

Use-Case Diagrammer

System Design



June 17, 2002
Sasmito Adibowo

Arcle Technologies
<http://www.arcle.com>



Table of Contents

About This Document	3
General Description	3
Product Functions	3
Design Goals	3
General Constraints	3
Domain Environment	4
User Interface Design	5
User Interface Flow	5
Screens and Dialogs Design	5
Object-oriented Design	7
Overview of the Subsystems	7
UML problem domain classes	8
Drawing Objects Classes	11
User Interface Classes	14
Code Samples	19
DrawingObject composites	19
Mouse logic implementation	20
Mouse singleton states	22
Stereotype renderings	23
Relationship adapter	24
Setting up the window	26
Conclusion	27



1 About This Document

This document is the specification for the application program entitled *Use-Case Diagrammer*. It serves as a blueprint that will be used to build the initial prototype of the system. After completion of the prototype, it is expected that this document will be revised to improve its details further.

The third revision of the document updates the specification dated on May 26, 2002. The scope of the design has been re-expanded to also include behavioral design patterns.

2 General Description

2.1 Product Functions

Use-Case Diagrammer is an end-user application software for the interactive generation and manipulation of use-case diagrams. The system is equipped with a graphical user interface that allows intuitive manipulation of the objects in the diagram.

In the future, this product is expected to be evolved into a full-fledged UML diagramming application. Development branches of the system may also include general-purpose diagramming packages.

2.2 Design Goals

The following are the objectives to be accomplished by the design of the system:

- Demonstrate the applications of the *Gang of Four* (Gamma, Helm, Johnson, and Vlissides) design patterns.
- Able to be extended into a full-fledged UML drawing package.
- Provide reusable components to create any diagramming software.
- When implemented, the application will be sophisticated enough to prove usable to end-users.

2.3 General Constraints

Constraints are limitations imposed to the design, and will persist into the implementation of the system. Listed below are the restrictions placed on the system:

- Built on the Java programming language.
- Uses Swing GUI components.
- Incorporate creational design patterns.
- Incorporate structural design patterns.
- Incorporate behavioral design patterns.



3 Domain Environment

This section outlines the problem domain environment that the system will operate on. It also serves to define the scope of the system with respect to its logical operational environment.

Figure 1 depicts the significant problem domain entities within the scope of Use-Case Diagrammer. Since the main purpose of the system is to draw use-case diagrams, the UML elements involved are only those needed to describe such diagrams.

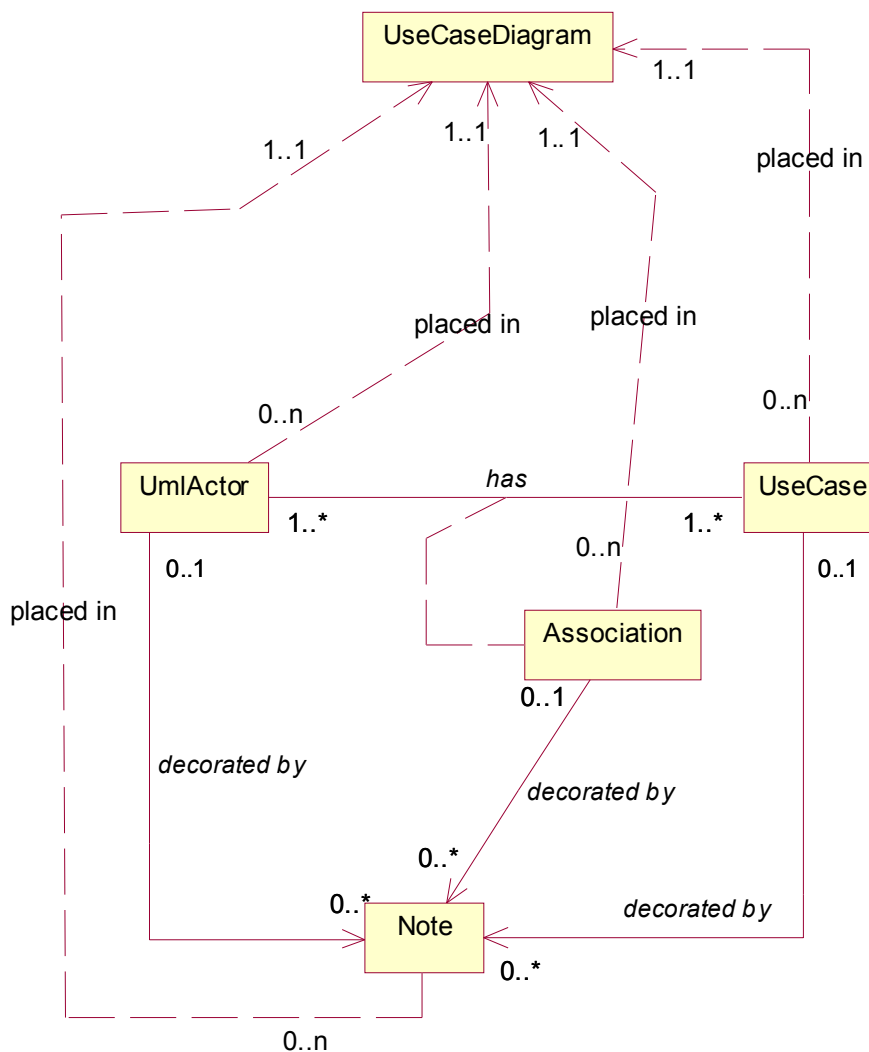


Figure 1 Problem-domain entities.



All of the UML elements are placed in a use-case diagram. For simplicity, the system only accommodates one view per diagram. Although in reality, the same UML element may be placed in several views. Each use-case diagram may only contain *actors*, *use-cases*, *notes*, and *associations* among the various elements.

A use-case may only be connected to an actor, not to other use-cases. Likewise, actors may not connect to other actors. The connector between an actor and a use-case is called an *association*.

Each element in the diagram may optionally be attached to one or more *notes*. The note contains descriptive text that may provide verbose explanation of the various aspects of a diagram. A note element may also be stand-alone, not being connected to any other elements.

4 User Interface Design

The user interface design describes the master plan of the system's user interface. It also includes screen designs for major user interface elements.

4.1 User Interface Flow

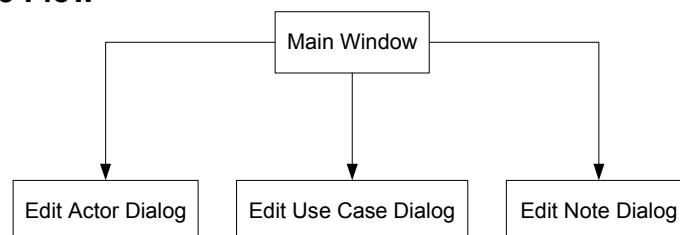


Figure 2 User interface flow (without returning arrows).

The flow between dialogs and windows is kept simple. Initially, the system displays the main window. Using this window, the user may add and move the drawing objects. Modification of drawing objects is done through a modal dialog, brought up when the user double-clicks the corresponding object. Each dialog is specific to the type of object being edited, but has the same superclass to help user interface consistency and reuse. When the user has completed editing, he or she dismisses it and returns to the main window. At this point, the main window repaints its drawing area to reflect the changes just made.

4.2 Screens and Dialogs Design

This section provides the intended screen shots of Use-Case Diagrammer. Because of differences in the tools used in design and those to implement the system, the actual style of the user interface elements may vary from those in the design. Furthermore, only significant elements are included with the



design. This allows further improvements to the user interface without major changes in the design.

4.2.1 Main Window

The **main window** is the primary work area of the user. It contains a drawing window, tool bar, status bar, and a menu bar.

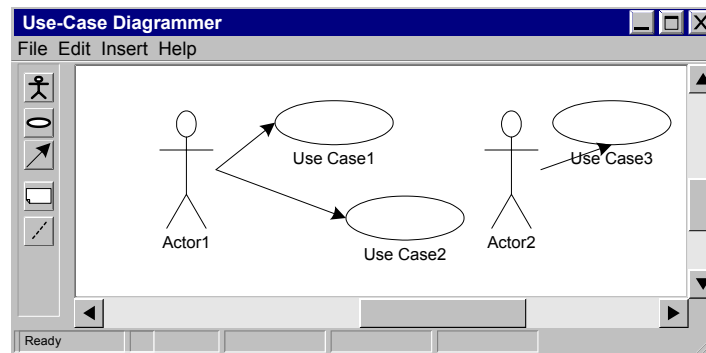


Figure 3 The main window

Occupies the largest area in the main window, the **drawing window** display objects of the current drawing. This window is used to add, move, resize, and delete drawing objects. When the user double-clicks an object in this window, it displays the corresponding editor dialog for the object. This window is equipped with scroll bars used when the drawing size exceeds the display size.

A **tool bar** is also provided in the main window to help the user to add objects in the diagram. When the user wants to add an object, he or she first clicks on a button in the toolbar, and then clicks on an unoccupied area in the drawing window. Afterwards, the desired type of drawing object will appear in that area with its attributes initialized to the default.

At the bottom of the main window, the **status bar** displays the current state of the system. It is also used to display simple help texts that provide hints to the user on what does the highlighted command does.

The **menu bar** stores the commands available while working in the main window. It provides file-operation and object editing commands, and a command to access the on-line help system. The commands in the menu bar are a superset of those that are available in the tool bar.

4.2.2 Edit dialogs

The editor dialogs of the drawing objects are similar to each other. Each provides text fields to modify the name of the object, its stereotype and a large text area to enter extended documentation for the object. At the top-right of each dialog, an icon is provided to remind the user of the type of object being edited.

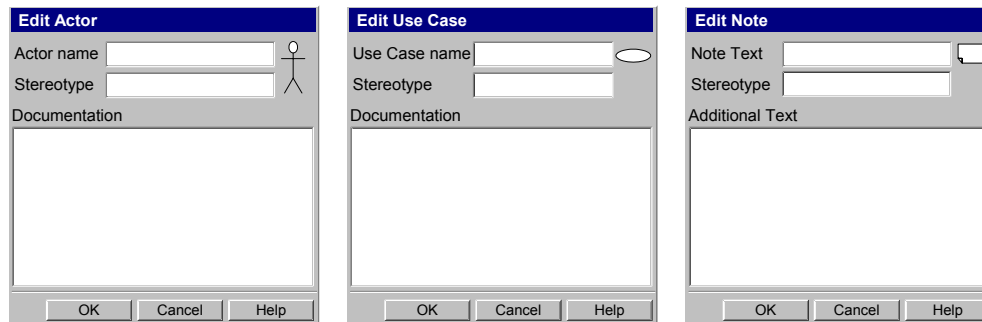


Figure 4 Drawing object editor dialogs.

5 Object-oriented Design

This section provides a description on the architecture of the system. It depicts significant static structures and elements that will be placed in the system. An object-oriented approach is practiced in the system's design, using UML diagrams where applicable.

5.1 Overview of the Subsystems

The diagram in **Figure 5** shows that the UML problem domain classes and the generic diagramming classes provide the building blocks that the system is to be built on. The use-case diagrammer classes either inherits from, depends on, or composed of (in some parts) the classes from those building blocks.

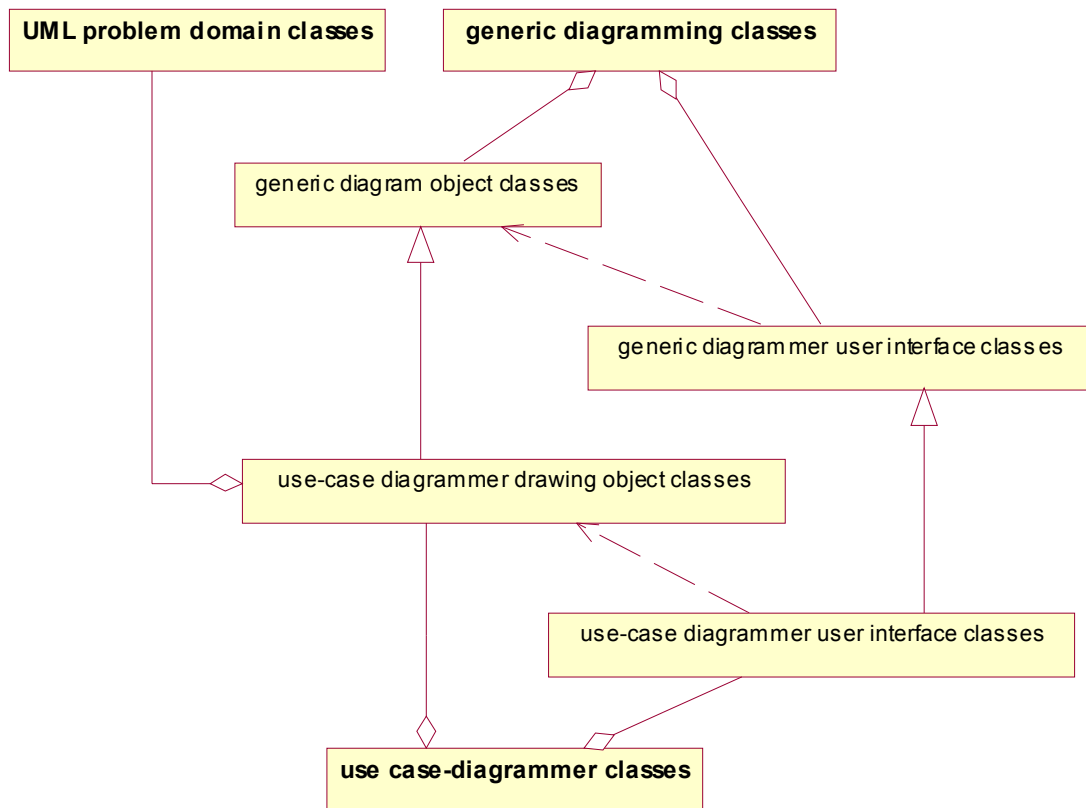


Figure 5 Subsystems overview

5.2 UML problem domain classes

Figure 6 depicts the classes which captures information on the entities in the problem domain. This collection of classes is deliberately made large and nearly complete, to span the problem domain of UML diagrams better.

Although not all of the classes are used in the system, the comprehensiveness will simplify evolution of the system into a full-fledged UML diagramming software.

Every *element* in UML is represented by an object of `UmlElement` class. This is the base class for more specific elements in UML. As UML provides extensibility mechanisms through *stereotyping*, *tagged values*, and *constraints*, the support classes include `UmlStereotype`, `UmlTagValue`, and `UmlConstraint`, respectively.

Elements that connect UML *things* (described shortly) are called *relationships*. This is represented through `UmlRelationship`. The relationship has four specializations: *association*, *dependency*, *generalization*, and *realization*. The support classes for those specializations are named similarly.



UML *things* are abstractions that are first-class citizens in a model. The support classes represent it as `UmlThing`. A thing may be specialized into *annotational things*, *behavioral things*, *grouping things*, and *structural things*. These specializations are represented by classes of similar names. Only a small subset of structural things is in the scope of Use-Case Diagrammer, those are use-cases and actors.

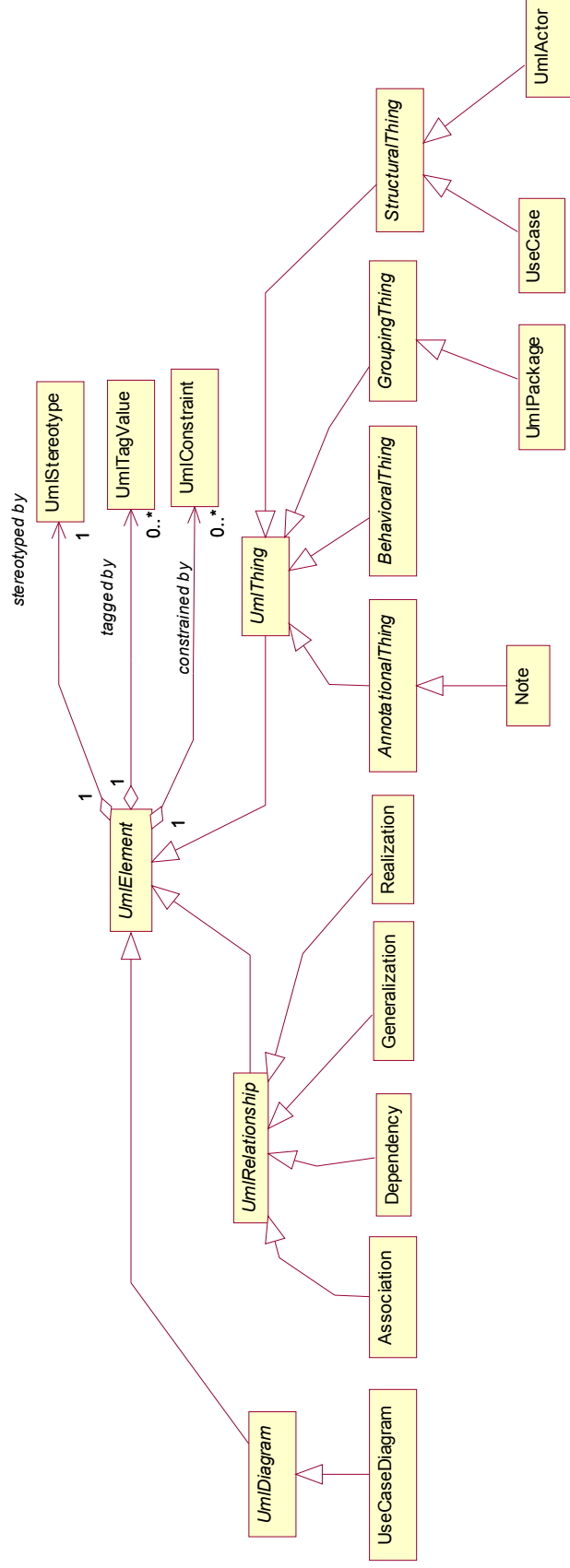


Figure 6 UML diagram problem domain support classes



5.3 Drawing Objects Classes

The drawing objects are the creatures placed on the drawing area. They control the objects' appearance and how they are logically manipulated. The user interface classes depend on these classes to provide object-based drawing functionalities to the user.

5.3.1 Generic diagram object classes

These classes provide support for creating block-connector-block type diagrams. The classes and their interconnections are illustrated in ?.

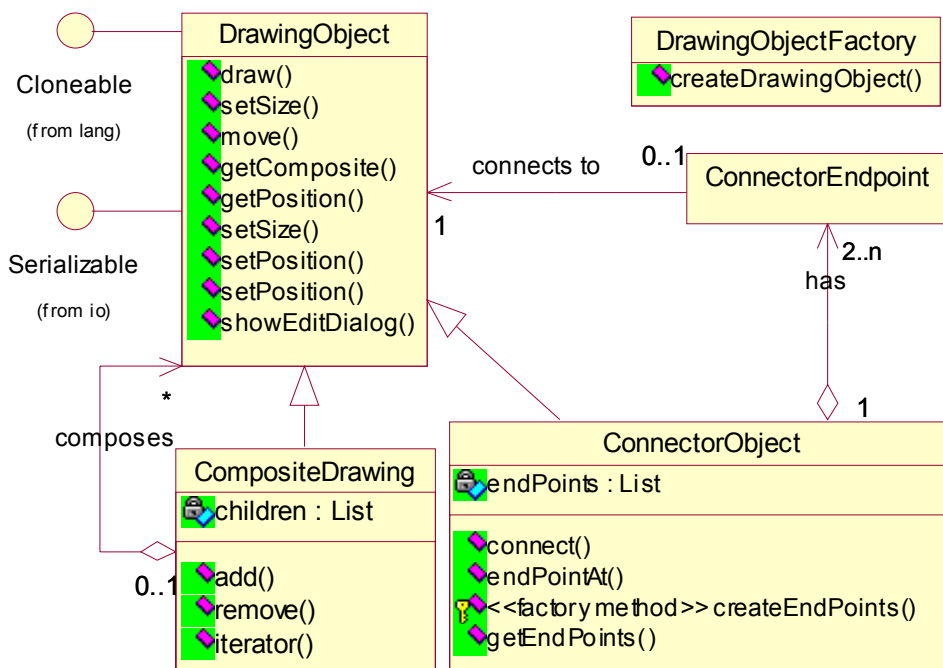


Figure 7 Drawing Object classes

Class DrawingObject represents the base class for all draw-able objects in the work area. It is in a sense a *pseudo-abstract* class – it is not declared abstract, but it does nothing much on its own. Objects of DrawingObject are to be created by subclasses of DrawingObjectFactory.

The CompositeDrawing class provides mechanisms to group other drawing objects. It does so by employing the *composite* design pattern.

A ConnectorObject connects two or more drawing objects. It owns at least two ConnectorEndpoint objects that each is to be attached to a DrawingObject.

5.3.1.1 Implementing Undo

The DrawingObject class implements the Serializable interface to allow it to be saved into a temporary medium. This application of the *memento* pattern



is intended primarily to allow *undo* functionality. Before the application of an action to an object, the object's state is saved into a temporary buffer. When the user later decides to undone the operation, the object is loaded back from the buffer.

Single-level undo may easily be implemented by saving the current state of the diagram (consisting of the states of all objects) before any command that produces side effects are executed. While multi-level undo might be implemented by keeping a list of commands, along with the previous states of the objects they modify.

5.3.2 Use-case diagrammer drawing object classes

Since UML diagrams fit into the block-connector-block paradigm, these classes extend from `DrawingObject` to obtain the functionality of its supporting classes. The current implementation is limited only to support use-case diagrams, but they are structured such that it will ease further development that supports the creation of other UML diagrams.

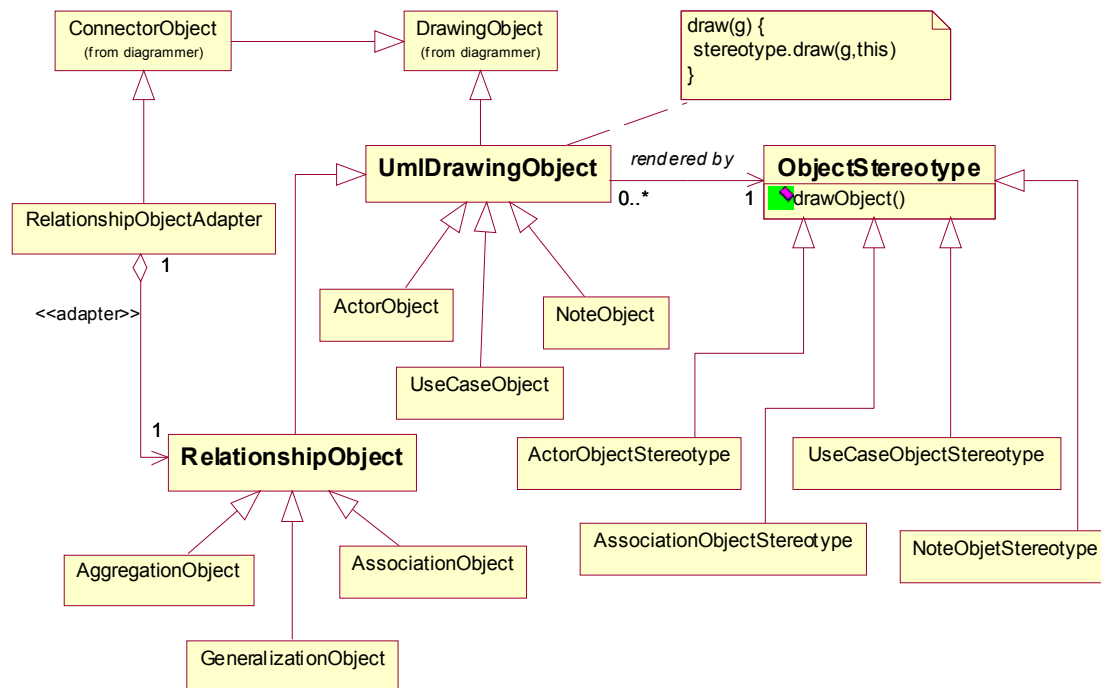


Figure 8 Use-case diagrammer drawing objects

Each instance of `UmlDrawingObject` delegates its screen rendering to its corresponding stereotype strategy object (represented by the `ObjectStereotype` class). The strategy allows nonstandard – and even user-defined – renderings that are based on the object's stereotype. This application of the



strategy pattern is used in conforming the UML standard that allows non-standard renderings through stereotyping.

Implementation of `RelationshipObject` is rather unique. Derivation from the problem domain requires that this class be both a `ConnectorObject` and a `UmlDrawingObject`. A problem arises since Java does not support multiple inheritance. The design works around this problem by composing `RelationshipObject` into `RelationshipObjectAdapter`. The latter class *adapts* the former by forwarding (most of) its method calls.

5.3.2.1 Factory classes

These classes create instances of the drawing objects. All of the factories descend from `UmlDrawingObjectFactory`. This is to allow the creation of UML drawing elements stereotyped differently from the default.

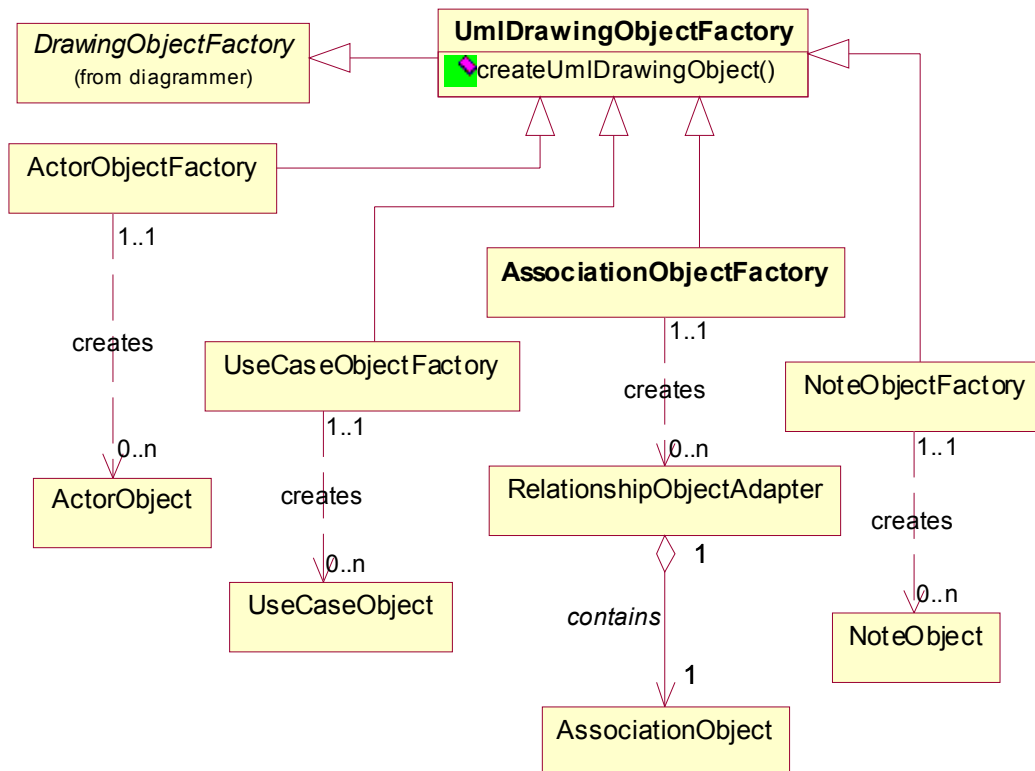


Figure 9 Factory classes

Since the implementation of `RelationshipObject` is uncommon, `AssociationObjectFactory` does not directly return instances of `AssociationObject` (a subclass of `RelationshipObject`). Instead, it creates and returns a `RelationshipObjectAdapter` that contains the newly created `AssociationObject` packaged in it.

5.3.2.2 UML realizations

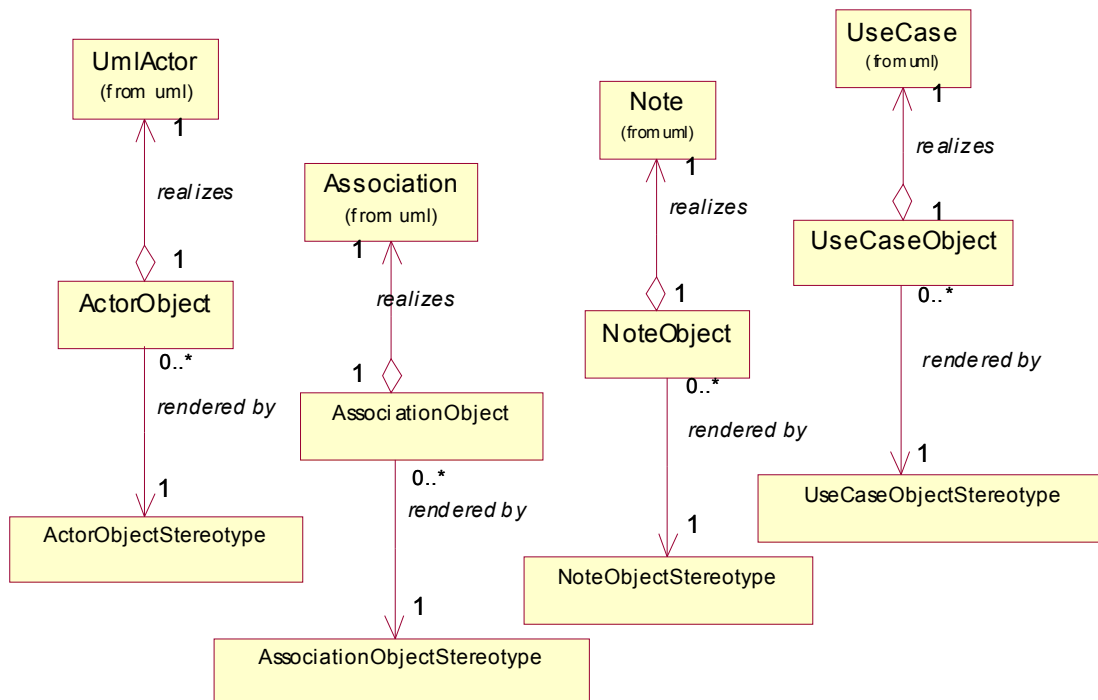


Figure 10 UML realizations by the drawing objects

Each `UmlDrawingObject` instance realizes the properties of a UML element by owning an instance of the proper subclass of `UmlElement`. Apart from attaining flexibility, this approach is taken because Java does not support multiple inheritance.

5.4 User Interface Classes

This section describes the group of classes that manages the user interface. These classes are further clustered into two groups: generic diagrammer classes and use-case diagrammer classes.

5.4.1 Generic Diagrammer user interface classes

These classes are to be used as base classes for the primary user-interface classes of a vector-drawing application. It contains standard elements expected from such software: The main canvas, a tool bar, and a menu bar. Additionally, it also provides a base class for defining dialogs used to modify the drawing objects. An overview of the classes is provided in **Figure 11**.

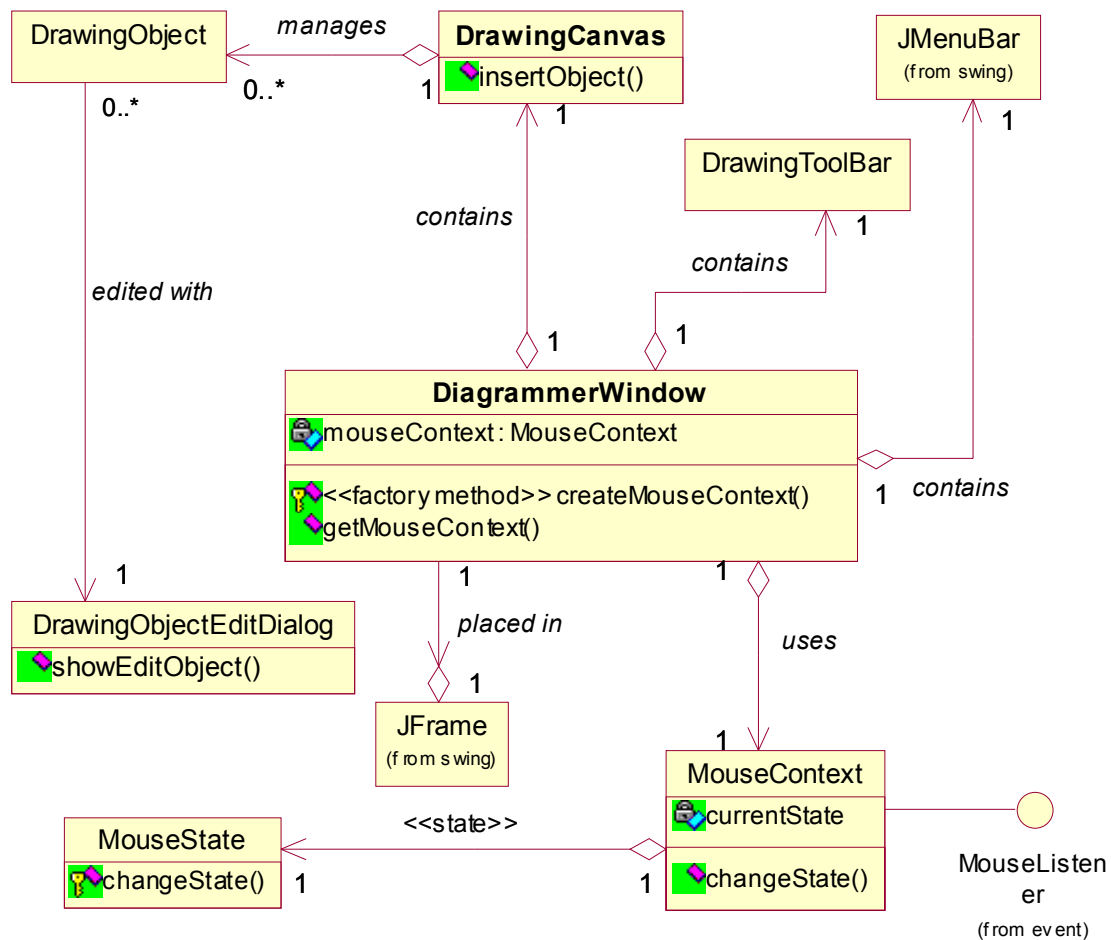


Figure 11 Generic diagrammer user interface classes

Since mouse operations in a drawing application is very context-sensitive – possibly translates into many states in the drawing window – the *DiagrammerWindow* class *delegates* its mouse operations to a *MouseContext* instance (most likely be a subclass obtained through the *createMouseContext()* *factory method*). This object employs the *state* design pattern – further forwards its method calls to instances of *MouseState*, which particular instance is selected depending on the current state.

5.4.1.1 User interface hierarchy

Figure 12 shows the hierarchy of the generic diagrammer user interface classes. Because it is meant only to provide drawing functionalities, the *DrawingCanvas* is derived from *JComponent*, thus it may not contain other components. While the *DiagrammerWindow* is derived from *JPanel* so that it is

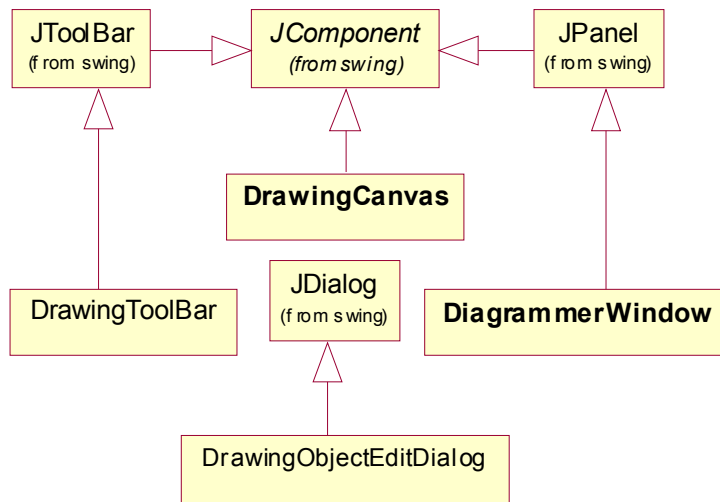


Figure 12 Generic diagramming user interface classes hierarchy

able to contain the tool bar, the status bar, and the menu bar. This derivation makes it possible for `DiagrammingWindow` to also be contained in other panels, making it possible to simplify applications such as adapting it into an applet. This flexibility will be lost if the class is derived from `JFrame`.

To provide scrolling functionality, `DrawingCanvas` is placed inside a `JScrollPane` object, which in turn placed in the `DiagrammingWindow`. This is an application of the *decorator* pattern, which case the `JScrollPane` object decorates the `DrawingCanvas` object with scroll bars.

5.4.2 Use-Case Diagrammer classes

As specified by its superclass, the diagrammer window employs the *state* design pattern. It does so through an instance of `UmlDiagrammerMouseContext`. The context then uses singleton classes to implement its various states. The diagrammer window, the context class and its various state classes are shown in **Figure 13**.

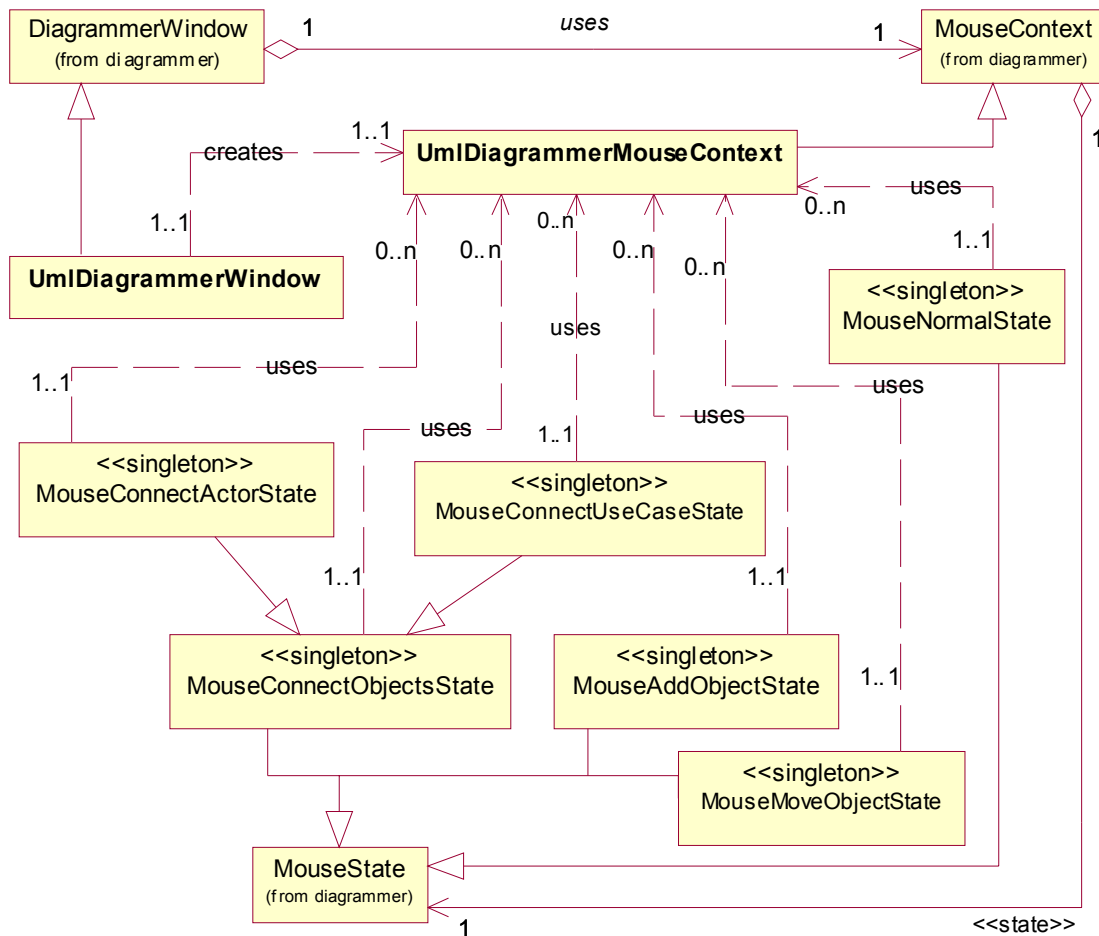


Figure 13 Use-case diagrammer user interface classes

5.4.2.1 Mouse States

The state diagram of **UmDiagrammerMouseContext** is depicted in **Figure 14**.

Each event that causes a state transition is either a mouse event (captured by implementing the **MouseListener** interface) or an event internally-generated by the system itself (most likely be a consequence of a user command).

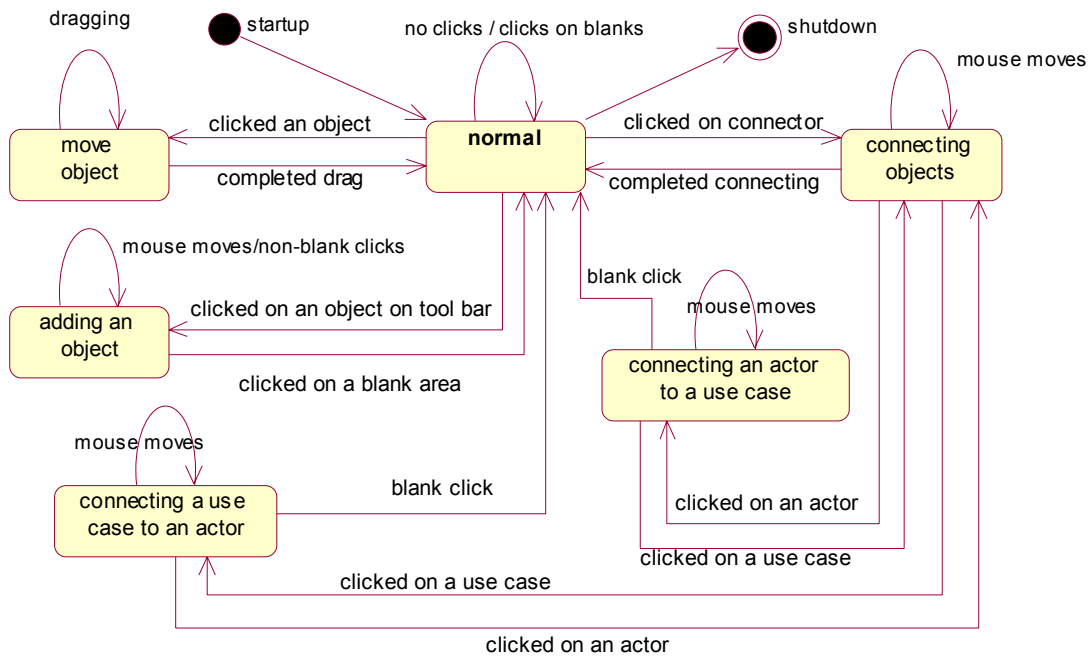


Figure 14 UmlDiagrammerMouseContext state diagram

5.4.2.2 State Classes

Each state in **Figure 14** is implemented by a singleton class. These classes do the real work for `UmlDiagrammerMouseContext`. The mappings among the state names (of **Figure 14**) and its implementing state classes (shown in **Figure 13**) are shown in the table below.

State Name	State Class
normal	MouseNormalState
move object	MouseMoveObjectState
adding an object (either an actor or a use case)	MouseAddObjectState
connecting objects	MouseConnectObjectsState
connecting a use case to an actor	MouseConnectUseCaseState
connecting an actor to a use case	MouseConnectActorState

5.4.2.3 Diagram object editors

These classes implement the editor dialogs specified in section 4.2.2. Because of their inherent similarity, they are all derived from `UmlDrawingObject-`



EditDialog. The common superclass then, in turn, derived from DrawingObjectEditDialog. Their inheritance hierarchy is shown in **Figure 15**.

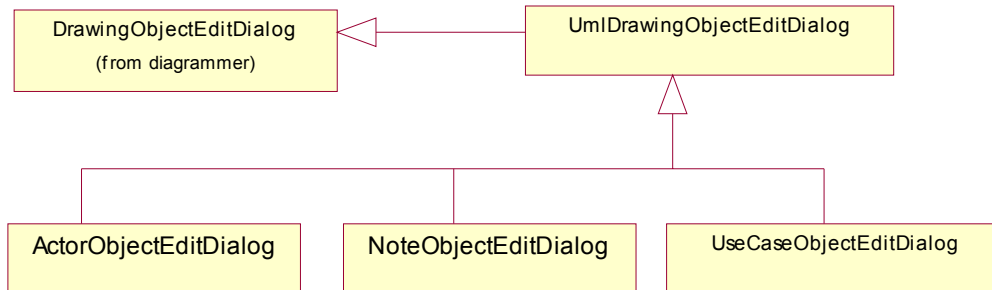


Figure 15 Diagram object editor dialog classes

6 Code Samples

The following are several examples of the source code that may reside in the system. They describe how the design patterns used will map into code. These codes are intended for illustrative purposes only, and not as functional code fragments.

6.1 DrawingObject composites

All DrawingObject subclasses must define the common methods to manipulate the object. They could be moved, resized, and painted without the need to know whether the object in question is composite or not.

When the need to manipulate composite-objects arises, the `getComposite` method will be called on the object. When it is a composite, the object will return the proper instance of `CompositeDrawing` (or its subclass) which manages the composition. If the object is simple (i.e., a single object), the method will return `null`.

6.1.1 The simple case

At the simple case, the DrawingObject subclass is a single object; it does not consist of other DrawingObject instances. Therefore, its `getComposite` method will return `null`. The default behavior is already defined in its DrawingObject superclass.

```
package arcle.diagrammer;
// ...
public class DrawingObject implements Cloneable {
    // ...
    public CompositeDrawing getComposite() {
        return null;
    }
    // ...
}
```



6.1.2 The composite case

At the composite case, a CompositeDrawing instance manages a list of composed DrawingObject instances. It provides methods to add, remove, and traverse composed objects.

Additionally, it handles the painting and movement of each child object in the composition. When the position of the composition is moved, so are the positions of its children. Likewise are the sizes of each child. While the drawing for the composition is simply the result of traversing each child and calling its draw method.

```
package arcle.diagrammer;
public class CompositeDrawing extends DrawingObject {
    private List children;
    public CompositeDrawing() {
        children = new LinkedList();
    }

    public void add(DrawingObject child) {
        children.add(child);
    }

    public void remove(DrawingObject child) {
        children.remove(child);
    }

    public Iterator iterator() {
        return children.iterator();
    }

    public CompositeDrawing getComposite() {
        return this;
    }

    public void draw(Graphics graphics) {
        // draw each child in the composition
    }

    public void setSize(int dx, int dy) {
        // set the size of the composition and the sizes of each child
        // object relative to the composition
    }

    public void move(int dx, int dy) {
        // move the composition and each child object
    }

    public void setPosition(int dx, int dy) {
        // set the position of the composition and the positions of
        // each child.
    }
}
```

6.2 Mouse logic implementation

As specified in section 5.4.1, the mouse-handling logic in a diagrammer window uses the *state* pattern. Conforming to this pattern, the program logic is split into two class hierarchies. One hierarchy is the context classes – rooted from MouseContext – which contains all the data for the mouse logic. The other are the state classes – rooted from MouseState – which implements the program logic for the various states.



6.2.1 **MouseListener**

To support existing Swing paradigms, the `MouseListener` implements the `MouseListener` interface defined by Swing. This decision will facilitate the use of `MouseListener` objects in user interface classes that accepts an `addMouseListener` request.

The context forwards all `MouseListener`-defined methods to its current state. Additionally, each state may optionally define `enterState` and `exitState` methods to be notified of state changes in the context.

State changes in the context is handled by the `changeState` method. This method is given package access, since it is only meant to be used by the corresponding `MouseListener` class.

To set the first state of the context, derived classes must override the `getInitialState` method. This method is called by the constructor to initially define the value of `currentState`.

```
package arcle.diagrammer;
// ...
public abstract class MouseListener implements MouseListener {
    private MouseState currentState;

    public MouseListener() {
        currentState = getInitialState();
    }

    protected abstract MouseState getInitialState();

    /**
     * (package access).
     */
    final void changeState(MouseState newState) {
        try {
            currentState.exitState(this);
        } finally {
            currentState = newState;
            currentState.enterState(this);
        }
    }

    public void mouseClicked(MouseEvent arg0) {
        currentState.mouseClicked(this,arg0);
    }

    public void mouseEntered(MouseEvent arg0) {
        currentState.mouseEntered(this,arg0);
    }

    public void mouseExited(MouseEvent arg0) {
        currentState.mouseExited(this,arg0);
    }

    public void mousePressed(MouseEvent arg0) {
        currentState.mousePressed(this,arg0);
    }

    public void mouseReleased(MouseEvent arg0) {
        currentState.mouseReleased(this,arg0);
    }
}
```



6.2.2 MouseState

The real program logic for mouse handling is placed in `MouseState`-derived classes. These classes should not store data on their own; they must rely on a corresponding object from a `MouseContext`-derived class to store the data. This leads to the definition of a `MouseContext` subclass for each family of `MouseState` derivatives.

State changes are done through the `changeState` method. This method is made protected so that state changes may only occur from the internal state logic.

```
package arcle.diagrammer;
// ...
public class MouseState {
    // ...
    protected final void changeState(MouseContext context, MouseState newState) {
        context.changeState(newState);
    }

    public void enterState(MouseContext context) { }
    public void exitState(MouseContext context) { }
    public void mouseClicked(MouseContext context,MouseEvent arg0) { }
    public void mouseEntered(MouseContext context,MouseEvent arg0) { }
    public void mouseExited(MouseContext context,MouseEvent arg0) { }
    public void mousePressed(MouseContext context,MouseEvent arg0) { }
    public void mouseReleased(MouseContext context,MouseEvent arg0) { }
}
```

6.3 Mouse singleton states

Each mouse-handling state is implemented as a singleton class. This decision was made so that these state classes are easily accessible and only requires small management overhead – reducing the need of being passed around in object creations. A drawback of this design is now the instances – being static objects – *must not* store data in their own objects; They must rely on the context for data storage.

Therefore, the class `UmlDiagrammerMouseContext` (code not shown) is provided for storing data of these singleton classes. Upon entry of each state's method, the context parameter is cast into `UmlDiagrammerMouseContext`, which is a subclass of `MouseContext`. The states then use data in that context.

```
package arcle.uml_diagrammer;
// ...
public class MouseNormalState extends MouseState {
    private MouseNormalState() { }
    // ...
    private static MouseNormalState theInstance = null;

    public static final MouseNormalState getInstance() {
        if(theInstance == null) {
            theInstance = new MouseNormalState();
        }
        return theInstance;
    }
    // ...
}
```



6.4 Stereotype renderings

The UML standard states that non-default rendering of UML things are done through stereotyping. The system satisfies this standard by defining strategy renderers and factory classes that creates drawing objects accompanied by those renderers.

6.4.1 Strategy renderers

Each `UmlDrawingObject`-derived classes do not define the algorithm used to render itself on the screen. It instead relies on a separate stereotype object to provide the drawing functionality. This application of the *strategy* pattern allows object renderings to be varied greatly, including providing support for user-defined renderings.

```
package arcle.uml_diagrammer;
//...
public class UmlDrawingObject extends DrawingObject {
    private ObjectStereotype theStereotype;
    private String name;
    //...
    public ObjectStereotype getStereotype() {
        return theStereotype;
    }

    public void setStereotype(ObjectStereotype s) {
        theStereotype = s;
    }

    public String getName() {
        return name;
    }

    public void setName(String n) {
        name = n;
    }

    public void draw(Graphics g) {
        ObjectStereotype s = getStereotype();
        if(s != null) {
            s.drawObject(g,this);
        }
        else {
            renderDefault(g);
        }
    }

    /**
     Provide a default rendering just in case the stereotype does not exist.
    */
    protected void renderDefault(Graphics g) {
        Point p = getPosition();
        if(p != null) {
            String str = getName() + ":" + getClass().getName();
            g.drawString(str,p.x,p.y);
        }
    }
}
```

Each `ObjectStereotype`-derived class provides the strategy for a certain rendering algorithm. Several of these strategy classes are defined for the normal UML renderings. More classes may be defined in the future to employ user-defined rendering algorithms.

```
package arcle.uml_diagrammer;
//...
public class ObjectStereotype {
    //...
```



```
    public void drawObject(Graphics graphics, UmlDrawingObject object) { }  
}
```

6.4.2 Factory stereotypes

The factory classes for UmlDrawingObject-derived classes not only create objects of that type. These classes also create the initial stereotype that is the rendering strategy for the object.

```
package arcle.uml_diagrammer;  
// ...  
public class UseCaseObjectFactory extends UmlDrawingObjectFactory {  
    // ...  
    public UmlDrawingObject createUmlDrawingObject(String stereotypeName) {  
        UseCaseObjectStereotype stereotype;  
        if(stereotypeName == null) {  
            stereotype = new UseCaseObjectStereotype();  
        }  
        else if(stereotypeName.equals("someStereotype")) {  
            stereotype = new UseCaseObjectStereotype();  
            stereotype.setSomething("blabla");  
            stereotype.setAnother(1234);  
        }  
        else if(stereotypeName.equals("coolStereotype")) {  
            stereotype = new SomeCoolUseCaseStereotype();  
            stereotype.setSomething("cool, dude!");  
        }  
        else {  
            // unknown stereotype  
            stereotype = new UseCaseObjectStereotype();  
            stereotype.setSomething("unknown");  
        }  
        UmlDrawingObject obj = new UseCaseObject();  
        obj.setStereotype(stereotype);  
        return obj;  
    }  
}
```

6.5 Relationship adapter

The problem described in section 5.3.2 leads to the use of object composition to simulate multiple inheritance. This *hack* is done through the application of the *adapter* pattern. Implementation leads to the definition of the adapter itself and the factory class that create the adapted object along with its adapter.

6.5.1 Adapter implementation

A UML relationship is a visible UML object. Additionally it is also capable of connecting other drawing objects. Therefore, the RelationshipObject also have methods for managing object interconnections. Since UmlDrawingObject is derived from DrawingObject, the drawing system cannot readily use these connection-management methods. This is due to the fact that the system can only deal with ConnectorObject-derived classes when managing connectors.

```
package arcle.uml_diagrammer;  
// ...  
public class RelationshipObject extends UmlDrawingObject {  
    // ...  
    /*-----  
    BEGIN: additional operations for a ConnectorObject  
    */  
}
```



```
public boolean connect(DrawingObject object, ConnectorEndpoint where) {
    // define the connect algorithm here
}

public ConnectorEndpoint endPointAt(int x, int y, int tolerance) {
    // look for the end point at the specified point
    // and then return it
}

protected List createEndpoints() {
    // create the list of endpoints and return it
}

/*-----
END: additional operations for a ConnectorObject
*/
}
```

The drawing system deals with relationship objects through instances of RelationshipObjectAdapter, a subclass of ConnectorObject. The adapter contains the real RelationshipObject, and forward its method calls to the contained object.

```
package arcle.uml_diagrammer;
// :
public class RelationshipObjectAdapter extends ConnectorObject {
    /**
    The adaptee object.
    */
    public RelationshipObject theRelationshipObject;

    public RelationshipObjectAdapter(RelationshipObject ro) {
        theRelationshipObject = ro;
    }

    /*-----
BEGIN: ConnectorObject methods
    */

    public boolean connect(DrawingObject object, ConnectorEndpoint where) {
        return theRelationshipObject.connect(object,where);
    }

    public ConnectorEndpoint endPointAt(int x, int y, int tolerance) {
        return theRelationshipObject.endPointAt(x,y,tolerance);
    }

    /**
    To be defined by derived classes to create and initialize
    the list of ConnectorEndPoint (factory method).
    @see ConnectorEndPoint
    */
    protected java.util.List createEndpoints() {
        return theRelationshipObject.createEndpoints();
    }

    /*-----
END: ConnectorObject methods
    */

    /*-----
BEGIN: DrawingObject methods
    */
    public void draw(Graphics graphics) {
        theRelationshipObject.draw(graphics);
    }

    public void setSize(int dx, int dy) {
        theRelationshipObject.setSize(dx,dy);
    }

    public void move(int dx, int dy) {
        theRelationshipObject.move(dx,dy);
    }
}
```



```
public CompositeDrawing getComposite() {
    return theRelationshipObject.getComposite();
}

public Point getPosition() {
    return theRelationshipObject.getPosition();
}

public void setSize(Dimension size) {
    theRelationshipObject.setSize(size);
}

public void setPosition(int dx, int dy) {
    theRelationshipObject.setPosition(dx,dy);
}

public void setPosition(Point pos) {
    theRelationshipObject.setPosition(pos);
}

public void showEditDialog(JFrame ownerFrame) {
    theRelationshipObject.showEditDialog(ownerFrame);
}

/*-----
END: DrawingObject methods
*/
}
```

6.5.2 Adapter Factory

Currently, the only subclass of RelationshipObject is AssociationObject. Its corresponding factory, AssociationObjectFactory, creates objects of AssociationObject packaged in a RelationshipObjectAdapter object.

```
package arcle.uml_diagrammer;
//...
public class AssociationObjectFactory extends UmlDrawingObjectFactory {
    // overrides DrawingObjectFactory's method
    public DrawingObject createDrawingObject() {
        return createRelationshipAdapter(null);
    }

    public UmlDrawingObject createUmlDrawingObject(String stereotypeName) {
        ObjectStereotype stereotype;
        if(stereotypeName == null) {
            // create default stereotype
            stereotype = new AssociationObjectStereotype();
        }
        else {
            // non-standard stereotypes are not yet supported
            stereotype = new AssociationObjectStereotype();
        }
        AssociationObject as = new AssociationObject();
        as.setStereotype(stereotype);
        return as;
    }

    public RelationshipObjectAdapter createRelationshipAdapter(
        String sterotypeName) {
        return new RelationshipObjectAdapter((RelationshipObject)
            createUmlDrawingObject(sterotypeName));
    }
}
```

6.6 Setting up the window

The following fragment illustrates how DiagrammerWindow creates and lays out its child components. Notice that the *decorator* pattern is employed by JScrollPane which adds scroll bar decorations to the DrawingCanvas object.



```
// ...  
public class DiagrammerWindow {  
    // ...  
    protected void setupContent() {  
        toolBar      = new DrawingToolBar();  
        canvas       = new DrawingCanvas();  
        menuBar      = new JMenuBar();  
        canvasScrl   = new JScrollPane(canvas);  
        statusBar    = new JContainer();  
        // ...  
        setLayout(new BorderLayout());  
        add(canvasScrl, BorderLayout.CENTER);  
        add(toolBar, BorderLayout.WEST);  
        add(statusBar, BorderLayout.SOUTH);  
        // ...  
    }  
}
```

7 Conclusion

This concludes the design specification for Use-Case Diagrammer. It is recognized that this specification does not cover all possible aspects of the implementation. This is because a good design only elicits those essential aspects needed to aid construction. An outright design will be equivalent to the system's source code itself.

Several details need to be refined at the time of implementation. These details are omitted from the design to give freedom to the implementor to augment the design. Among those aspects are:

- Diagram file format.
- Dialogs and mechanisms for setting colors, fonts, etc.
- Context-sensitive help.

Currently, the design is kept simple to ease the construction of the system. Further developments of the system may incorporate additional features. The following is a suggestion of the additional features to be implemented following the initial release of the system:

- Multi-level undo. This feature is accomplishable by using the *command* pattern, where each command could undo its own effects. A history of the command objects are kept to provide multiple undo/redo functionality.
- Multiple views of the same document. The responsibility of managing the objects may be factored out from *DiagrammerWindow* and into a separate document object. This object may be observed by several views – the *DiagrammerWindow* objects that are now are made *observers* of the document. This design resembles the Doc/View model of Borland's Object-Windows library.