Design Document

Group: Canvas

Software Engineering Project
Department of Computer Science
University of Helsinki

05/08/2005
Document version: 01

Project Members:
Duku-Kaakyire Michael,
Karppinen Tony Henrik,
Lamsal Pragya,
Välimäki Niko Petteri

Instructor:
Raine Kauppinen

Customer:
Inkeri Verkamo

Supervisor:
Juha Taina

Table of Contents

1. Introduction....4
2. Purpose of this document....4
3. Architectural design....4
4. Class design....7
4.1 Model, view and controller classes....8
4.2 Element related classes....10
4.3 Other classes....11
5. User interface....13
5.1 The Menubar....13
5.2 The Toolbar....14
5.3 The Diagram list....14
5.4. The Diagram view....14
5.5. The Status bar....15
6. Use cases and object collaboration....15
6.1 Basic functions....15
6.2 Diagram drawing....16
6.3 Element editing....24
7. Maintenance....25
7.4 Extendibility of the application....25

1. Introduction

This is a project for design and development of a diagram-drawing software on the course named Software Engineering Project at the department of Computer Science of the University of Helsinki.

The aim of this project is to develop a functional application that can realize the core requirements of the customer, while leaving a room for extension of the application in the future by other developers.

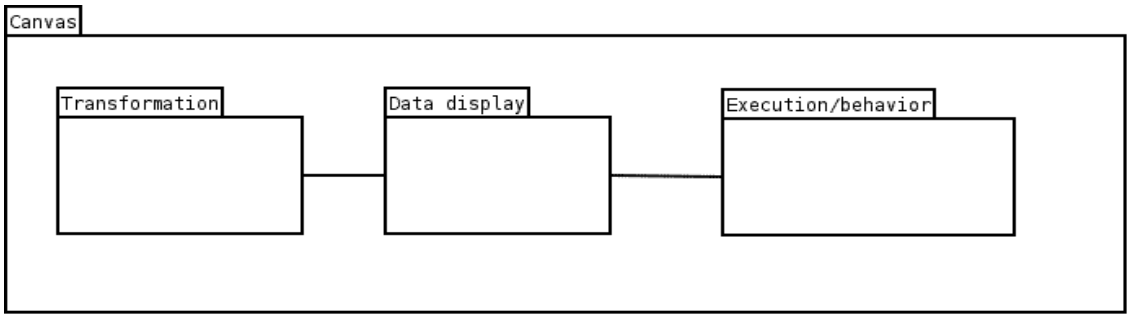
The homepage of this project group is http://www.cs.helsinki.fi/group/canvas/.

2. Purpose of this document

This document will describe the architecture of the application with such detail that one can implement the application by resting only to this document. This document is written based on the Requirements document. Class structure and object collaboration of the application are derived from user requirements, system requirements and use cases listed in the Requirements document. The requirements document can be found from the project group's homepage.

This document will serve also as an implementation document and it will be updated during the implementation phase. This version of the document has been frozen as an design document at the end of the design phase. The updated version and the frozen design document can be both found from the project group's homepage.

3. Architectural design

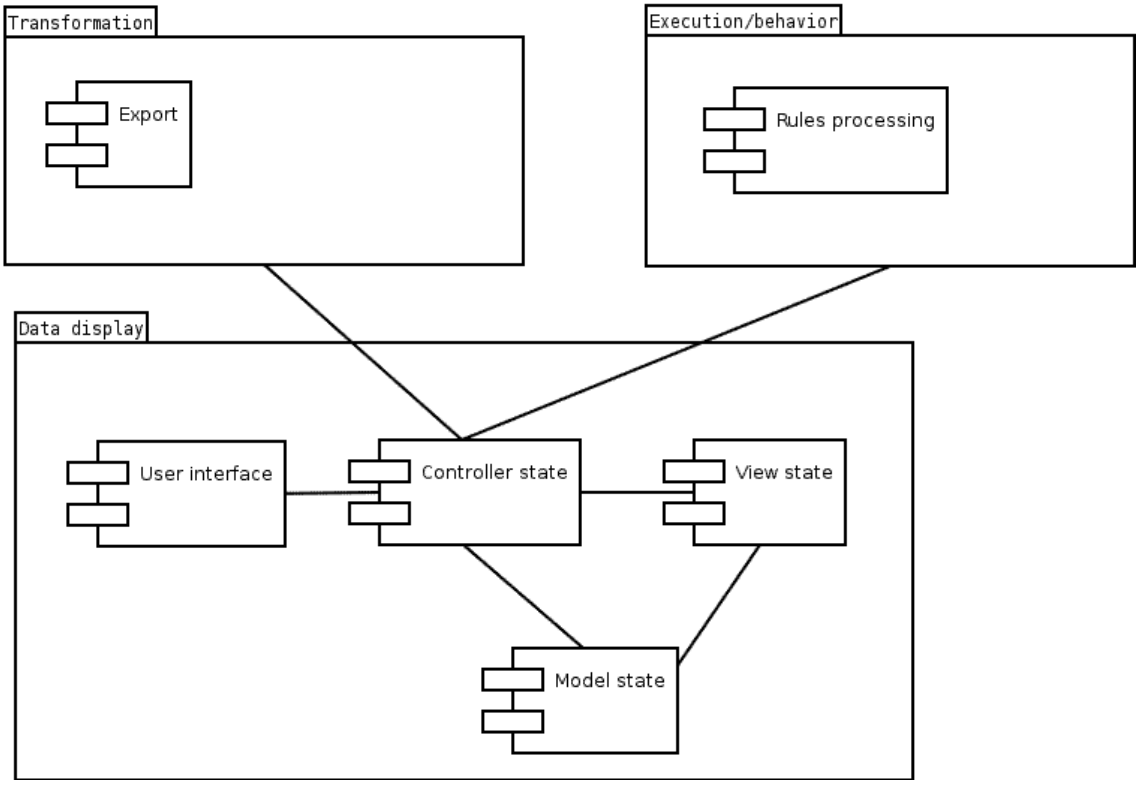


Picture 1: The three subsystems of the application

System is divided into three subsystems. The subsystems are Transformation, Execution/behavior and Data display as shown in picture 1. In this project we will mostly

be concentrating on the Data display subsystem, and just provide an extendable interface for future development of the Transformation and Execution/behavior subsystems.

The Transformation subsystem contains all the export functionality. The Execution/behavior subsystem is used for processing rules associated with diagrams or elements.



Picture 2: The Data display subsystem

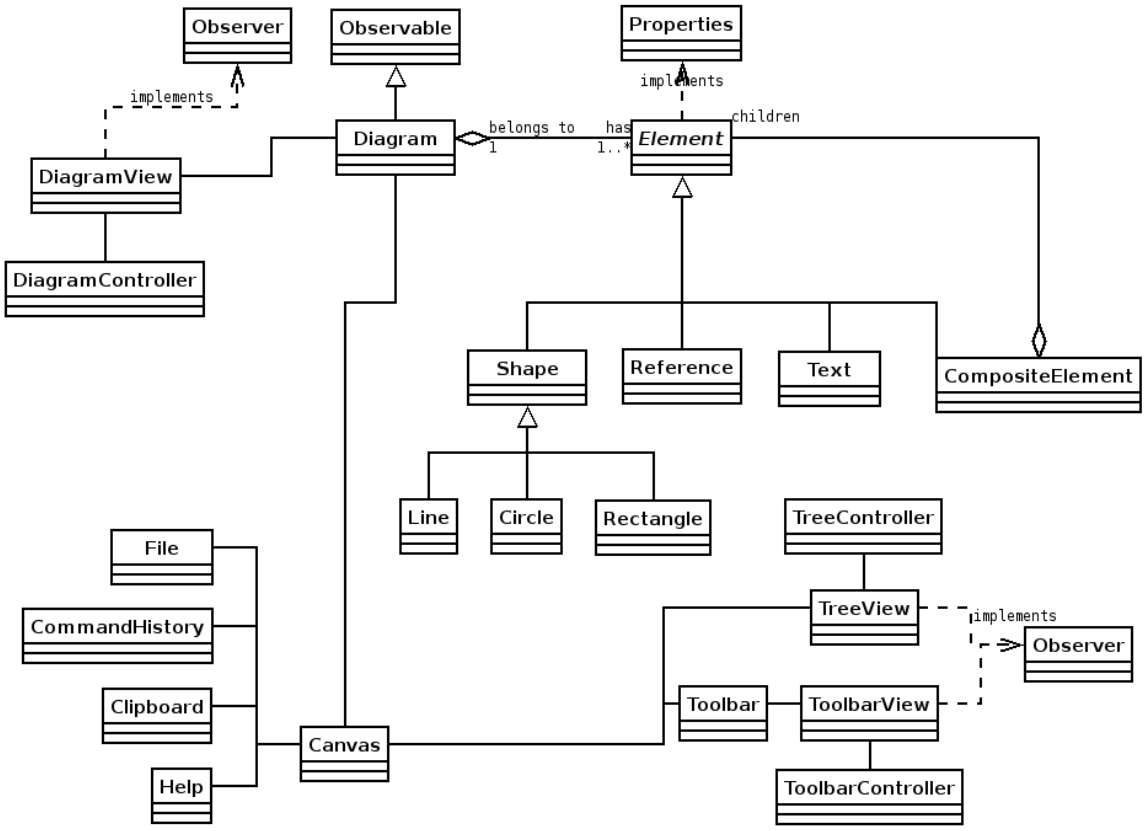
The Data display subsystem is divided into modules that are User interface, Controller state, View state and Model state as shown in picture 2. The User interface module handles all user interaction and takes care of the graphical user interface. The Controller state module handles all actions requested by the User interface module.

The View state module draws diagrams according to the diagram data and update requests from the Model state module. The Model state module stores the actual diagram data and provides an interface for reading and editing diagram information.

Interfaces to the Transformation and Execution/behavior subsystems are in the Controller state module.

4. Class design

Picture 3 shows the class diagram of the Data display subsystem. Detailed description about attributes and methods are also provided for the core classes.



Picture 3: Class diagram of the Data display subsystem

Model-view-controller (MVC) architecture is used with implementation of graphical user interface in Canvas. Table 1 shows how the classes are divided.

Model	**View**	**Controller (inside the view)**
Canvas	Canvas	Canvas
Diagram	DiagramView	DiagramController
	TreeView	TreeController
Toolbar	ToolbarView	ToolbarController

Table 1: Model-view-controller architecture

The view implements the visual display of the model and there can be several different views of the same model: E.g. DiagramView and TreeView in Canvas. Anytime the model is changed, each view of the model is notified so that it can change the visual

representation of the model on the screen. All the view classes implement the Observer interface.

The model is the information that the application is manipulating. The model notifies each view of the changes in the model data. All the model classes extend the Observable class.

The controller receives all the input events from the user and translates them into possible changes on the model. The controller communicates with the view to determine what object is being manipulated. As there is no need for several different controllers for one single view and the controller and the view are so tightly intertwined controllers are inner classes of the views in Canvas.

All is put together by creating the model first, then the view and a reference is put to the model as a member variable of the view. Controllers are created inside the view.

4.1 Model, view and controller classes

This section describes MVC classes Diagram, DiagramController, DiagramView, TreeView, TreeController, Toolbar, ToolbarController and ToolbarView.

Diagram

Diagram class holds the diagram data and provides accessor methods for modifying the diagram. It holds references to all elements in a diagram.

```
elements
addElement()
getElements()
```

DiagramController

This class handles all the events associated with a diagram in the graphical user interface.

DiagramView

DiagramView class represents a drawing area in an internal frame. It handles repainting diagrams as necessary via Observer interface.

TreeView

TreeView provides a text based view of all open diagrams in one window formatted as a tree. All the open diagrams ("documents") are connected to the root and Elements and included Diagrams are the nodes of the tree.

Treeview is updated via Observer interface when a Diagram is loaded, created or modified.

collapse()
collapseAll()
expand()
expandAll()
update()
addNode()
removeNode()
setProperties()

TreeController

TreeController is a controller class for the tree view of the diagram.

TreeController is created and shown when a Diagram is loaded or a new Diagram is created. TreeController listens to events from the graphical user interface.

Canvas

This is the application level class that is instantiated when the application starts. It handles initialization of other classes needed to run the program and all the required GUI classes. It is also a controller that handles the application level functions. This is the main controller of that program that is connected to all other classes.

ToolbarView

This is the class that handles the layout of the toolbar. Toolbar contains all the tools needed to draw the diagrams. This class is instantiated with the generic drawing tools when the application starts.

toolType (class)
id (?)

ToolbarController

Toolbar controller handles the events related to the ToolbarView class. It is called when something is done with the toolbar. For example when an element selection is made.

4.2 Element related classes

Element class

Element class is an abstract class that provides an interface for build in elements and for composing of elements. An element can be a single element or a composite of several composite elements and single elements.

```
// Attributes
children
// Methods
addElement()
removeElement()
getChildren()
paintElement()
```

CompositeElement

CompositeElement class is a composite of single elements or composite elements. The CompositeElement class contains information only about its child elements. When an operation is called for a composite element, it will call all the children with the given operation.

This class is the only class that implements methods addElement(), removeElement() and getChildren(). Calling these methods from any other subclass of Element will result in runtime exception.

Shape

Shape class represents a parent class for basic shapes, which are a line, a circle and a rectangle. Line, Circle and Rectangle classes are subclasses of Shape. They implement the actual functionality of each basic shape.

Reference

This is the class that holds references to external files or diagrams.

Text

Text class represents an element containing text. Text has special properties like font and font size. Text can also be divided over multiple lines.

font
fontSize
setText()
getText()

Properties

This is the class that defines a common properties interface for all the elements.

4.3 Other classes

Help

This class is responsible for showing the built-in help system for the user. The help system basically includes the same information as the manual for the application does and this is opened in new window. Help system is provided using the JavaHelp API.

Instance is created by the Canvas class when the Help command is selected from menu. The Canvas class calls the methods of Help.

ExportEPS

This class takes care of exporting the diagram to an EPS file. The class gets the diagram input as parameter from the Canvas class and outputs an EPS file once it is done with the conversion.

Instance is created by the Canvas class when the export functionality is used, the Canvas class also calls the methods of ExportEPS.

Clipboard

Clipboard is used for copy/paste actions of the application. Clipboard can contain both graphical data (element(s)) and text.

Instance is created by the Canvas class while the application is starting, methods are called by the Canvas class.

getContents
setContents

File

File class is responsible for the file actions (save, save as, load) of the application.

Instance of a File class is created when the save or load command is selected. File is called from the Canvas class.

loadDiagram
saveDiagram
getName
getPath
exists

CommandHistory

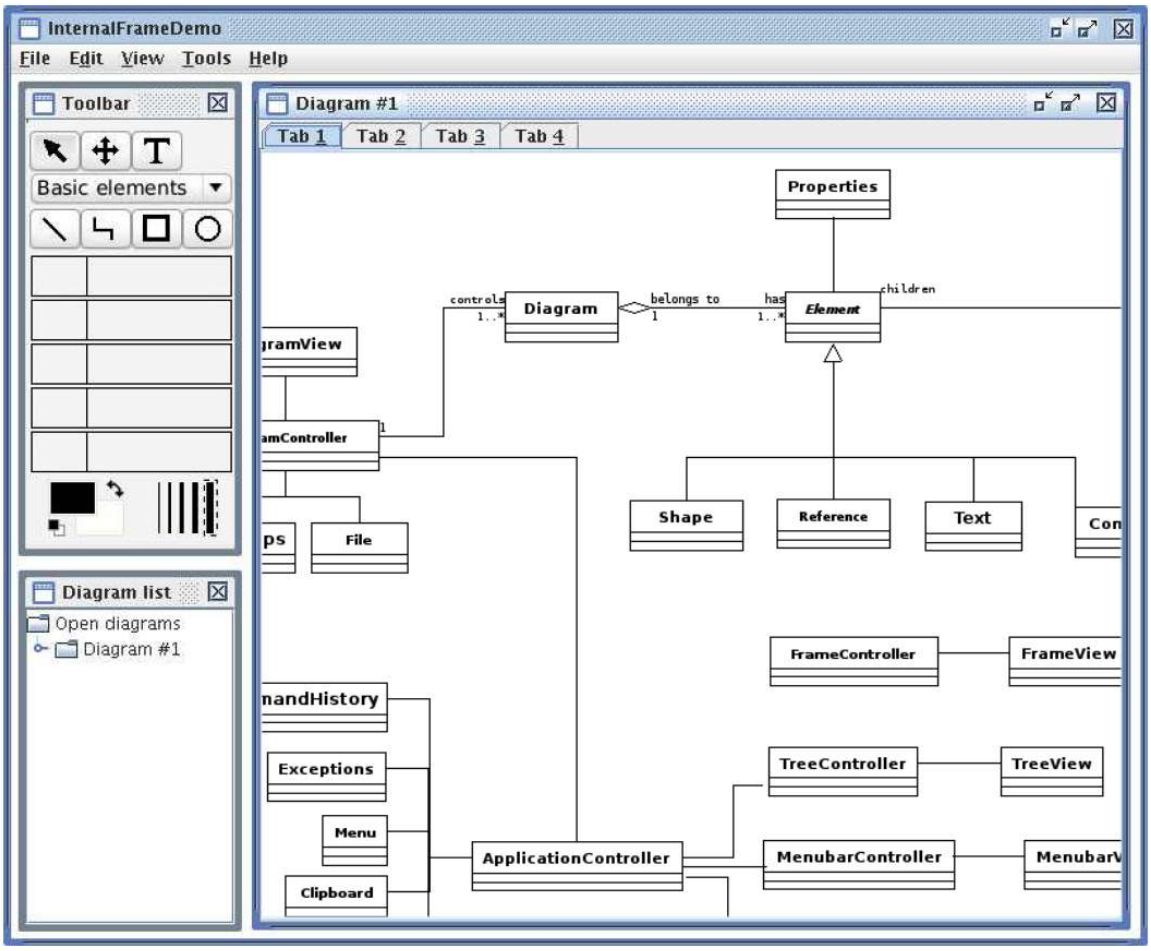
CommandHistory keeps track of actions performed in the application. From the user point of view this provides functionality of undo and redo. CommandHistory keeps track of history of maxSize steps.

Instance is created by the Canvas class while the program is started. The Canvas class calls the methods of CommandHistory.

maxSize
currentSize
history

5. User interface

An example of the application's graphical user interface is shown in picture 4.



Picture 4: The graphical user interface

The application has a main window on which combines all other elements. The main window consists of five main elements - the fixed elements are the menubar on the top of the window and the Status bar on the bottom of the screen. Floating elements are Toolbar, Diagram list and the open Diagram views. It is possible to show and hide the Toolbar and the Diagram list from the menubar. Open diagram views can be resized, minimized and maximized. It is possible to resize also the Diagram list and the Toolbar.

5.1 The Menubar

The Menubar is accessible and visible all the time and on the top level it contains the following items: File, Edit, View, Element, Tools and Help. File includes the basic file operations such as New, Open and Save as well as Export and Quit.

Edit includes the commands to copy, cut and paste the elements in diagrams and from View it is possible to hide some of the UI elements.

Element provides the tools to editing existing and combining the elements.

For more detailed information about the main menu, please see the User manual that can be found from the project group's homepage.

5.2 The Toolbar

The Toolbar consists of Select button, Move button, Text button, diagram type dropdown list, list of elements in the selected diagram type, color picker and line width selector.

Select is the default tool and will be active when the program is started. With select it is possible to select one or more elements in the diagram as well as move them. Moving the viewport can be done with the Move button. Text can be added to a diagram with the Text button.

Diagram type list includes only elements which belong to the selected diagram type, in other words the list of elements depends on the selected diagram type.

5.3 The Diagram list

The Diagram list is a tree representation of the current state of the application. On the top level of the diagram list are the open diagrams. Included diagrams are the nodes, and elements are the nodes or leaves in this view. It is possible to open and close branches which makes the navigation easy on this view.

By pressing the second mouse button on any of these elements will bring a diagram list specific menu. This menu includes the same commands as in the Diagram view.

5.4. The Diagram view

The Diagram view is the graphical representation of the diagram. One diagram view window represents a physical document, and the diagram view window is divided on several tabs, first tab represents the root diagram of the document and the other included diagrams are divided on the following tabs. Tabs are created and removed automatically as new diagrams are created and destroyed.

This is the view where all the drawing takes place. Some of the tools on the Toolbar (move, text, element) are used only on this this view, while some (colour, line width) can be used together with the diagram list as well.

5.5. The Status bar

The Status bar is located on the bottom of the main window and it contains various information about the state of the application. See the User manual for details. Information on the status bar is in the following format <diagram name> - <modified> - <message>.

6. Use cases and object collaboration

This paragraph will list use cases and describe object collaboration needed in each use case. There is also an example and error handling descriptions for every use case.

6.1 Basic functions

Opening a diagram from a file

Use case: The user selects the correct diagram file from a file listing or types in the correct path and filename. The program checks if the file is in the valid diagram format and loads the diagram to the program memory for viewing and processing. The program would also check through any syntactic rules defined to the diagram.

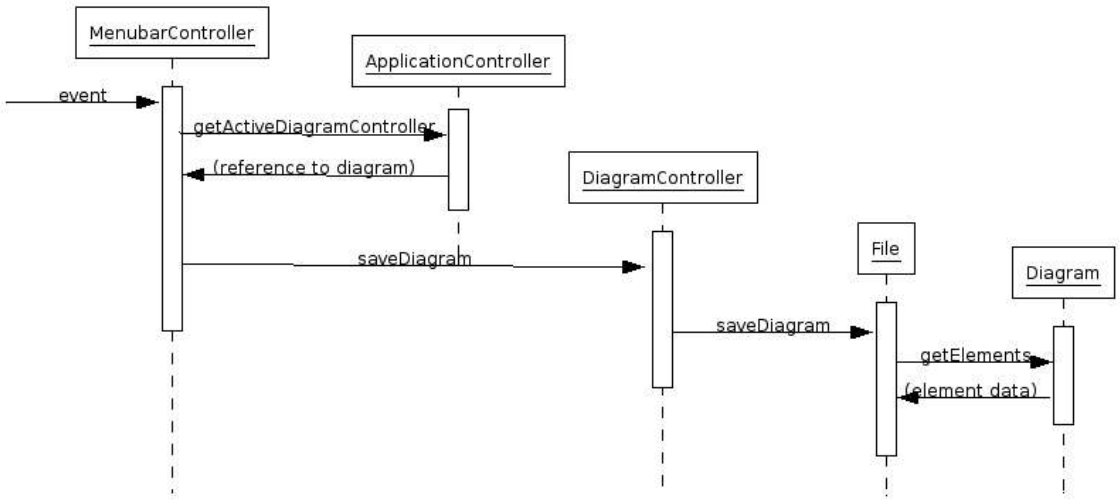
Example: The user opens a diagram from the file MyDiagram.cnv for editing.

Object collaboration: The class Canvas gets an event from MenubarController for opening an existing file. The application controller calls an openFile -method of File class. The openFile method takes the name of the file as an input parameter and opens that file for use by the user.

Saving and exporting to a file

Use case: The user chooses to either export or save the current diagram. When saving a file, the user must give a filename for storing the diagram data. If the diagram is opened from an existing file, it can be stored over the old file. Exporting can be done in encapsulated postscript format. When exporting a diagram, the user must give a filename. The user must be notified before writing over existing files. The program also handles writing of the actual diagram data to the file.

Example: The user saves the current open diagram to the file MyDiagram.cnv.



Picture 5: Object collaboration diagram

Object collaboration: The Canvas class gets an event from the system. Then it creates an instance of File class, and activates File's saveDiagram -method with a reference to the diagram to be saved. SaveDiagram -method serializes Diagram -instance including all the diagram's elements and saves the serialized data to the user assigned file.

Error handling:

Closing a diagram

Use case: The user chooses to close the diagram. Diagram is closed by closing all open views of the diagram. When closing the last view of the diagram, user must be notified for unsaved changes. All open diagrams are closed when the application is closed.

Example: The user closes the active diagram.

Object collaboration: The Canvas class gets an event from the system and checks if the diagram has been saved. If it is saved, it closes the diagram, otherwise it asks the user whether to save the diagram or not by using a message box. If the user clicks yes, it calls the saveDiagram method with a reference to the diagram to be saved. If the user clicks no, it simply removes diagram from memory and discards any modification.

6.2 Diagram drawing

Inserting a new element into a diagram

Use case: The user has to select the correct element to be inserted and then draw the element into the diagram. Drawing technique depends on the element. E.g. a rectangle is drawn by clicking the two corners of the rectangle. The program checks if the inserted element meets the syntactic requirements. If the element is not syntactically valid, it cannot be inserted into the diagram and the user would be notified.

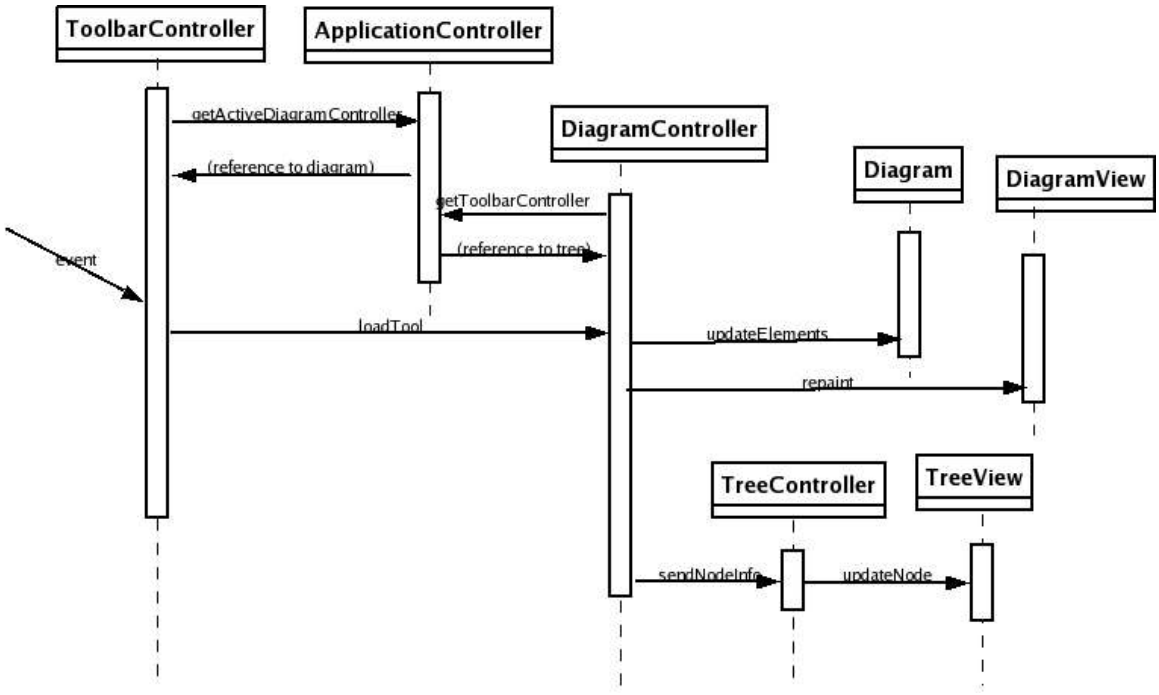
Example: The user selects a circle element. He draws it by first selecting the center point and then specifying the radius.

Object collaboration: ToolbarController gets an event from the system when a user clicks on a tool in the toolbar in order to insert into the diagram. ToolbarController stores the information about which tool or element is currently selected. When the user draws the element DiagramController gets the event. The DiagramController creates an instance of the element and inserts the element into the diagram. The Diagram class notifies observers.

Connecting elements together

Use case: The user selects a connection line element and draws the connection by selecting elements to be connected together The program makes syntactic checks for the connection line and the connected elements.

Example: The user draws two rectangles and connects them with a straight line.



Picture 6: Object collaboration diagram

Object collaboration: DiagramController gets the event and asks ToolBar for currently selected tool. Diagram is updated using the connection elements and the connecting element. The Diagram class notifies observers.

Editing an element

Use case: The user selects an element from the diagram to be edited. The selected element can be modified by resizing, moving or changing the properties of the element. The program retains connections between elements.

Example: The user writes a few lines of text into a rectangle.

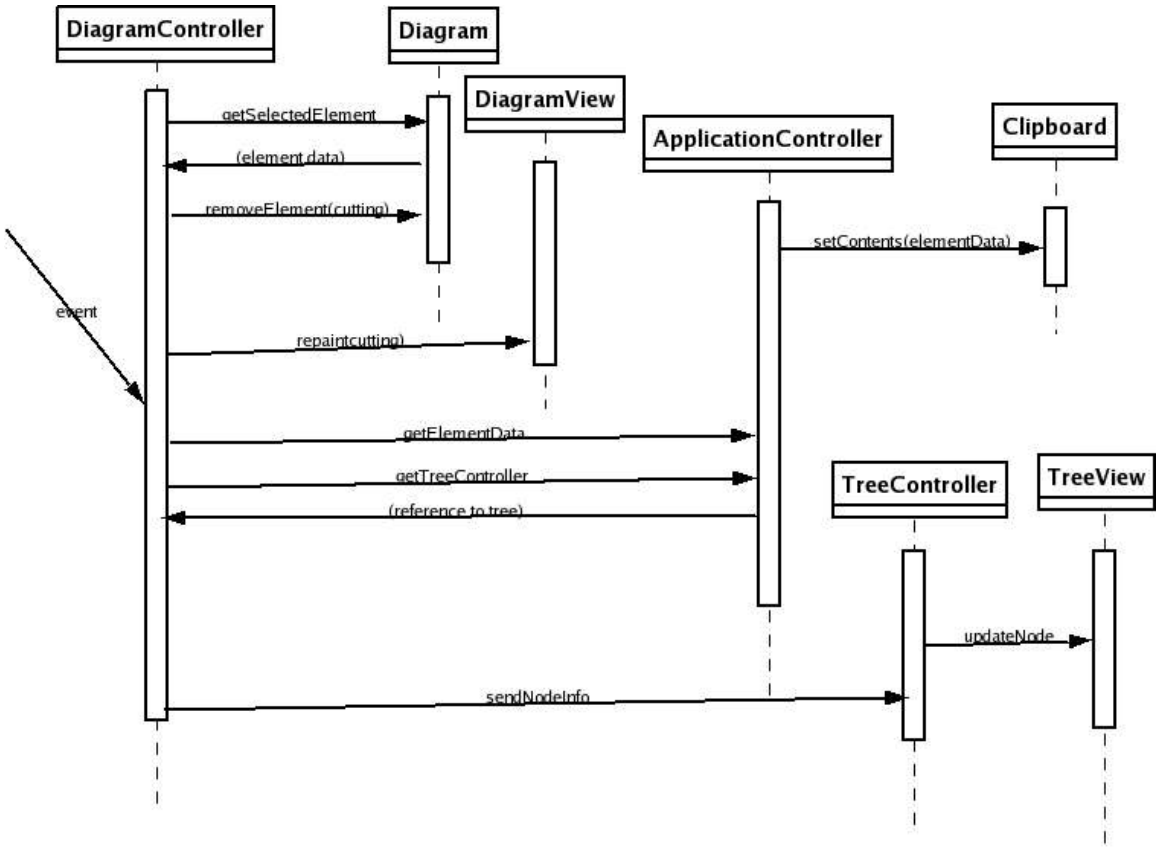
Object collaboration: DiagramController gets an event from the graphical user interface. DiagramController has to process the event and decide what action the user is currently doing. Different actions that can be committed by the user are for example moving and resizing of elements. DiagramController updates Diagram class accordingly. The Diagram class notifies observers.

Copying, cutting and pasting elements

Use case: The user can select one or more elements, and copy or cut them to the clipboard. Elements in clipboard can be pasted to another location on the diagram. The program checks if the operation is valid from syntactic properties of the affected elements.

Example: The user copies a rectangle to the clipboard and pastes several copies of it to the diagram.

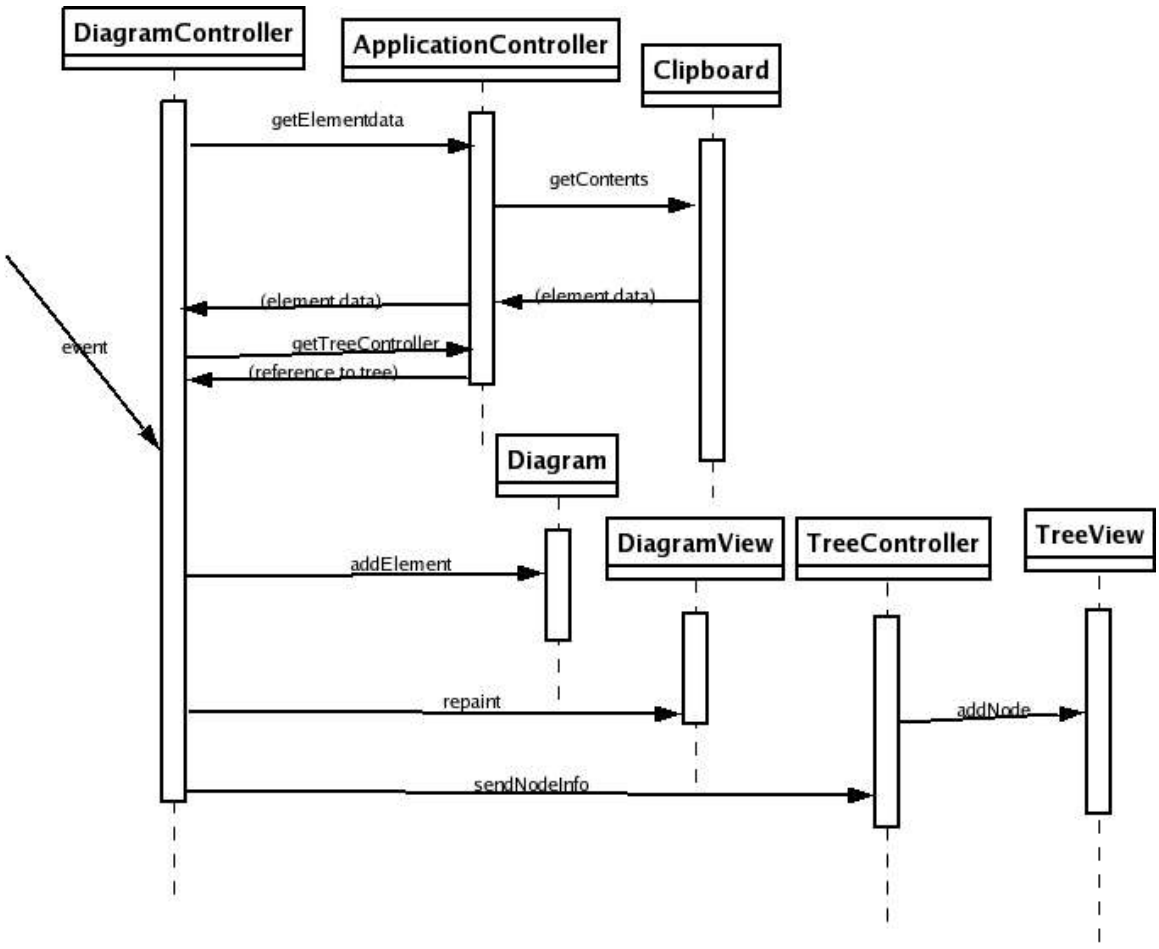
(i) **Copying/Cutting**



Picture 7: Object collaboration diagram

Object collaboration: DiagramController gets an event and a reference to the selected element is received from Diagram. If operation is cutting then the element is removed from the Diagram. The Clipboard contents is then set to the copied/cut element's data. The Diagram class notifies observers.

(ii)Paste

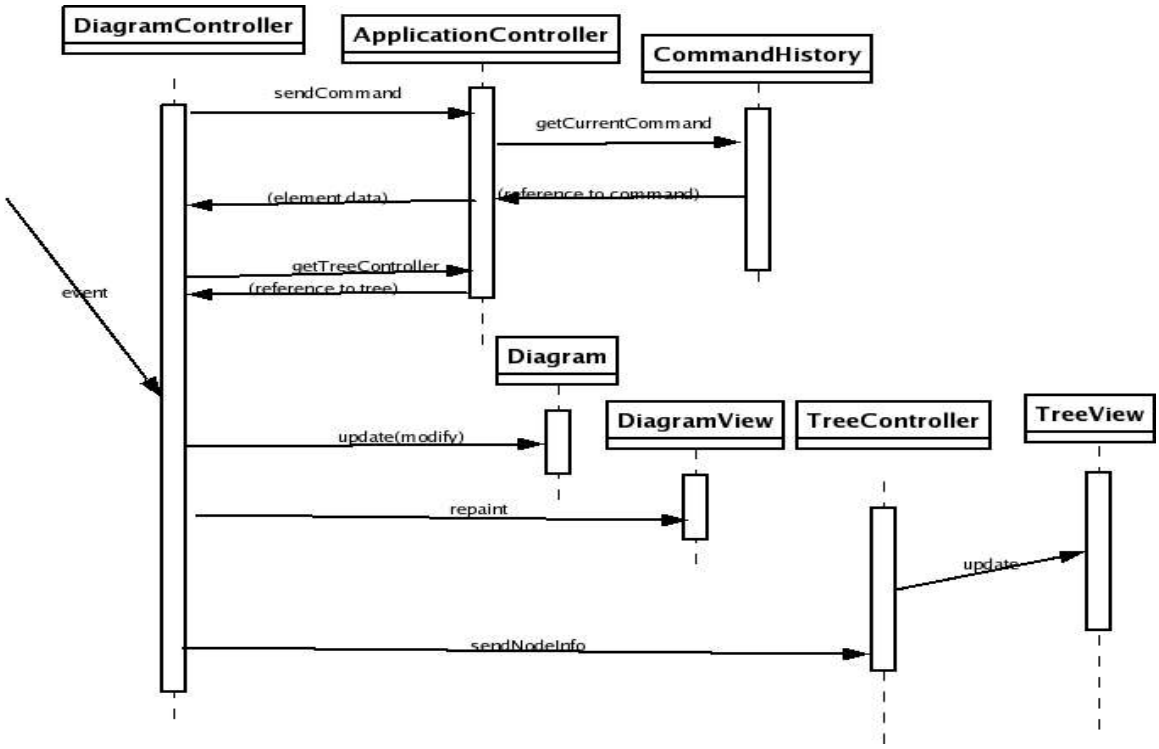


Picture 8: Object collaboration diagram

Object collaboration: DiagramController gets an event and requests the element to be pasted from the the Canvas class which gets the current contents from the Clipboard. The Diagram class notifies observers.

Undoing an operation

The user can undo last operations that were committed. The program keeps track of the operations the user has committed. Example: The user removes an element by accident. He undoes the last command and the program restores the removed element and it's connections.



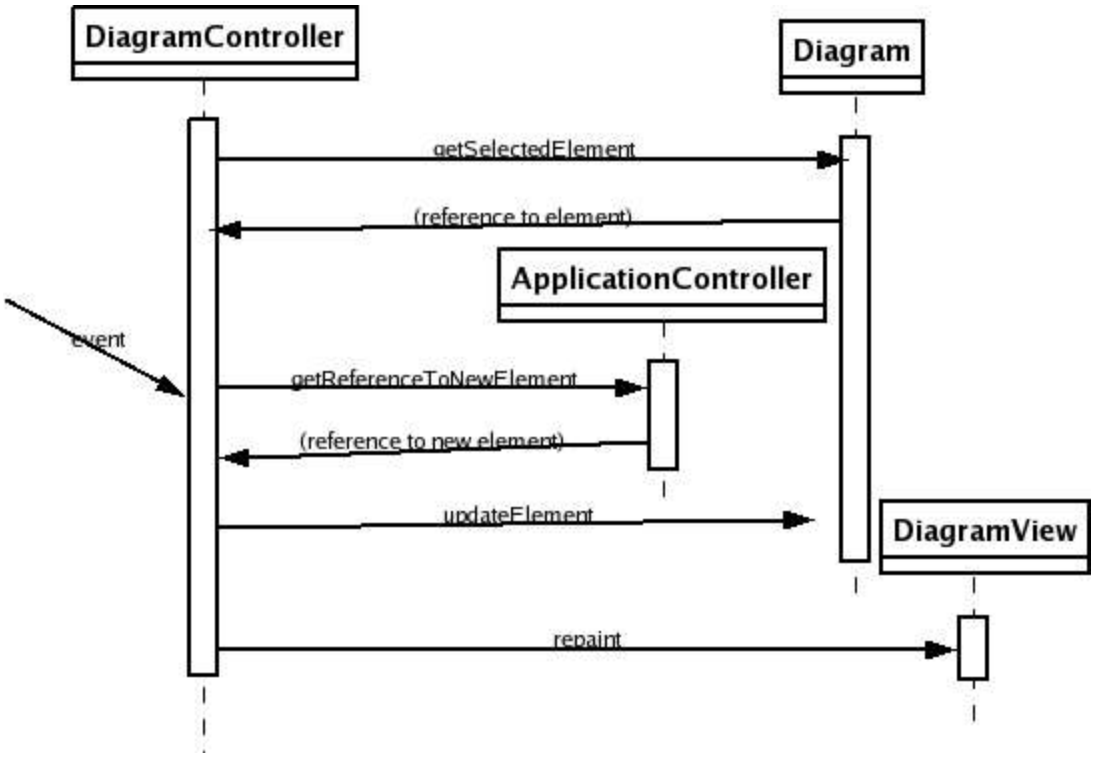
Picture 9: Object collaboration diagram

Object Collaboration: DiagramController gets an event from the system and sends the command to the Canvas class which gets the current command from CommandHistory. The Diagram class notifies observers.

Creating elements included inside other elements

The user picks an element to be inserted. Then he draws the new element inside an existing element. The program checks if the operation is valid from syntactic properties of the affected elements.

Example: The user draws a diagram of two circles inside a rectangle element.



Picture 10: Object collaboration diagram

Object Collaboration: DiagramController gets an event and obtains the element from Diagram. The new element to be inserted is obtained from the class Canvas(note the sources are variable: from Toolbar, another diagram etc). The element is updated and the Diagram class notifies observers.

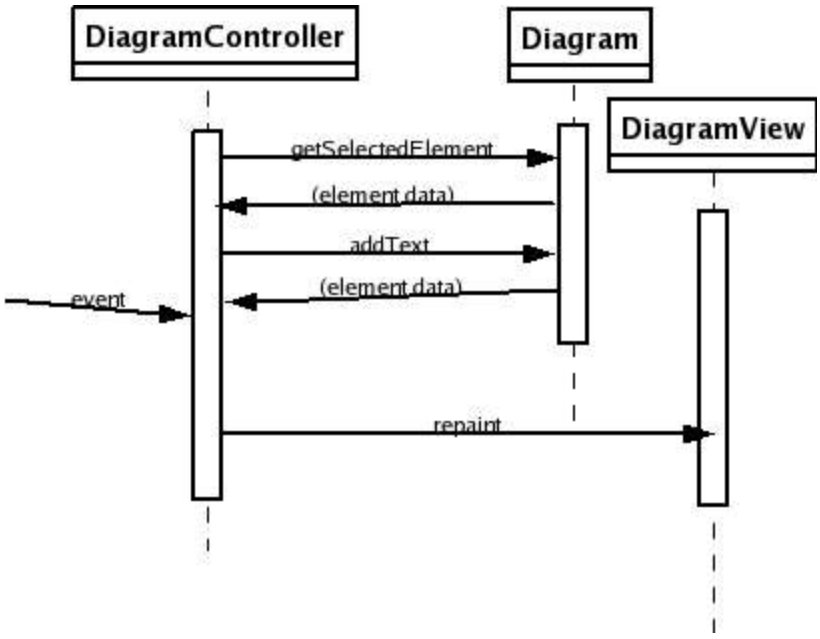
Including a text into the diagram

The user can include text independent of any element into the diagram. The user can also add links to external text files into elements. Text can be later modified or deleted by the user and the text will be saved with the diagram

Example: The user adds generic written information about the diagram within the diagram.

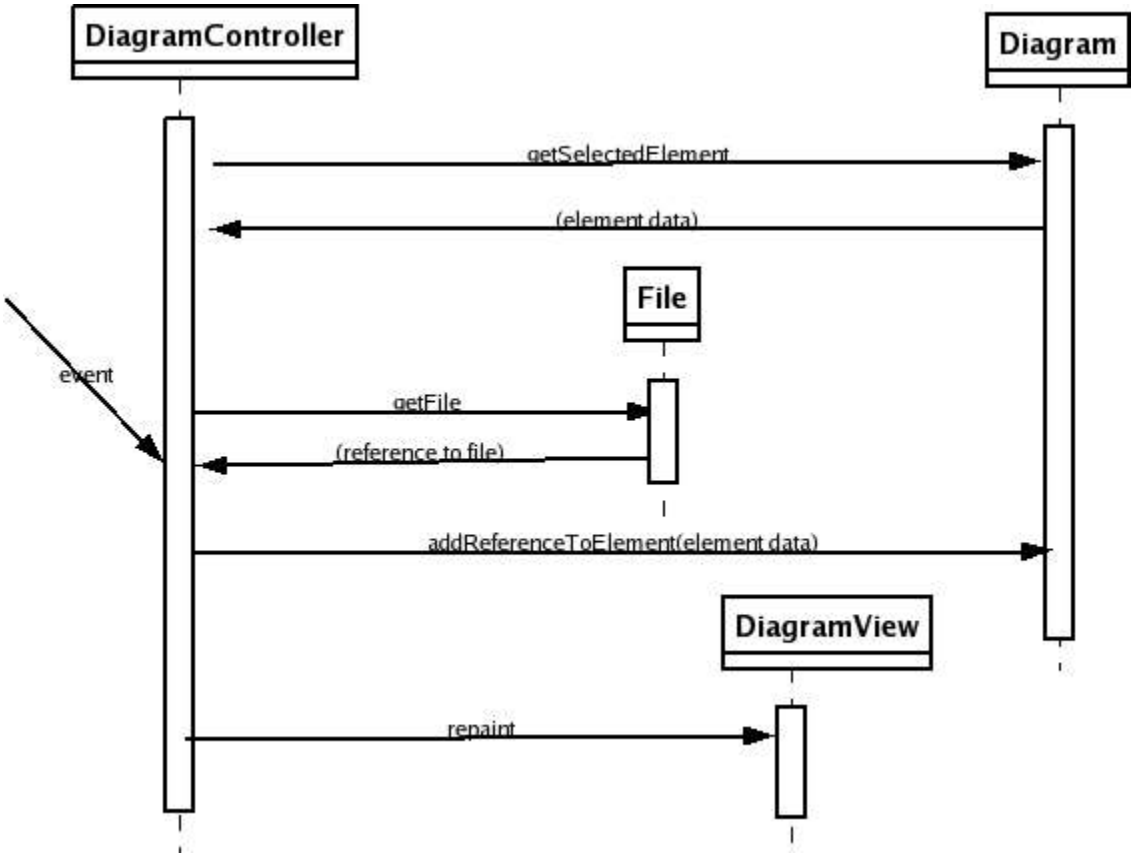
(ii)Including text into elements

Object Collaboration: DiagramController gets an event and obtains the selected element. The text is added to the element and the Diagram class notifies observers..



Picture 11: Object collaboration diagram

(ii)including a reference to element



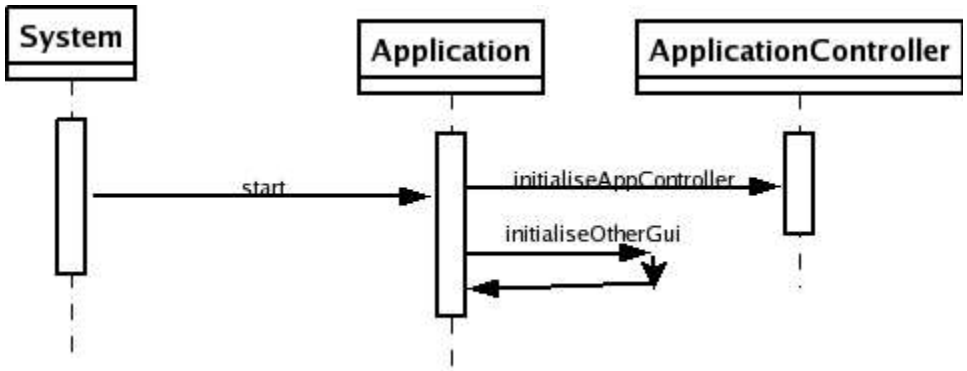
Picture 12: Object collaboration diagram

Object Collaboration: DiagramController gets an event and obtains the selected element as shown in picture 12. The referenced file is obtained from File and the Diagram class notifies observers.

Starting the application

Use case: The application is started by issuing a command or selected in a menu from the system.

Example: The user starts the application from Linux shell by issuing the command “java canvas”.



Picture 13: Object collaboration diagram

Object collaboration: The application Canvas receives an event from the system to start. Then it initializes the Canvas and all GUI classes as shown in picture 13.

6.3 Element editing

Defining a new element by composition

Use case: The user defines new elements by constructing them from other elements. Syntactic rules can be given to determine the behavior of the new element. The program provides basic shapes for the user to work with. New syntactic rules can be implemented with Java.

Example: The user creates a new by combining rectangle and circle shapes.

Object collaboration: DiagramController calls Diagram object with method groupElement. GroupElement method creates a new composite element from associated elements. The Diagram class notifies observers.

Error handling:

Editing existing elements

Use case: The user can edit the existing elements of the program by modifying the element's shape or by changing element's properties. Syntactic rules can be modified with Java. The program updates the diagram according to the rules associated with affected elements.

Example: The user modifies an old element by adding a new circle shape to it.

Object collaboration: DiagramController calls Diagram object with method updateElement. UpdateElement method modifies the composite element accordingly. The Diagram class notifies observers.

7. Maintenance

7.1 Extendibility of the application

Defining a new basic element

The user can extend the application by defining new elements using Java. The application provides an API for creating new elements. For example one could create a bezier line element for the application. New elements should follow the interface definition of class Element. Object collaboration works the same way as it works with build in elements.

Defining a new syntactic rule

One can create new syntactic rules for the application using Java. The application provides an API for creating new rules. For example one could create a must include rule for the application. New syntactic rules should follow the interface definition of rules processing subsystem. Object collaboration works the same way as it works with build in rules.