Technical University Hamburg-Harburg

Software Systems Department

Project Work

Review of the JHotDraw framework

Jolita Savolskyte

Information and Media Technologies
Matriculation No. 20668

April 2004

Table of Contents

Index of Figures

1. Introduction

1.1. Motivation

Software development is largely based on standards, conventions, underlying systems and architectures, most notably object-oriented frameworks. The frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate. A good framework can reduce the cost of developing an application by an order of magnitude because it lets you reuse both design and code. The JHotDraw framework [1] is a well-designed, well-maintained Java/Swing framework for jumpstarting the implementation of a structured drawing editor. Although not as comprehensive as other drawing editor frameworks [2], it can be extended to support many notations (e.g. UML class diagrams [3]). The extensions are carried out by adding classes. Given that the concepts involved in this extension mechanism can be represented at a higher level of abstraction, the possibility exists to define an XML dialect to represent those extensions. Although such dialect cannot be expected to specify all the features of an extension (the visual syntax, the interaction with the underlying Model in the sense of the Model-View-Controller paradigm, etc.) it can still contribute to generating a large portion of the implementation.

The implementation of most diagram editors follows the MVC (Model-View-Controller) design pattern (see Figure 1). This pattern prescribes a distribution of functionality among classes, into three categories:

- Classes responsible for Model management. Their responsibilities include as a minimum: (a) creation of model elements, (b) consistent update of model elements, (c) query, and (d) notification of changes to registered listeners (see below). Additional responsibilities may include providing support for persistence and undo.
- Classes responsible for View management. These classes serve two main purposes: displaying the model in a visual way, and accepting user input. As we will see, several views may be used concurrently to display different portions of

the same underlying model (e.g. this may manifest in terms of windows used to manipulate the same diagram). In its purest form, views are updated not directly as result from user input, but only after listening to the changes validated by the Model.

- Classes responsible for mediation between the view instance(s) and the model instance. These classes (may be in fact only one class) translate user input to requests for updates against the model, and realize the workflow (if any) that results from model updates.

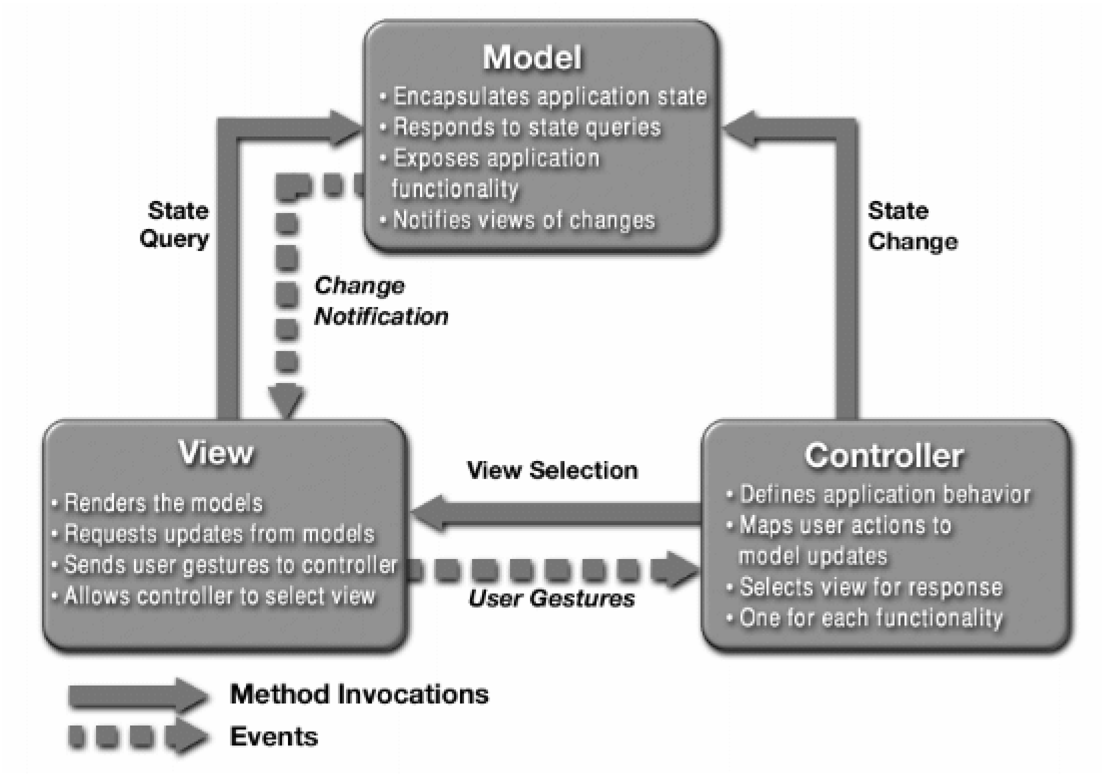

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 1: Structure of the Model-View-Controller design pattern (as shown in http://java.sun.com/blueprints/patterns/MVC-detailed.html)

The JHotDraw framework follows the MVC guidelines. The functionality that a JHotDraw-based diagram editor provides results from the interaction of: (a) Swing-based classes (related to the View management concern), (b) a very thin controller, and (c) the Model classes which are application dependent and are written entirely by the framework programmer. The Drawing could be (can be thought of) as the model as it contains the current state of the application; a separate class called `DrawingView` represents the drawing's appearance in the application's GUI (see Figure 2).

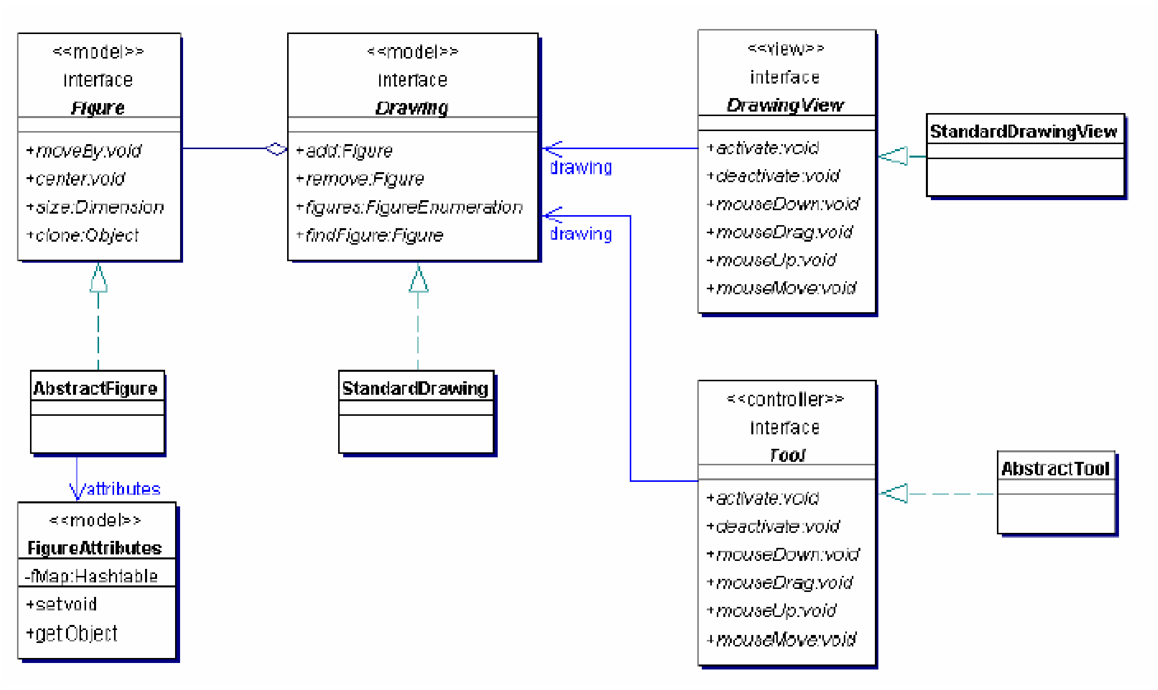


Figure 2: MVC in JHotDraw [taken from STS lecture material for Software Architectures]

In JHotDraw, you manipulate graphical objects in your view with the help of Tools, which partly play the role of a controller. Changes in the view should then be propagated to the model. But the model could be modified such that it notifies the view of changes. While JHotDraw is not pure in that respect, it still separates model, view, and controller. The separation of model, view, and controller is often not very distinct in practice [8].

When specifying the expected functionality of such editors, the same guidelines can be used to better structure the specification:

- The programmer may have a description of the entities in the Model (plus their consistency rules). In an ideal specification, the permissible state evolutions are also made explicit.
- Sooner or later, the programmer will decide about which GUI elements to use to stand for model elements (e.g. a rectangle with three compartments will stand for

a UML class element, and so on). Another example is the decision about which GUI operations correspond to which Model operations (e.g. pressing the Del key while a figure is selected corresponds to invoking the `deleteRequest()` operation on the instance associated to the graphical figure).

Therefore these aspects are important for the readability of the declarative specification of a diagram editor's functionality. These considerations justify our choice to specify them separately in the XML document used as input to the diagram-editor generator. The motivation stated above is provided by the supervisor of this project Miguel Garcia.

The XML dialect for JHotDraw reflects geometrical figures and their distribution on the drawing. The XML Schema document specifies the allowed elements and operations in the XML input document. Before this specification JHotDraw was not concerned with XML.

1.2. Structure of this report

This section gives a brief overview of the content of this project report.

Chapter 2 provides the classification and the description of the design patterns used in JHotDraw diagram editor, examples of their applications. The functionality of the framework in terms of classes and methods is discussed in chapter 3. Chapter 4 starts with the user view of JHotDraw, the GUI interface description. Then the XML dialect is introduced with its sample codes. Chapter 5 concludes this report.

2. Patterns in JHotDraw framework

JHotDraw is a Java version of the original Smalltalk HotDraw framework developed by Kent Beck and Ward Cunningham. Thomas Eggenschwiler and Erich Gamma did the rewrite in Java as an exercise in applying design patterns.

Patterns describe the purpose of a framework and can show many of the design details embodied in the framework. Application programmers can use a framework described in patterns language without having to understand in detail how it works.

2.1 Pattern classification

Design patterns vary in their granularity and level of abstraction. There are many design patterns so it is good to classify them. The patterns can be classified by two criteria: purpose and scope. The purpose reflects what a pattern does. They can have creational, structural, or behavioral purpose. The scope criterion "specifies whether the pattern applies primarily to classes or to objects" [4]. Class patterns are based on relationships between classes, which have mainly inheritance structures. Object patterns dynamically lets object reference each other. [10] Most patterns are in the Object scope.

	Purpose			
		Creational	**Structural**	**Behavioral**
Scope	***Class***	Factory Method		Template Method
	Object	Prototype	Composite Decorator Adapter	Strategy Observer State Command

Table 1: Pattern classification used in JHotDraw

Creational pattern deals with class instantiation. Class–creation patterns defer some part of the object creation process to subclasses, whereas object-creation patterns defer to another object.

Structural patterns deal with compositions of objects and classes. Structural class patterns are based on inheritance to build a structure. Structural object patterns use object references to build a structure. [10]

Behavioral patterns are used to distribute responsibility between classes and objects. The patterns define how the classes and objects should interact, and what responsibilities each participant has. Behavioral class patterns are based on inheritance. They are mostly concerned with handling algorithms and flow of control. Behavioral object patterns describe how objects can cooperate to carry out tasks that no single object can carry out alone. [10]

2.2 The Composite design pattern

The basic figures cannot represent the internal structure of some figures we want to have in a diagram. The complex figures may be composed of several simpler figures which show the attributes of those complex figures. The Composite design pattern allows a user to treat single components and collections of components identically.

Complicated figures like `ClassFigure` in JModeller can be composed of several simpler figures. For example, a `ClassFigure` has a `RectangleFigure`, one `TextFigure` for the class name and several `TextFigure`s for attributes and methods. The `ClassFigure` is a subclass of `GraphicalCompositeFigure` that in turn is a subclass of `CompositeFigure`. `GraphicalCompositeFigure` manages contained figures like the `CompositeFigure` does, but delegates its graphical presentation to another graphical figure whose purpose is to draw the container for all contained figures [9]. The `GraphicalCompositeFigure` class has default constructor (see Listing 1) that uses a `RectangleFigure` as presentation figure to draw a box around the whole container figure and constructor (see Listing 2) which creates a `GraphicalCompositeFigure` with a given graphical figure for presenting it. If `CompositeFigure` does not contain any figure, it is invisible in the drawing as it has no mechanism to draw itself. It depends on its contained figures to draw themselves therefore the CompositeFigure gets its appearance. However, the `GraphicalCompositeFigure` presentation figure can draw itself even when the `GraphicalCompositeFigure` contains no other figures.

```
public GraphicalCompositeFigure() {
    this(new RectangleFigure());
  }
```

Listing 1: Default constructor of `GraphicalCompositeFigure`

```
public GraphicalCompositeFigure(Figure newPresentationFigure) {
    super();
    setPresentationFigure(newPresentationFigure);
    initialize();
  }
```

Listing 2: The constructor of `GraphicalCompositeFigure` with attribute

The behavior of `GraphicalCompositeFigure` is delegated from the container to its parts by calling methods. An example how `CompositeFigure` delegates behavior to its components is the `draw()` method of the `CompositeFigure` class:

```
 /**
   * Draws only the given figures
   * @see Figure#draw
   */
  public void draw(Graphics g, FigureEnumeration fe) {
    while (fe.hasNextFigure()) {
      fe.nextFigure().draw(g);
    }
  }
```

The `draw()`method first gets the next figure of the enumeration and invokes the `draw()` method of the corresponding figure class to draw that figure.

`StandardDrawing` is a subclass of `CompositeFigure` and implements the `Drawing` interface. The Drawing is a container for figures and it consist of other figures, but it is itself a figure.

2.3 The State pattern

The intent of the State design pattern is to "allow an object to alter its behavior when its internal state changes. The object will appear to change its class". [4]

JHotDraw drawing editor has a tool palette from which users select a tool. Once the selected tool is done the tool palette is set back to `SelectionTool` by default. The editor class keeps track of one tool which we use to select and manipulate figures. So it is in one of three states: background selection, figure selection, and handle manipulation. The different states are handled by different child tools. For capturing the mouse event the programmer will need to customize the behavior of the selection tool. [10]

The users sometimes need to define their own set of tools. When we use the `SelectionTool` to select a `ClassFigure` it activates only the container of the `ClassFigure`, but not the `TextFigure`s contained in it. If we want to edit any of its contained `TextFigure`s we must double-click on a `ClassFigure`. This action is carried out by a `CustomSelectionTool`, which is a subclass of `SelectionTool`. It recognizes double-clicks and popup menu triggers. If double-click or popup trigger is encountered a hook method is called which handles the event [9]. This method is overridden in subclass `DelegationSelectionTool` to delegate mouse selection to a specific `TextTool` if the figure selected inside a `CompositeFigure` is a `TextFigure`. `CustomSelectionTool` overrides the `handleMouseClick()` and `handleMouseDoubleClick()` methods of its superclass (see Listing 3).

```
/**
     * Hook method which can be overridden by subclasses to provide
     * specialized behavior in the event of a mouse double click.
     */
    protected void handleMouseDoubleClick(MouseEvent e, int x, int y) {
        Figure figure = drawing().findFigureInside(e.getX(), e.getY());
        if ((figure != null) && (figure instanceof TextFigure)) {
            getTextTool().activate();
            getTextTool().mouseDown(e, x, y);
        }
    }
protected void handleMouseClick(MouseEvent e, int x, int y) {
        deactivate();
    }

/**
     * Terminates the editing of a text figure.
```

```
      */
    public void deactivate() {
        super.deactivate();
        if (getTextTool().isActive()) {
          getTextTool().deactivate();
        }
    }
```

Listing 3: Methods of `DelegationSelectionTool` class

The main idea in this pattern is to introduce an abstract class called `AbstractTool` to capture mouse events (Figure 3). The `AbstractTool` class declares an interface common to all classes that represent different operational states. Subclasses of `AbstractTool` implement the state-specific behavior. The `AbstractTool` class is a default implementation of the interface `Tool`. Tool plays the role of the State. In encapsulates all state specific behavior. `DrawingView` plays the role of the StateContext. A selection tool delegates the state specific behavior to its current child tool. A tool can be in the following states:

disabled<->enabled[unusable<->usable[active<->inactive]]<->always_usable[active<->inactive]<->disabled

where each square bracket indicates a state nesting level and arrows possible state transitions. Unusable tools are always inactive as well and disabled tools are always unusable as well. [9] The State pattern separates the state from its context. The objects with defined interface become the states and the context uses the object. So, we can say that the `DelegationSelectionTool` is one more state for the drawing view as the context which accepts the user input and delegates it to the tool.

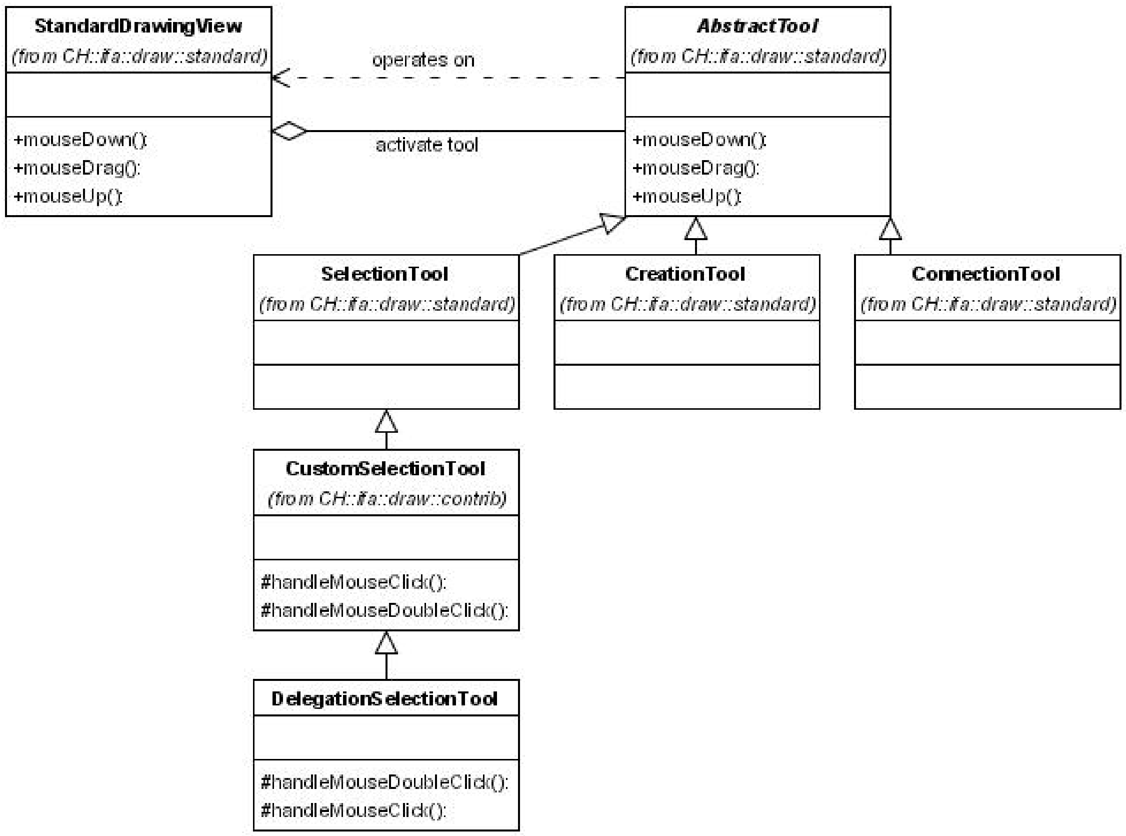


Figure 3: State pattern in JHotDraw [3]

2.4 The Template method

The Template Design Pattern is one of the most widely used and useful design patterns. It defines a certain algorithm in a parent class leaving some of the details to be implemented in subclasses. Subclasses can redefine some parts of the algorithm without changing its structure. If a base class is an abstract class, then it is a simple form of the Template method pattern.

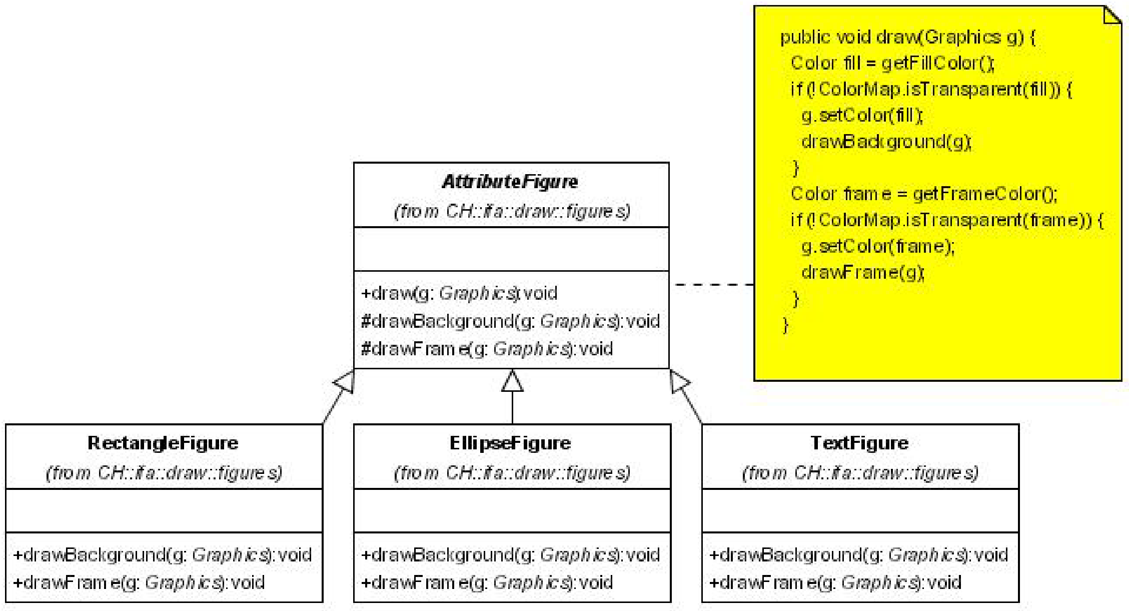
An example of a Template method is in the `AttributeFigure` class. An `AttributeFigure` keeps track of an open ended set of attributes of a figure. This class provides a template method `draw(Graphics)` calling `drawBackground()` followed by `drawFrame()`(see Figure 4). These two methods are called Hook methods. They contain a default implementation of drawing the background and the frame of the figure respectively and they are overridden in derived classes such as `RectangleFigure`, `TextFigure` and so on. Usually Hook methods are intended to be overridden. The following code demonstrates this principle in `AttributeFigure` and `RectangleFigure` classes:

```java
public abstract class AttributeFigure extends AbstractFigure {
...
  /**
   * Draws the figure in the given graphics. Draw is a template
   * method calling drawBackground followed by drawFrame.
   */
  public void draw(Graphics g) {
    Color fill = getFillColor();
    if (!ColorMap.isTransparent(fill)) {
      g.setColor(fill);
      drawBackground(g);
    }
    Color frame = getFrameColor();
    if (!ColorMap.isTransparent(frame)) {
      g.setColor(frame);
      drawFrame(g);
    }
  }
  /**
   * Draws the background of the figure.
   */
  protected void drawBackground(Graphics g) {
  }

  /**
   * Draws the frame of the figure.
   */
  protected void drawFrame(Graphics g) {
  }
...
}

public class RectangleFigure extends AttributeFigure {
...
public void drawBackground(Graphics g) {
    Rectangle r = displayBox();
    g.fillRect(r.x, r.y, r.width, r.height);
  }

  public void drawFrame(Graphics g) {
    Rectangle r = displayBox();
    g.drawRect(r.x, r.y, r.width-1, r.height-1);
  }
...
}
```


AttributeFigure
(from CH::ifa::draw::figures)
+draw(g: Graphics):void
#drawBackground(g: Graphics):void
#drawFrame(g: Graphics):void
RectangleFigure
(from CH::ifa::draw::figures)
+drawBackground(g: Graphics):void
+drawFrame(g: Graphics):void
EllipseFigure
(from CH::ifa::draw::figures)
+drawBackground(g: Graphics):void
+drawFrame(g: Graphics):void
TextFigure
(from CH::ifa::draw::figures)
+drawBackground(g: Graphics):void
+drawFrame(g: Graphics):void

```
public void draw(Graphics g) {
  Color fill = getFillColor();
  if (! ColorMap.isTransparent(fill)) {
    g.setColor(fill);
    drawBackground(g);
  }
  Color frame = getFrameColor();
  if (! ColorMap.isTransparent(frame)) {
    g.setColor(frame);
    drawFrame(g);
  }
}
```

Figure 4: Template method UML diagram

There are more examples of Template method pattern in JHotDraw:

- The methods `moveBy(int, int)` and `displayBox(Point, Point)` in the `AbstractFigure` class are the template methods and defines a sequence of methods which always should be called.

```
public void moveBy(int dx, int dy) {
    willChange();
    basicMoveBy(dx, dy);
    changed();
  }

public void displayBox(Point origin, Point corner) {
    willChange();
    basicDisplayBox(origin, corner);
    changed();
  }
```

  The methods `basicMoveBy()` and `basicDisplayBox()` are the Hook methods and they are overridden in subclasses e.g. `PolyLineFigure`.

- The `open(DrawingView)` method of the `DrawApplication` class implements the Template design pattern, where `open(DrawingView)` is the template and methods such as `createTools(JToolBar)` and `createDrawing()` provide hooks that may be overridden in the subclasses e.g. JavaDrawApp, MDI_DrawApplication. The method `open(DrawingView)` initializes all the framework components and combines them together to create the application. [11]

- The Template design pattern is used in the `LineConnection` class. It provides `connectEnd(Connector)` and `disconnectEnd()` as Template methods. The subclasses `AssociationLineConnection` and `InheritanceLineConnection` need to complete the provided Hook methods `handleConnect()` and `handleDisconnect()` to establish a relationship between classes. To establish a connection between two figures the Template method performs a sequence of steps. It tests whether the two figures can be connected by calling `canConnect()`, then sets the start and end figures, and finally calls the Hook method `handleConnect()`.

2.5 The Factory method

The Factory method "defines an interface for creating an object, but let subclasses decide which class to instantiate". Factory Method lets a class defer instantiation to subclasses [4].

This pattern is used in JHotDraw in order to create the user interface components at one particular location. The menus and tools are created in the `DrawApplication` class by the separate methods. This class contains many Factory methods like `createTools(JToolBar)`, `createMenus(JMenuBar)` which in turn call `createFileMenu()`, `createEditMenu()` and so on. Menus in the menu bar are created by the separate methods, i.e. the `createFileMenu()` method creates the File menu. New menus in an application can be created using the same technique. The subclasses of `DrawApplication` such as `JavaDrawApp`, `JModellerApplication`, `PertApplication` of `DrawApplication` alter or add new menus and their menu items to meet the requirements of the application. The `createTools()` method in `DrawApplication` adds only the selection tool. Therefore, `JavaDrawApp` override this method to add additional tools for creating geometrical figures (e.g. `RectangleFigure`, `EllipseFigure`, `LineConnection`) (see Listing 4).

```
protected void createTools(JToolBar palette) {
    super.createTools(palette);

    Tool tool = new ZoomTool(this);
    palette.add(createToolButton(IMAGES + "ZOOM", "Zoom Tool", tool));

    tool = new UndoableTool(new TextTool(this, new TextFigure()));
    palette.add(createToolButton(IMAGES + "TEXT", "Text Tool", tool));
    tool = new UndoableTool(new CreationTool(this, new RectangleFigure()));
    palette.add(createToolButton(IMAGES + "RECT", "Rectangle Tool", tool));
     …
  }

protected void createMenus(JMenuBar mb) {
    super.createMenus(mb);
    addMenuIfPossible(mb, createAnimationMenu());
    addMenuIfPossible(mb, createImagesMenu());
    addMenuIfPossible(mb, createWindowMenu());
  }
```

Listing 4 : `createTools()` and `createMenus()` in `JavaDrawApp`

The application also overrides `createSelectionTool()` to create own `MySelectionTool` which "interprets double clicks to inspect the clicked figure" [9] whereas `JModellerApplication` overrides it to create `DelegationSelectionTool` which reacts on the right mouse button and shows a popup menu.

2.6 The Decorator design pattern

The Decorator design pattern offers a flexible alternative to subclassing. The main difference between subclassing and the Decorator pattern is this: with subclassing, you work with the class, whereas in the Decorator pattern, you modify objects dynamically. When you extend a class, the change you make to the child class will affect all instances of the child class. With the Decorator pattern, however, you apply changes to each individual object you want to change. [12]

An example of the Decorator pattern in JHotDraw is `UndoableTool` (see Listing 5). It implements an interface `Tool` and its constructor accepts another Tool to be decorated.

The UndoableTool class also defines a Tool called myWrappedTool to reference the decorated Tool.

```
public class UndoableTool implements Tool, ToolListener {

  private Tool myWrappedTool;

  public UndoableTool(Tool newWrappedTool) {
    setEventDispatcher(createEventDispatcher());
    setWrappedTool(newWrappedTool);
    getWrappedTool().addToolListener(this);
  }
  ...
}
```

Listing 5: The UndoableTool class' constructor

The Decorator pattern is used in the createTools(JToolBar) method of the JavaDrawApp class:

```
Tool tool = new UndoableTool(new CreationTool(this, new RectangleFigure()));
```

We create an instance of the UndoableTool and pass it the CreationTool it should decorate. An interface of the UndoableTool object is identical to the CreationTool. Any calls that the UndoableTool gets, it conveys to its contained CreationTool object and adds its own functionality. The way it forwards the invocation of the mousedown() method to the wrapped object, could be seen in this example:

```
public void mouseDown(MouseEvent e, int x, int y) {
    getWrappedTool().mouseDown(e, x, y);
```

Another example of the Decorator pattern is DecoratorFigure. A decorator figure is used to decorate other figures with decorations like borders. It defines a Figure called myDecoratedFigure to reference the decorated figure and the constructor accepts a Figure to be decorated. DecoratorFigure forwards all its method invocations to its contained figure. Subclasses of the DecoratorFigure such as BorderDecorator (see Figure 5) may override these methods to extend and alter behavior.

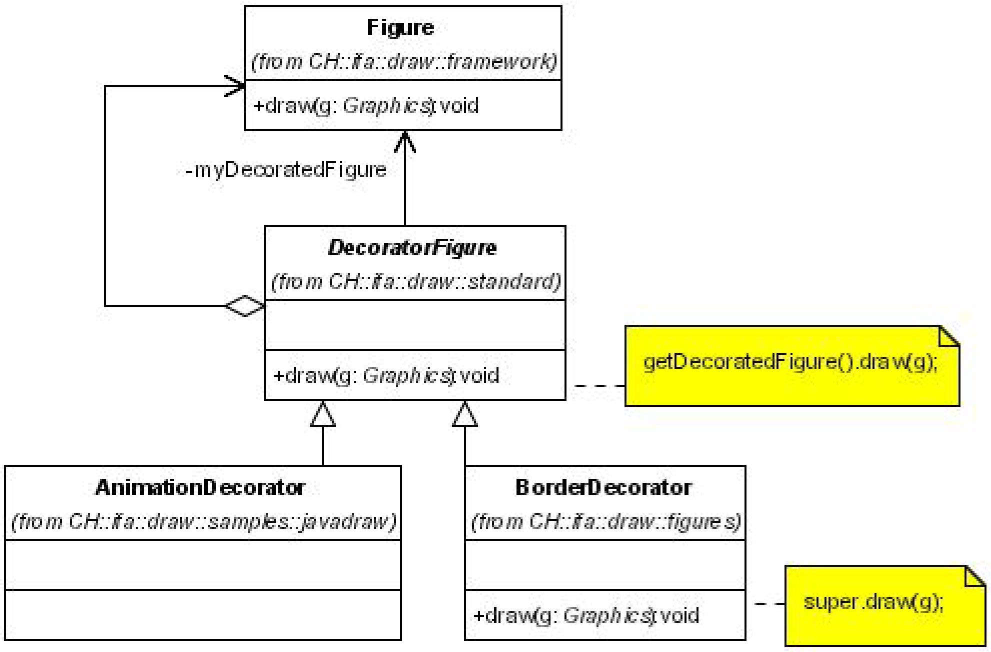


Figure
(from CH::ifa::draw::framework)
+draw(g: Graphics):void
-myDecoratedFigure
DecoratorFigure
(from CH::ifa::draw::standard)
+draw(g: Graphics):void
getDecoratedFigure().draw(g);
AnimationDecorator
(from CH::ifa::draw::samples::javadraw)
BorderDecorator
(from CH::ifa::draw::figures)
+draw(g: Graphics):void
super.draw(g);

Figure 5: Decorator pattern using `DecoratorFigure`

2.7 The Prototype design pattern

The Prototype design pattern specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. [4]

`CreationTool` in the `cretateTools(JToolBar)` method of the `JavaDrawApp` class uses the Prototype design pattern to initialize each tool with an instance of the figure it is intended to create (see Figure 6). Every creation tool in JHotDraw creates a new Figure prototype by cloning an instance of a `Figure` subclass. The constructor of the `CreationTool` class receives this instance of the type of figure to be created. The cloning mechanism is defined in the `clone()` method of the `AbstractFigure` class which provides a default implementation for the `Figure` interface (see Listing 6).

```
public Object clone() {
    Object clone = null;
    ByteArrayOutputStream output = new ByteArrayOutputStream(200);
    try {
      ObjectOutput writer = new ObjectOutputStream(output);
      writer.writeObject(this);
      writer.close();
    }
    catch (IOException e) {
      System.err.println("Class not found: " + e);
    }

    InputStream input = new ByteArrayInputStream(output.toByteArray());
    try {
      ObjectInput reader = new ObjectInputStream(input);
      clone = reader.readObject();
    }
    ...
    return clone;
}
```

Listing 6: The `clone()` method of the `AbstractFigure` class

We can only clone objects that are declared to implement the `Cloneable` interface as it is marked in the `Cloneable` class [13]. The `Figure` interface fulfils this requirement so we can clone any instance of the `Figure` subclass. The `Figure` interface is declared as `Serializable` therefore in the `clone()` method we can write the bytes to an output stream and read them back in to create a complete data copy of that instance of a class. This cloning method is called a deep copy because the clone and the original objects are independent [4].

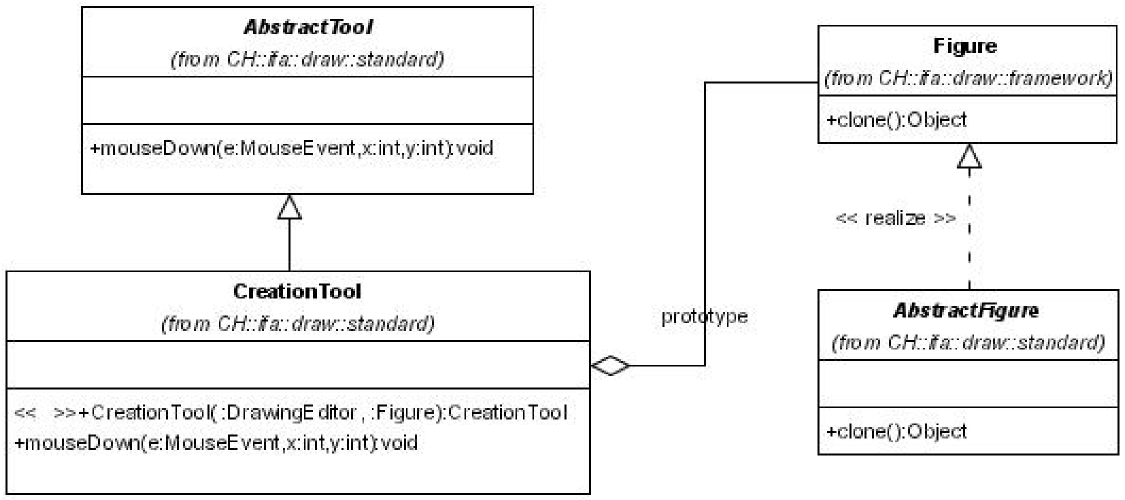

AbstractTool
(from CH::ifa::draw::standard)
+mouseDown(e:MouseEvent,x:int,y:int):void
Figure
(from CH::ifa::draw::framework)
+clone():Object
<< realize >>
CreationTool
(from CH::ifa::draw::standard)
<< >>+CreationTool(:DrawingEditor, :Figure):CreationTool
+mouseDown(e:MouseEvent,x:int,y:int):void
prototype
AbstractFigure
(from CH::ifa::draw::standard)
+clone():Object

Figure 6: Prototype pattern in JHotDraw

Another example of the Prototype design pattern is in `ConnectionTool`. This class is used to create tools in JHotDraw framework that can be used to instantiate any kind of `ConnectionFigure`. In order to create a particular tool the `ConnectionTool` constructor is called and the prototype of the `ConnectionFigure` passed into it.

2.8 Command design pattern

The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects. [4]

JHotDraw provides menu items, buttons which allow the user to tell the program what action to perform. When the user selects one of these actions, the program receives an `ActionEvent`, i.e. the `actionPerformed` event, which it must capture by subclassing.

Each choice in a menu bar is an instance of a `CommandMenu` class. A `DrawApplication` class creates the menus and their menu items together with the rest of the user interface. Each menu item is configured to have an instance of a concrete `Command` subclass, which could be wrapped inside the other `Command` subclass (see Decorator pattern):

```
protected JMenu createEditMenu() {
    CommandMenu menu = new CommandMenu("Edit");
    menu.add(new UndoableCommand(new DeleteCommand("Delete", this)));
    ...
}
```

When the user selects a menu item, it causes an `actionPerformed` event:

```
public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    for (int i = 0; i < getItemCount(); i++) {
      JMenuItem item = getItem(i);
      if (source == item) {
        Command cmd = (Command) hm.get(item);
        if (cmd != null) {
            cmd.execute();
        }
        break;
...
  }
```

The method gets a selected menu item and the corresponding Command. The command calls its `execute()` method and `execute()` carries out the operation. `CommandMenu`s do not know which subclass of `Command` they use.

The important reason for using Command design patterns is that they provide a convenient way to store and execute an Undo function. Each `Command` object can remember what it just did and restore that state when requested to do so. [14]

3. Internal working of the JHotDraw framework

3.1 Context menus for the figures

Different context menus (PopupMenu) appear for distinct figures. A PopupMenu for associations allows switching between directed and not directed associations and between associations and aggregations. A PopupMenu for classes allows adding attributes and methods.

3.1.1 Context menus for an association line

A popup menu is constructed during creation of the corresponding figures.

An `AssociationLineConnection` represents an association relationship (has-a) between two classes (represented by their `ClassFigures`). An association can either be bi-directional or uni-directional. An association can be turned into an aggregation that can be regard a special kind of association. [9]

The method `clone()` of the class `AbstractFigure` clones the figure and it calls the method `readObject(ObjectInputStream)` of the class `AssociationLineConnection` to read a serialized `AssociationLineConnection` from an input stream and activate the popup menu.

```
s.defaultReadObject();
setAttribute(Figure.POPUP_MENU, createPopupMenu());
```

The `createPopupMenu` method creates a context menu that is initially empty. The method `add()` in the following code listing 7 (Line 03) appends the popup menu with the item “aggregation” to the end. The popup menu of the newly created association line contains two menu items “aggregation” and “uni-directional”. “It allows switching between associations and aggregation and directed and not-directed associations depending on the current kind of association. For uni-directional association, the reference from the destination class to the source class is removed, while for bi-directional association, this relation is established again.” [9]

```java
protected JPopupMenu createPopupMenu() {
        JPopupMenu popupMenu = new JPopupMenu();
        popupMenu.add(new AbstractAction("aggregation") {
                public void actionPerformed(ActionEvent event) {
                    setAggregation(!isAggregation());
                    if (isAggregation()) {
                        ((JMenuItem)event.getSource()).setText("no aggregation");
                    }
                    else {
                        ((JMenuItem)event.getSource()).setText("aggregation");
                    }
                }
            });
        popupMenu.add(new AbstractAction("uni-directional") {
                public void actionPerformed(ActionEvent event) {
                    setUniDirectional(!isUniDirectional());
                    if (isUniDirectional()) {
                        ((JMenuItem)event.getSource()).setText("bi-directional");
                        JModellerClass startClass = ((ClassFigure)startFigure()).getModellerClass();
                        JModellerClass endClass = ((ClassFigure)endFigure()).getModellerClass();
                        endClass.addAssociation(startClass);
                    }
                    else {
                        ((JMenuItem)event.getSource()).setText("uni-directional");
                        JModellerClass startClass = ((ClassFigure)startFigure()).getModellerClass();
                        JModellerClass endClass = ((ClassFigure)endFigure()).getModellerClass();
                        endClass.removeAssociation(startClass);
                    }
                }
            });

        popupMenu.setLightWeightPopupEnabled(true);
        return popupMenu;
    }
```

Listing 7: The method `createPopupMenu` for an association line

In detail, the `handlePopupMenu` method of class `CustomSelectionTool` is triggered by a right mouse click. It checks if an association line has an attribute that is equivalent to the `Figure.POPUP_MENU`. If this is true then the method `showPopupMenu()` is called with the following parameters;

1. Figure - for which a popup menu should be displayed
2. x and y - coordinates where the popup menu should be displayed
3. comp - component which invoked the popup menu

This method displays a popup menu if the menu is associated with the Figure (the Figure's attributes are queried for `Figure.POPUP_MENU` which is used to indicate an association of a popup menu with the Figure) [9].

The class `AbstractAction` in above code listing (Line 3) is an anonymous class. It handles performed actions on the context menu of the association line. The method `actionPerformed()` from this class is invoked after the menu item is selected. The method tests the kind of connection of the line by getting the start or the end decoration. Selecting the menu item "aggregation" will invoke the method `isAggregation()` that tests whether an association is an aggregation or not. Regardless of the result, the method `setAggregation()` is called. It turns an association into an aggregation or vice versa. Whether an association is an aggregation or not is determined by an internal flag that turns an association into an aggregation. Now a new instance of `AggregationDecoration` with a default diamond size is created and the start decoration is set. And finally the selected menu item is changed to "no aggregation" or "aggregation" respectively.

3.1.2 Context menu for ClassFigure

A `ClassFigure` is a graphical representation for a class in a class diagram. A `ClassFigure` separates the graphical representation from the data model. A class has a class name, attributes and methods. Accordingly, a `ClassFigure` consists of other parts to edit the class names, attributes and methods respectively. [9] The user adds attributes and methods by using context menus.

A mouse listener handles mouse down events and gets the position of the last click inside the view. The event is delegated to the currently active tool "`UndoableTool`". `UndoableTool` invokes the method `mouseDown()` of the class `CreationTool`. `CreationTool` uses the Prototype design pattern (see section 2.2.5). Thus, the prototype figure which is used to create new figures of the same type by cloning the original prototype figure. It detects that the prototype figure is a `ClassFigure`. `CreationTool` invokes the method `clone()` of the class `GraphicalCompositeFigure` to clone and initialize the figure. The `GraphicalCompositeFigure` manages contained figures like the `CompositeFigure` does, but delegates its graphical presentation to another (graphical) figure which purpose it is to draw the container for all contained figures [9].

The Popup menu is created during initialization of a `ClassFigure`.

```
01 protected JPopupMenu createPopupMenu() {
02         JPopupMenu popupMenu = new JPopupMenu();
03         popupMenu.add(new AbstractAction("add attribute") {
04                 public void actionPerformed(ActionEvent event) {
05                     addAttribute("attribute");
06                 }
07             });
08         popupMenu.add(new AbstractAction("add method") {
09                 public void actionPerformed(ActionEvent event) {
10                     addMethod("method()");
11                 }
12             });
13 
14         popupMenu.setLightWeightPopupEnabled(true);
15         return popupMenu;
16     }
```

Listing 8: The method `createPopupMenu` for `ClassFigure`

The class `AbstractAction` in the listing 8 (line 3), a subclass of `javax.swing.AbstractAction`, is an anonymous class that has a method `actionPerformed()` (line 4). An instance of this class is created in the argument of the `add()` method (line 3). It appends a new menu item "add attribute" to the newly created menu. The second item "add method" is added to the menu by invoking the same method again.

The method `setAttribute` of the class `GraphicalCompositeFigure` sets a popup menu as an attribute to `ClassFigure`:

```
setAttribute(FigureAttributeConstant.getConstant(Figure.POPUP_MENU),
createPopupMenu());
```

The procedure to show a popup menu for `ClassFigure` is analogous to the one described for the association line.

3.2 Deletion of a ClassFigure

The deletion of a `ClassFigure` is an example of removing the composite figure. The command `Delete` removes the selected figures from the active drawing. Deletion of a `ClassFigure` removes all of its contained text figures: class name, attribute and method text figures. Note that if a `ClassFigure` is removed, all its incoming and outgoing connection lines are also deleted.

Situation: The user selects the `ClassFigure` Class4 and deletes it (see Figure 7).

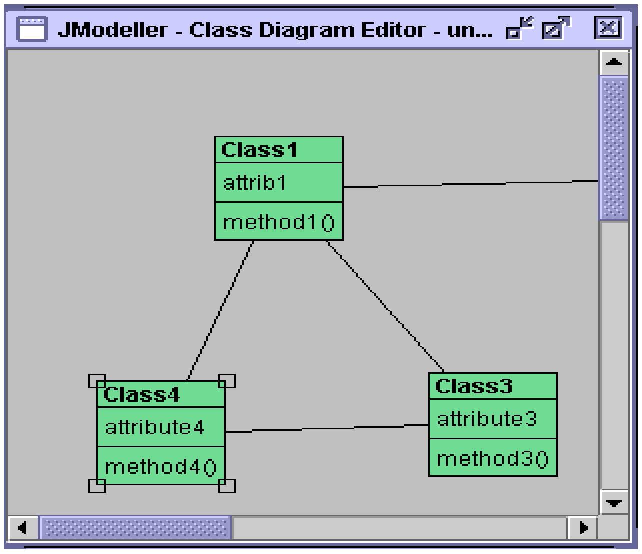


Figure 7: The selected `ClassFigure` for the deletion

The method `processKeyEvent` of class `java.awt.`Component is invoked after the button Delete in the keyboard is pressed. It processes the key event "Delete" occurring on the `ClassFigure` by dispatching them to any registered `KeyListener` objects. This method is called when key events are enabled for the component. Depending on the key event, the corresponding method of the class `StandardDrawingView,` in this case `keyPressed()` is called. It handles key down events by getting the integer `keyCode` for the `Delete` key on the keyboard. If the pressed key is `Delete` or `Backspace` then the command is tested for execution. A command is executable if the view is valid and at least one figure is selected in the current activated view.

To execute the command the method `execute()` of the class `UndoableCommand` is invoked. It takes care of finding the right method to execute the command. For the command Delete the method `execute()` of the class `DeleteCommand` (see Listing 9) is invoked.

```
1 public void execute() {
2     super.execute();
3     setUndoActivity(createUndoActivity());
4     getUndoActivity().setAffectedFigures(view().selection());
5     deleteFigures(getUndoActivity().getAffectedFigures());
6     view().checkDamage();
7   }
```

Listing 9: The method `execute()` of the class `DeleteCommand`

It gets an enumeration over the currently selected figures by invoking the method `selection()` of the class `StandardDrawingView` (Line 4). The method returns a reverse list of all selected figures, which is the parameter of the method `setAffectedFigures()`. It constructs an array of the selected figures.

The method `deleteFigures()` of the class `FigureTransferCommand` deletes the selection from the drawing. The enumeration of the dependent figures on the selected figure is found as listeners of the figure. The figure Class4 has two `AssociationLineConnection` listeners. Before the deletion of the `ClassFigure` it is tested whether a drawing contains a given figure. If it is true then the figure is deleted by invalidating it. After the figure Class4 is removed all of its dependent figures `AssociationLineConnection` are also removed.

3.3 Locators

Locators can be used to locate a position on a figure. They define where handles or connectors should be placed on a figure. The `Locator` interface is implemented by

`AbstractLocator`, which in turn defines two subclasses, `OffsetLocator` and `RelativeLocator`.

Relative Locator is used to represent positions relative to a figures boundary. Positions are specified as percentages of the figure (i.e. 50% along and 50% down will locate the centre of the figure). Relative Locator comes with a set of convenient static methods that specify the 'compass points' on a figure (i.e. `northWest()` specifies the top left hand corner of a figure etc.). [11]

Offset Locator is used to specify a position relative to another locator. They are often used in conjunction with relative locators for example; to specify a point a few pixels off of the centre of a figure. [11]

3.4 Handles

Handles are used to manipulate the figures on a drawing directly. The `AbstractHandle` class implements the `Handle` interface and provides default behavior for all handles in the framework.

JHotDraw predefines several types of handles: `ChangeConnectionHandle`, `ElbowHandle`, `LocatorHandle`, `TriangleRotationHandle` and `PolygonHandle`. The reuse handles across different types of figure are rare since Handles are created to be specific to the figure. Therefore developers have possibility to write their own handles for new or altered figures either by sub-classing `AbstractHandle` or one of the above classes.

The `BoxHandleKit` class has a set of methods to create Handles for the common locations on a figure's display box. The figures such as rectangle, ellipse, round rectangle, triangle and diamond contain handles at each corner as well as the north, south, east, west sides of the figure because the display box for all these figures is a rectangle. Positioning of a handle on a figure can be altered by overriding the `locate()` method. It returns a point around which the handle will be centered.

Handles can have different appearances, for example in the below picture there are four distinct styles of handles, an empty rectangle, a white rectangle, a green rectangle and a yellow circle. The different appearances can be used to denote different behaviors i.e. in this example the empty rectangles are null handles and perform no action other than to show selection. The white rectangles are the default handles in JHotDraw and provide resize functionality. The green rectangle in the top middle of the `TextFigure` allows the connector to be disconnected from the figure. Finally the yellow circles provide custom behavior specific to a figure i.e. on the `ElbowConnector` they allow each segment of the connector to be repositioned independently while on the `TextFigure` the single yellow circle changes the font size of the text.

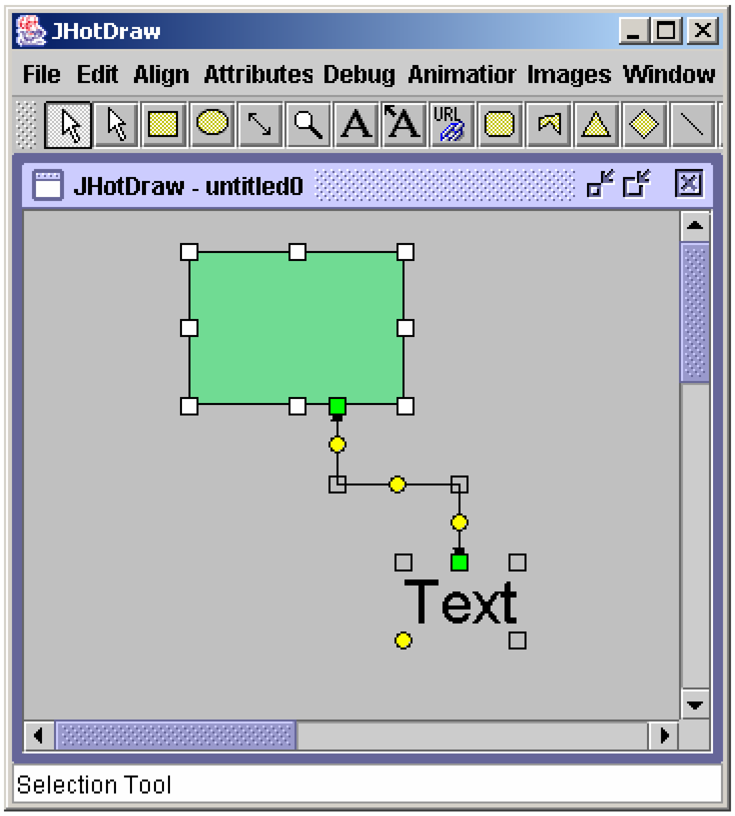


Figure 8: Handles on the figures

The method handles() is responsible for adding handles to the figures which is overridden for the particular figures. This method is called whenever the selected figure's handles are needed to be drawn on the DrawingView:

```
public HandleEnumeration handles() {
    List handles = CollectionsFactory.current().createList();
    BoxHandleKit.addHandles(this, handles);
    return new HandleEnumerator(handles);
  }
```

The HandleTracker class implements interactions with the handles of a Figure. Handles define three important methods invokeStart(), invokeStep() and invokeEnd(). Each of these methods are called by a user initiated event, invokeStart() is invoked whenever the user clicks the mouse down on a handle, invokeStep() whenever a user drags a handle and invokeEnd() whenever the user releases the mouse button. Every interaction with a handle will therefore invoke a sequence of method calls where invokeStart() is called at the beginning of the interaction, invokeStep() optionally may be called in the middle and invokeEnd() will be called when the interaction is over. This granularity across the interaction allows the developer to control how the handle responds to the user input. [11]

The method mousePressed() of the StardardDrawingView class handles mouse down events. The event is delegated to the currently active tool. First, it gets the position of the last click inside the view. Then, findHandle() finds a handle at the given coordinates. If there are no selected handles then it finds a figure at the given coordinates. The Drag tracker for the figure is created.

3.5 Connectors

Connectors are used to decide where to locate a connection point on a figure. A Connector knows its owning figure and can determine either the start or the endpoint of a given connection. A connector has a display box that describes the area of a figure for which it is responsible. A connector may not necessarily be visible. JHotDraw comes with presupplied connectors of various types. The `Connector` interface is implemented by `AbstractConnector`, which is extended by `ChopBoxConnector`, `LocatorConnector` and `ShortestDistanceConnector`.

When the start connection of the figure is found the connector is set by calling a `connectorAt(int, int)` method that returns a `ChopBoxConnector` by default. The `ChopBoxConnector` locates connection points by chopping the connection between the centers of the two figures at the display box [9].

The connection points are changed when the figure is moved. The connection line is always pointed to the center of the each figure. Depending on the center point of figure, the angle is calculated to other figure. The connection point may be anchored to the certain point of the figure's perimeter or its handles.

Figures can refuse to take part in connections by overriding the `canConnect()` method of the `LineConnection` class. In this case, this method returns false. It checks if two figures that we want to connect can be connected when the mouse is released on the other figure. Several rules for the connection between figures can be made, for example,

- We can connect rectangle with rectangle, and circle with circle;
- We cannot connect rectangle with circle.

Connectors belong to Figures and can be obtained by invoking a figures `connectorAt(int,int)` method. This method returns the connector at the given location defined by the two integers in the parameter list. Figures can have several different types of connector at different locations but usually they contain only one.

Connectors can be visible if a Handle for connections is created using the ConnectionH`andle` class. A connector can be made visible only when the method `draw()` is called. By default connectors are invisible.

4. XML Specification for JHotDraw

4.1 The user view of JHotDraw

The main purpose of JHotDraw framework is to create drawing editors that provides support for programs ranging from simple paint package style to more complex ones that have rules about how their elements can be used and altered. The framework provides support to create and edit user defined shapes or behavioral constraints in the editor and animation. The framework does not take any account of semantic knowledge of the underlying problem domain. More graphic editors such Rational Rose, AutoFocus and Flowchart editor are overviewed in the work of Gerin Klein [5].

The applications created by the framework are not in 'bitmap' format. It maintains a collection of shapes that are separate from each other and can be manipulated as separate entities. Even thought the figures are defined as rectangular, developers are free to define their own shapes inside that rectangle which is visible to the user. It also has many built-in commonly used figures but it also gives room for developers to create custom figures depending on their own application area.

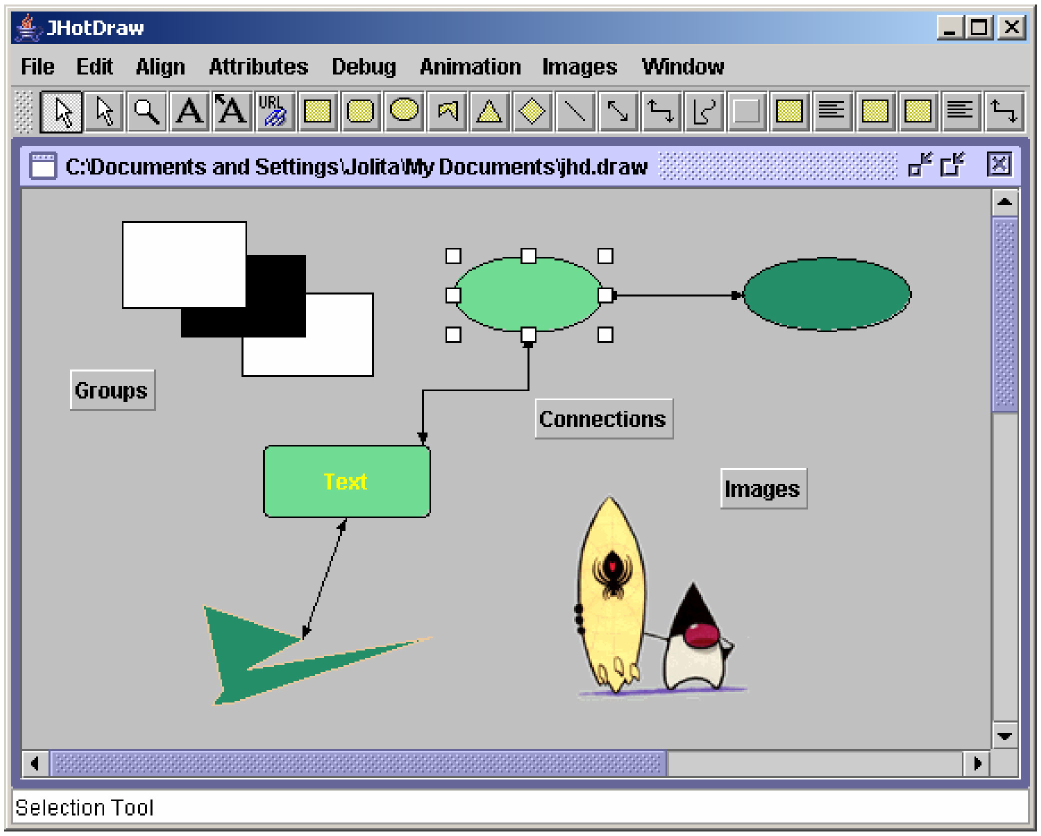

JHotDraw
File Edit Align Attributes Debug Animation Images Window
C:\Documents and Settings\Jolita\My Documents\jhd.draw
Groups
Connections
Text
Images
Selection Tool

Figure 9: A screenshot of the JHotDraw JavaDraw application

The tool bar contains icons for the different kinds of tools available to the application. A toolbar palette can be switched from horizontal to vertical mode and vice versa by dragging it up and down. At any point in time, exactly one tool of all toolbars is selected, which appears pushed down. By default, the selection tool is selected, whenever the work with the current tool is finished.

The status bar of the application is the white area running along the bottom of the window. It displays a Tool Tip, a short description of the tool if you move the mouse pointer over a tool button. It displays information about the current state of the application (i.e. currently selected tool).

The menu bar resides at the top of the application window and consists of menus which contain partially context sensitive menu items. They are grouped by functionality. The default menu bar contains menus for File, Edit, Align, Attributes and Debug operations.

Any other application may add, alter or remove menus or menu items to reflect its requirements.

The user interacts with the diagram editor by mouse and keyboard operations. Objects are selected by clicking on them and can be manipulated using the mouse, keyboard (Delete command) or the menu commands. There are no help and context menus (popup menus) on the figures implemented in the JavaDraw application.

4.2 What is an XML Dialect?

Any "flavor" of XML defined by a DTD that is designed to support a specialized purpose, such as BIOML (BIOpolymer Markup Language), CML (Chemical Markup Language), MathML, CDF, TalkML (an experimental XML for voice browsers), XFRML, etc. [6]

XML is a simple, very flexible text format derived from SGML. XML is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [15]. XML does not define a fixed set of mark-up tags nor does it give tags a certain meaning. This is left to specific applications and application domains.

XML itself is extensible in the sense, that it can be used to define any type of elements which would be a great advantage to use it with JHotDraw. In general, a series of element and attribute types created for some particular domain are said to form a “dialect” of XML. It is taken into consideration what data should be included and what structure of elements and attributes is used to represent those dialects. In order to create dialects, XML has the concept of a document type definition (DTD) or XML Schema Definition Language. Each DTD or XML Schema is a specification that indicates the structure of documents that conform to that dialect.

4.3 XML Dialect for JHotDraw

As explained in section 1.1, to extend the JHotDraw framework an XML dialect can be defined to represent those extensions. It could be used XSLT to read an XML document defined by XML Schema and generate a text document which could look like a Java program. The actual implementation of the task of this project is explained further.

The JHotDraw graphical editor has the ability to open and load a document. The editor has an internal storage format with the file extension "draw" and serialization storage format with the file extension "ser". The implementation is to extend this editor to process a XML document defined by a XML dialect as input format with a new file extension "xml". It is used to open an XML input file which has a number of figures defined in the drawing. Actually, it contains almost the same information as a "draw" file, but in XML format. The structure of the "draw" file you can find attached in the Appendix 1.

An XML Schema specifies the syntax for JHotDraw XML input document. The XML Schema defines only the basic elements of the JHotDraw diagram editor. It specifies that the root element Model contains Entities and Allowed-operations (see Appendix 2). The XML element Entities may contain an arbitrary number of graphical figures elements i.e. Rectangle, Ellipse, Circle. For example, each Rectangle element has to have one obligatory attribute ID, which is unique for all Rectangle elements, and a sequence of three sub-elements (StartPoint, Width, Height), which determine the position of the specified rectangle in the drawing view. Two elements such as FillColor, FrameColor are optional. If they are not specified the default framework colors are used. The XML Schema also specifies a rule for the connection of figures. We cannot connect two different type figures, i.e. a rectangle and an ellipse. The connection of figures in XML document indicates the element Connect which is a sub-element of Allowed-operations. This is only a partial XML specification for JHotDraw graphical editor. It defines the geometrical view of the drawing. The following example demonstrates a sample XML document against defined XML Schema:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSPY v2004 rel. 2 U (http://www.xmlspy.com)-->
<Model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="D:\SA\XML\Model.xsd">
    <Entities>
        <Rectangle ID="rec1">
            <StartPoint x="57" y="42"></StartPoint>
            <Height>98</Height>
            <Width>72</Width>
        </Rectangle>

        <Rectangle ID="rec2">
            <StartPoint x="260" y="118"></StartPoint>
            <Height>127</Height>
            <Width>78</Width>
        </Rectangle>
    </Entities>
    <Allowed-operations>
        <Connect source="rec1" target="rec2"/>
        </Allowed-operations>
</Model>
```

Listing 10: A sample XML input document

The corresponding output drawing generated using the above listed sample XML document looks as follows:

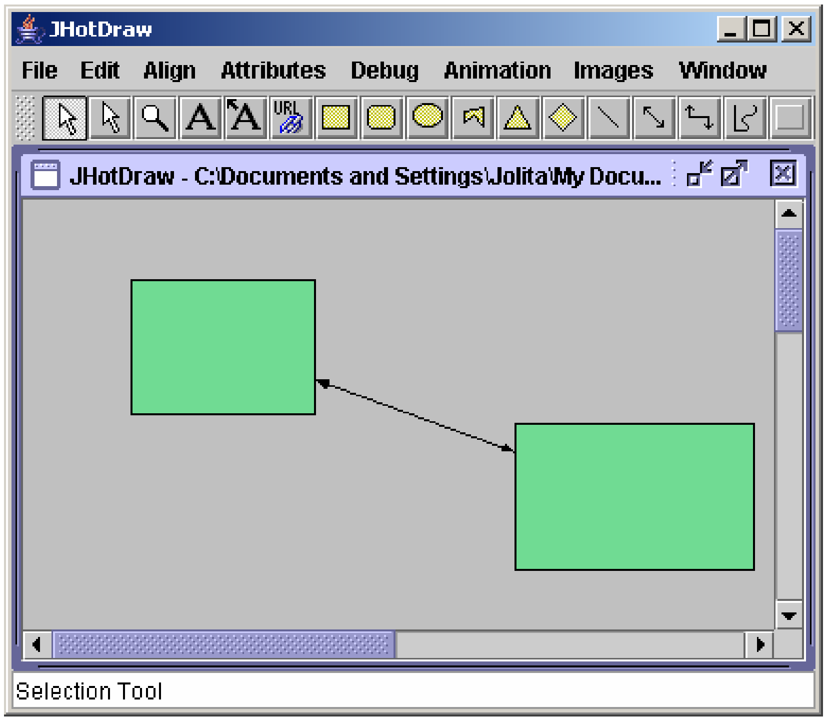

JHotDraw
File Edit Align Attributes Debug Animation Images Window
JHotDraw - C:\Documents and Settings\Jolita\My Docu...
Selection Tool

Figure 10: Generated drawing

The XML Schema definitions are translated into Java classes generated using XMLSPY built-in code generator. It automatically generates class definitions corresponding to all declared elements or complex types which redefine any complex type in the XML Schema. Each element and complex type in XML Schema document corresponds to a generated Java class. The JHotDraw diagram editor then consists of these generated classes together with its supported classes in JHotDraw class library.

An XML document can be read and parsed using DOM data structures or the generated Java code of defined XML Schema document. In this work both of the possibilities are used in the mixture. This is shown in the next code to read a string from an XML input document:

```
public String readString() throws IOException, Exception {
          if(!mod.hasEntities()) {
             System.out.println( "There is no figures specified in the XML docu
ment" );
             return null;
          }
              if (number == -1){
                  number = number +1;
                  return "CH.ifa.draw.samples.javadraw.BouncingDrawing";
          }
        try{
            if (childNo < getNoOfEntities()) {
                NodeList nl = mod.getEntities().getDomNode().getChildNodes();

                if (nl.item(childNo).getNodeType() == 1) {
                    childNo = childNo +1;
                    return "CH.ifa.draw.samples.javadraw.AnimationDecorator";
                  }
                else {
                    int noFigure = childNo -1;
                    String entity = nl.item(noFigure).getNodeName();
                    if (entity == "Rectangle") {
                        return "CH.ifa.draw.figures.RectangleFigure";
                    }
                    if (entity == "Ellipse") {
                        return "CH.ifa.draw.figures.EllipseFigure";
                    }
                    ...
                }
            }
```

The process of creating a drawing in JHotDraw works in the composite figure principle. The BouncingDrawing is a container for all figures in the drawing. After proofing that the XML document has at least one defined figure, the BouncingDrawing is returned although this is not specified in the XML file. Then the number of figures is read by the method `getNoOfEntities()` and each specified figure is restored.

An `AnimationDecorator` decorates the figures with an animation (see section 2.6 The Decorator design pattern). The encapsulated figure reads the StartPoint, Width and other attributes from the XML document. To restore the drawing from the "draw" file the `AnimationDecorator` reads the velocity value from the file. In contrast the XML document does not specify this parameter and for simplicity it is set as default to 4 by the framework. The above described procedure shows how the drawing could be painted using XML input specification.

5. Conclusions

This project report provides a description of design patterns used in the JHotDraw framework, the JHotDraw graphical editor's internal working and the XML-based extension of it.

Design patterns are considered an excellent way of achieving highly reusable software architecture. Documenting the framework architecture using patterns helps to understand the framework in a perspicuous language to other developers and coordinate the process of learning the framework's features. Patterns separate and prevent classes from knowing much about one another. JHotDraw is a sophisticated framework that is composed of many design patterns, such as Composite, Template method, Factory method, Decorator, Prototype, and Command as described in details in Chapter 2, and other patterns. Design patterns makes a framework look less complex than it actually is by having a higher level of abstraction, and helps to resolve how to customize and adapt the framework to suit the needs of the developer.

After analyzing the design patterns used in the JHotDraw framework and the procedure to create particular components such as figures, handles, connectors, a XML dialect was specified to extend the framework. It is an XML Schema document which defines the structure of XML input file to generate the drawing. Currently specified XML dialect defines the configuration only for the basic figures in the framework. It could be extended to support the full functionality of the editor, transformed into Java source code using XSLT and using that code to generate the editor.

Further work can proceed in one of several directions: (1) supporting "multi-platform" generation (e.g. targeting JHotDraw and Visio with the same spec), or (2) applying the underlying concepts to forms-based UIs, thus achieving model-based GUI development, or (3) MDA on the server side, as exemplified by AndroMDA or XDoclet.

Bibliography

[1] http://www.jhotdraw.org/

[2] Chapter 27: Experiences with Semantic Graphics Framework. Andreas Rosel and Karin Erni Andreas Rösel. In Implementing Application Frameworks: Object-oriented frameworks at work. ISBN 0471252018

[3] Kaiser, W. (2001). Become a programming Picasso with JHotDraw. Retrieved January 12, 2003 from http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html

[4] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. Design patterns elements of reusable object-oriented software. Boston, MA: Addison Wesley, 1995.

[5] G. Klein. Generating graphical editors for graph-like datastructures. Master's thesis, Chair of Computer Science II, Technische Universität München, 1999. http://www.doclsf.de/grace/grace.ps.gz

[6] XML Glossary http://www.cise.ufl.edu/~bjyu/bk/Proposal/glossary.htm

[7] JHotDraw Applications. Available at: http://www.jhotdraw.org/survey/applications.html

[8] Kaiser, W. Letters to the Editors. Available at: http://www.javaworld.com/javaworld/jw-03-2001/jw-0323-letters.html

[9] JHotDraw source code. Available at: http://jhotdraw.sourceforge.net/

[10] D. I. Tanase. Documenting frameworks using pattern languages. Master Thesis, University of Northern Iowa, December 2003. Available at: http://www.cs.uni.edu/~wallingf/miscellaneous/student-papers/tanase-thesis.pdf

[11] D. Kirk. JHotDraw Pattern Language. Department of Computer and Information Sciences University of Strathclyde, Glasgow. Available at: http://softarch.cis.strath.ac.uk/PLJHD/Patterns/JHDDomainOverview.html

[12] B. Kurniawan. Using the Decorator Pattern, May 2003. Available at: http://www.onjava.com/pub/a/onjava/2003/02/05/decorator.html

[13] Interface Cloneable http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Cloneable.html

[14] James W.Cooper. The Design Patterns. October 2, 1998.

[15] http://www.w3.org/XML/

Appendix 1: “draw” file

The content of the sample “draw” file which defines connected rectangle and ellipse figures with particular coordinates.

```
CH.ifa.draw.samples.javadraw.BouncingDrawing 3
   CH.ifa.draw.samples.javadraw.AnimationDecorator
      CH.ifa.draw.figures.RectangleFigure "no_attributes" 72 37 110 88  4 4
   CH.ifa.draw.samples.javadraw.AnimationDecorator
      CH.ifa.draw.figures.EllipseFigure "no_attributes" 301 112 114 127  4 4
   CH.ifa.draw.figures.LineConnection 2 182 103 305 154
      CH.ifa.draw.figures.ArrowTip 0.4 8.0 8.0 "noFillColor" "noFrameColor"
      CH.ifa.draw.figures.ArrowTip 0.4 8.0 8.0 "noFillColor" "noFrameColor"  0 0 0
      CH.ifa.draw.standard.ChopBoxConnector REF 2
      CH.ifa.draw.figures.ChopEllipseConnector REF 4
```

Appendix 2: XML Schema file

XML Schema document Model.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v2004 rel. 2 U (http://www.xmlspy.com) by Jolita (Home) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:complexType name="RectangleType">
        <xs:choice minOccurs="3" maxOccurs="unbounded">
            <xs:element name="FillColor" minOccurs="0"/>
            <xs:element name="FrameColor" minOccurs="0"/>
            <xs:element ref="StartPoint"/>
            <xs:element name="Width" type="xs:int"/>
            <xs:element name="Height" type="xs:int"/>
        </xs:choice>
    </xs:complexType>
    <xs:element name="StartPoint">
        <xs:complexType>
            <xs:attribute name="x" type="xs:int" use="required"/>
            <xs:attribute name="y" type="xs:int" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="EntitiesType">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Rectangle" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:complexContent>
                        <xs:extension base="RectangleType">
                            <xs:attribute name="ID" type="xs:string" use="required"/>
                        </xs:extension>
                    </xs:complexContent>
                </xs:complexType>
            </xs:element>
            <xs:element name="Ellipse" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:complexContent>
                        <xs:extension base="RectangleType">
                            <xs:attribute name="ID" type="xs:NCName" use="required"/>
                        </xs:extension>
                    </xs:complexContent>
                </xs:complexType>
            </xs:element>
        </xs:choice>
    </xs:complexType>
    <xs:element name="Entities" type="EntitiesType"/>
    <xs:complexType name="ModelType">
        <xs:sequence>
            <xs:element ref="Entities"/>
            <xs:element name="Allowed-operations">
                <xs:complexType>
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="Connect">
                            <xs:complexType>
                                <xs:attribute name="source" type="xs:string" use="required"/>
                                <xs:attribute name="target" type="xs:string" use="required"/>
                            </xs:complexType>
                        </xs:element>
                    </xs:choice>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Model" type="ModelType">
        <xs:key name="Entity">
            <xs:selector xpath="./Entities/*"/>
            <xs:field xpath="@ID"/>
        </xs:key>
        <xs:keyref name="sourceRef" refer="Entity">
            <xs:selector xpath="./Allowed-Operations/connect"/>
            <xs:field xpath="@source"/>
        </xs:keyref>
        <xs:keyref name="targetRef" refer="Entity">
            <xs:selector xpath="./Allowed-Operations/connect"/>
            <xs:field xpath="@target"/>
        </xs:keyref>
    </xs:element>
</xs:schema>
```