editing program using GEF and Draw2D. You can see the same “model”, “parts”, “commands” and “policies” package on the left Package Explorer. And the theory and implement are almost the same as our thesis project.

![Figure Appendix C 1 RCP Graphical Editing Example](image)

Next, let us look at the testing RCP application. We can see a program with two menus on the top, and one editor in the frame in Figure 34. The editor has a palate which can allow us to draw new nodes and new connections on the diagram. In the editing area, we can see three rectangles which all set to semi-transparent.

This example shows the difference between one plug-in project and a RCP project, while they both use GEF. Compared to plug-in, which has to run within Eclipse, RCP is mainly used for desktop application.
Figure Appendix C 2 RCP Transparency Exhibition