Command Line Vector Graphics Software Project Report

ZHANG Wengyu 21098431d
XU Le 21096101d
CHEN Derun 21098424d
YE Haowen 21098829d

COMP2021 Object-Oriented Programming (2021) - Project Group 1

Contents

1 Introduction

This document describes the design and implementation of the Clevis tool by group 1. The project is part of the course COMP2021 Object-Oriented Programming at PolyU.

2 Design

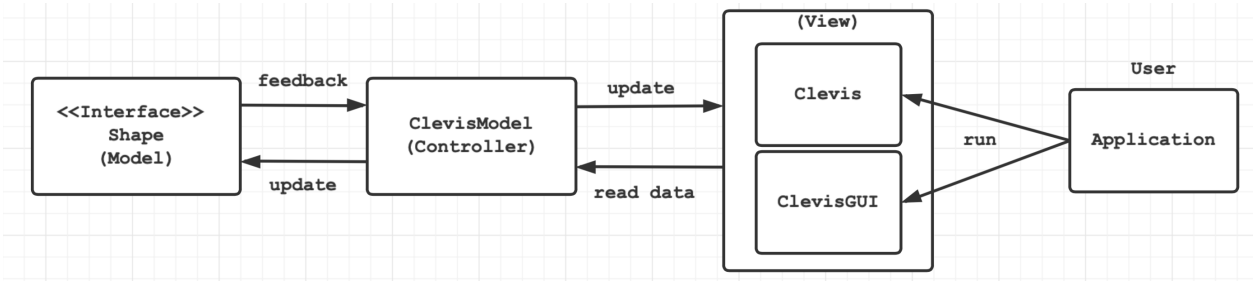

Figure 1: Overall Design Diagram

In general, the project is mainly based on the MVC design pattern (See Figure 1). The **Shape interface** is the basic model, **ClevisModel class** is the controller, the **Clevis class** & **ClevisGUI class** is the View, eventually user use the application to run the **Clevis class** or **ClevisGUI class**.

ClevisModel reads data from the **Clevis** and passes them to **Shape interface** for updating, while **ClevisModel** also get the changed data from Model and update on the View for the user.

2.1 Specifications

The assumed specifications of this project is the following:

- Every Shape that user creates will have a globally unique name, so we can link each Shape with its name and store it into a HashMap for data retrieval.
- We set the error (EPS) of our floating point calculation to **1E-6** which is sufficiently smaller than the radius of Circle we create in **pick and move** is **0.05**.
- Any shape that is created later will have a larger z-order than all previous shapes, and a group shape should have a larger z-order than all its children at the time of its creation.
- The name of the project file is "ClevisProjectGroup1", run `Application` class to launch Clevis(CLI) or ClevisGUI.
- The coordinates (0,0) is the top-left corner of the coordinate system in the project, with x-axis increasing to the right and y-axis increasing to the bottom.

2.2 Model Class

According to the project description, it requires objects of **rectangle**, **line**, **circle**, **square** to be drawn, as well as **group** and **bounding box** to be generated. Therefore we create a `Shape` interface, implemented by class `Line`, `Rectangle`, `Circle`, and `Group`, while class `Square` and `BoundingBox` extends class `Rectangle` as main model of the project (See Figure 2).

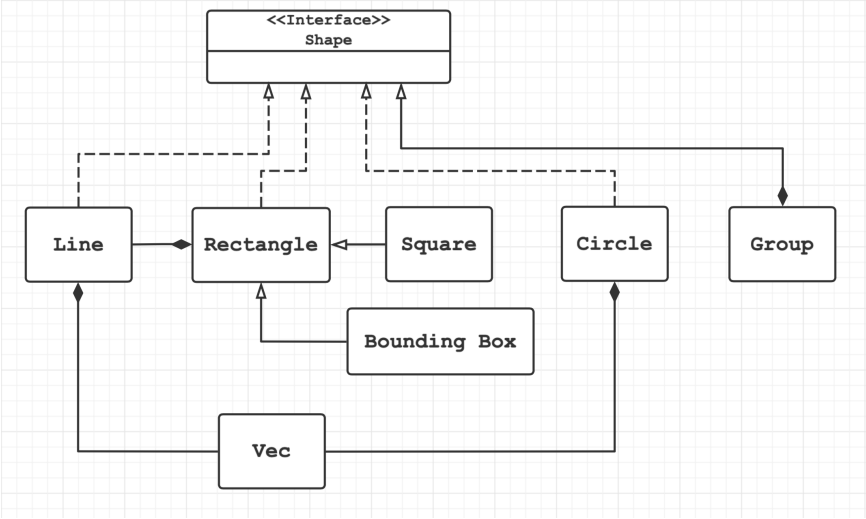

Figure 2: Model UML Class Diagram

Since the project is highly related to computational geometry, we use a vector class `Vec` (See Figure 2) as the fundamental element (i.e. a vector is a point, and a point is a vector) which will further form `Line` (two `Vectors`), `Rectangle` (four `Lines`), `Square` (a special case of `Rectangle`), `Circle` (one `Vector` with `double` **radius**), and `Group`(a list of `Shapes`). Besides, it is convenient for vectors to compute inner product and outer product. With this encapsulation, determining whether two Shape are intersected can be divided into 3 sub-problems:

- Line A and Line B (also useful for Rectangle and Rectangle, Rectangle and Square, Square and Square, Rectangle and Line, Square and Line)
- Line A and Circle B (also useful for Rectangle and Circle, Square and Circle)
- Circle A and Circle B

(Detail implementation see in 3.12 [REQ12])

2.2.1 Vec Class

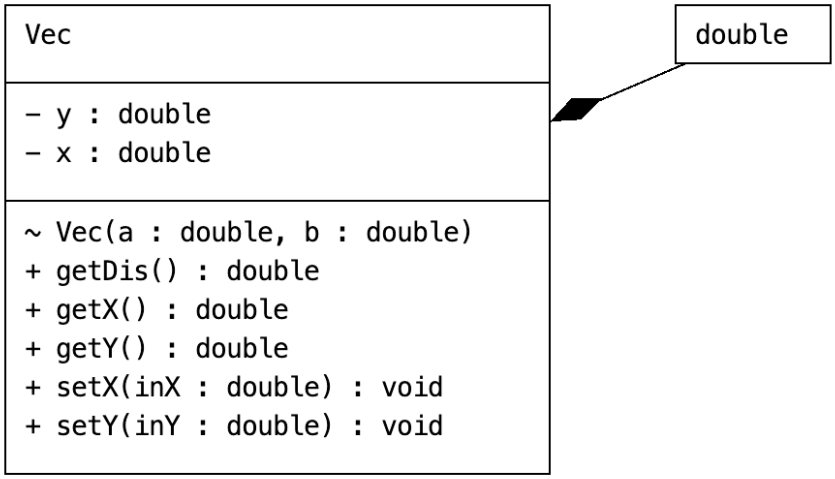

Figure 3: Vec Class

`Vec` class implements vector. It takes two `double` field `x` and `y` as two coordinates for the vector, initialized by the constructor (See Figure 3).

- `getDis()` method returns the length of the vector through calculating the distance between its coordinators and (0,0).
- `getX()` and `getY()` methods return the field `x` and `y`.
- `setX(inX: double)` and `setY(inY: double)` methods assign values to two fields

2.2.2 Shape Interface

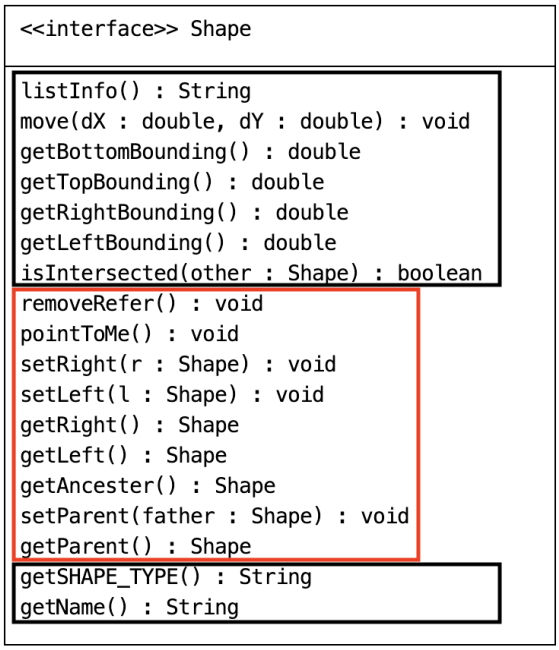

Figure 4: Shape Interface

`Shape` interface is an abstraction of different types of shapes including `Line`, `Rectangle`, `Circle`, `Square` as well as `Group` and `BoundingBox` in ClevisModel (See Figure 4).

For abstract methods in black box, named **functional method**:

- `getName()` method returns the name of the Shape.
- `getSHAPE_TYPE()` method returns the type name of the Shape.
- `listInfo()` method returns the information of the Shape, for [**REQ13**].
- `move()` method moves the Shape, for [**REQ10**].
- `getBottomBounding()`, `getTopBounding()`, `getRightBounding()`, `getLeftBounding()` methods get 4 corresponding bounds of the Shape, for [**REQ9**].
- `isIntersected()` method checks whether this shape is intersected with another, for [**REQ12**].

For abstract methods in red box, named **relationship method**:

- methods for LinkedList Deque, see details in 2.5.3 **Deque by LinkedList**.

2.2.3 Line Class

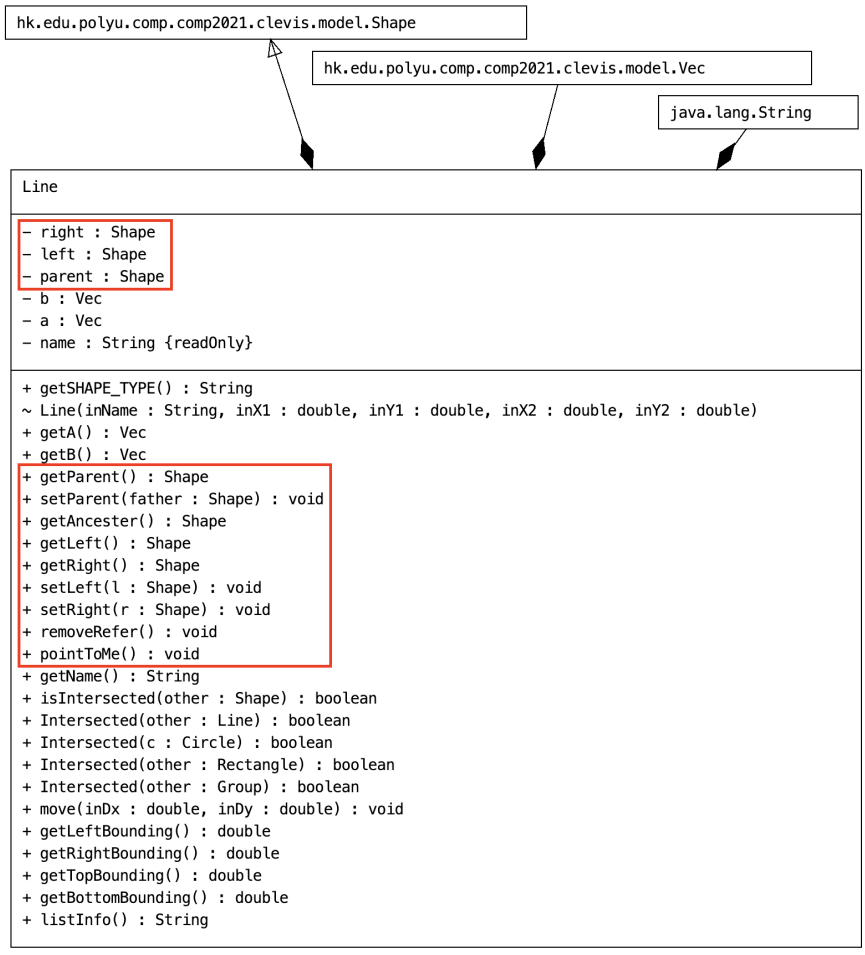

Figure 5: Line Class

`Line` is the class of **line** object in ClevisModel, implementing the `Shape` interface (See Figure 5).

It takes two `Vec` fields `a` and `b` for a line is formed by two vectors, as well as a `final String` field `name` for the name of the shape.

For methods not in red box:

- constructor `Line()` takes 4 parameters:

  `inName` for the name of the Line object;

  `inX1` and `inY1` for the x and y coordinate of Vec `a` respectively;

  `inX2` and `inY2` for the x and y coordinate of Vec `b` respectively;

- `isIntersected(other:Shape)` method checks which `other` Shape is checked with `this` Shape, then invokes the corresponding `Intersected(other:OTHER_TYPE)` below and return the boolean value.

- `Intersected(other:Line)` method checks whether the **Line** is intersected with **another Line**, see details in 3.12 **REQ12**.

- `Intersected(other:Circle)` method checks whether the **Line** is intersected with a **Circle**, see details in 3.12 **REQ12**.

- `Intersected(other:Rectangle)` method checks whether the Line is intersected with a **Rectangle**. It invokes the `isIntersected(other:Shape)` method to check whether this Line is intersected with each four Lines of the `other` Rectangle.

- `Intersected(other:Group)` method checks whether the Line is intersected with a **Group**. It invokes the `isIntersected(other:Shape)` method to check whether this Line is intersected with Shapes in the `shapeList` of the `other` Group by invoke `getShapeList()` method in the Group class.

- `move(inDx:double,inDy:double)` method moves the Line horizontally by `inDx` and vertically by `inDy`. It add the moved distance `inDx` and `inDy` to the original coordinates of Vec `a` and `b`.

- `getBottomBounding()` method returns the maximum value between **y** coordinate of Vec `a` and `b` as the bottom bound of the Line.

- `getTopBounding()` method returns the minimum value between **y** coordinate of Vec `a` and `b` as the top bound of the Line.

- `getRightBounding()` method returns the maximum value between **x** coordinate of Vec `a` and `b` as the right bound of the Line.

- `getLeftBounding()` method returns the minimum value between **x** coordinate of Vec `a` and `b` as the left bound of the Line.

2.2.4 Rectangle Class

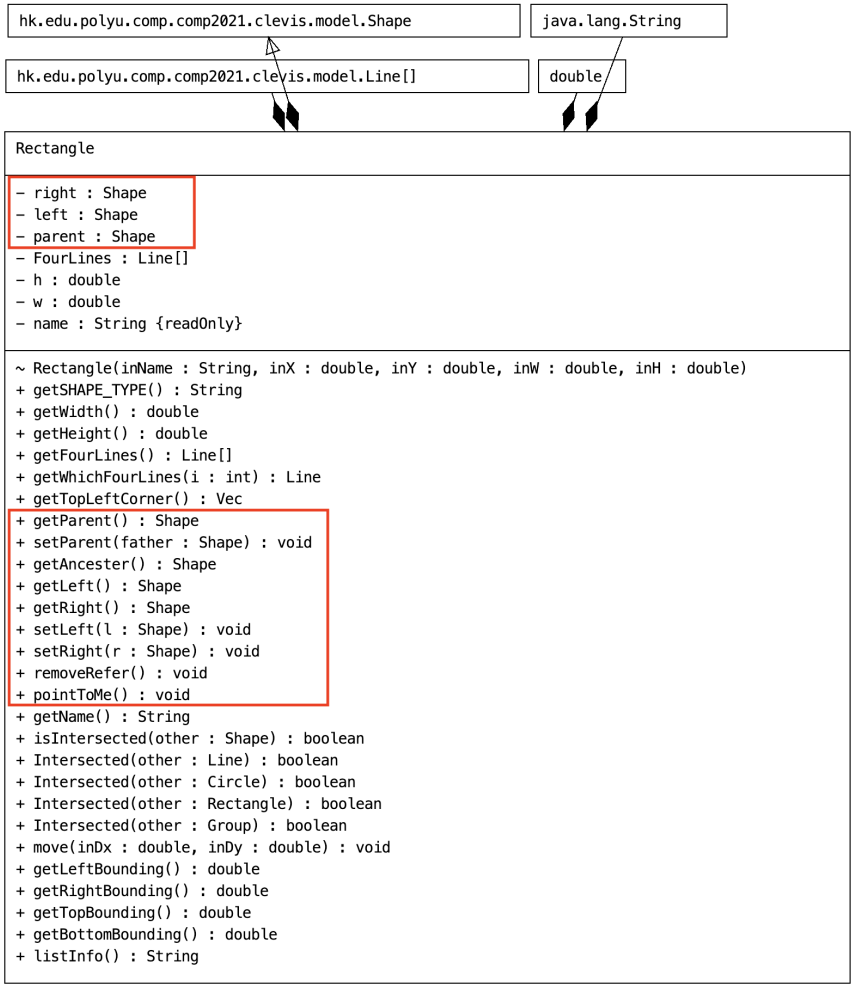

Figure 6: Rectangle Class

`Rectangle` is the class of **rectangle** object in ClevisModel, implementing the `Shape` interface (See Figure 6). The `Rectangle` is formed by four `Lines` object as four segments of the rectangle.

It takes a `Line[]` field `FourLines` to store four `Lines` object, and `double` field `h` and `w` for the **height** and **width** of the rectangle respectively, as well as the `String` field `name` for the name of the Rectangle.

For methods not in red box:

- constructor

  `Rectangle (inName:String, inX: double, inY:double, inW:double, inH:double)` initializes the `name`, field `h` and `w`. According to the coordinates of top-left corner `inX`, `inY`, width `h` and height `w` to generate four `Line` instances and store in the field `FourLines` with a fixed order of index (0: line-Left, 1: line-Bottom, 2: line-Right, 3: line-Top).

- `getFourLines()` and `getWhichFourLines(i:int)` methods

  returns the field `FourLines`, and a specific Line in it, respectively.

- `getTopLeftCorner()` method returns the Vec `a` of the left-Line (`getWhichFourLines(0)`)
- `isIntersected(other:Shape)` method checks which `other` Shape is checked with `this` Shape, then invokes the corresponding `Intersected(other:OTHER_TYPE)` below and return the boolean value.
- `Intersected(other:Line)` method checks whether the **Rectangle** is intersected with **another Line**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Line in the field `FourLines` is intersected with the **another Line**, and return the boolean result.
- `Intersected(other:Circle)` method checks whether the **Rectangle** is intersected with **another Circle**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Line in the field `FourLines` is intersected with the **another Circle**, and return the boolean result.
- `Intersected(other:Rectangle)` method checks whether the **Rectangle** is intersected with **another Rectangle**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Line in the field `FourLines` is intersected with the **another Rectangle**, and return the boolean result.
- `Intersected(other:Group)` method checks whether the **Rectangle** is intersected with **another Group**. It invokes `isIntersected(other:Shape)` method from the `Shape` interface to check whether the **another Group** is intersected with the **Rectangle**, and return the boolean result.
- `move(inDx:double,inDy:double)` method moves the Rectangle horizontally by `inDx` and vertically by `inDy`. It moves each Line in the field `FourLines` by invoking the `move(inDx:double,inDy:double)` in the `Shape` interface.
- `getLeftBounding()` method returns the **x** coordinate of field Vec `a` of the Line-left (`getWhichFourLines(0)`).
- `getRightBounding()` method returns the **x** coordinate of field Vec `a` of the Line-right (`getWhichFourLines(2)`).
- `getTopBounding()` method returns the **y** coordinate of field Vec `a` of the Line-top (`getWhichFourLines(3)`).
- `getBottomBounding()` method returns the **y** coordinate of field Vec `a` of the Line-bottom (`getWhichFourLines(1)`).

2.2.5 Circle Class

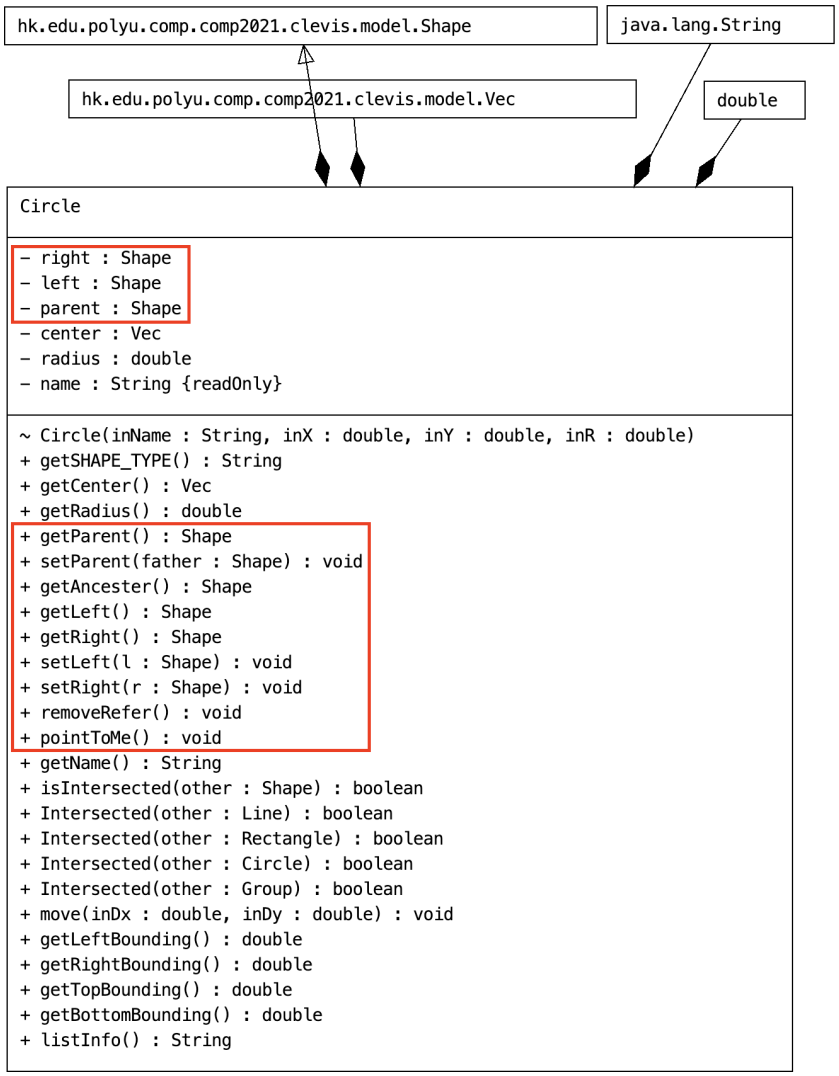

Figure 7: Circle Class

`Circle` is the class of **circle** object in ClevisModel, implementing the `Shape` interface (See Figure 7).

It takes a `Vec` field `center` as the **center vector** of the circle, and a `double` field `radius` as the **radius** of the circle, initialized by the constructor.

For methods not in red box:

- `isIntersected(other:Shape)` method checks which `other` Shape is checked with `this` Shape, then invokes the corresponding `Intersected(other:OTHER_TYPE)` below and return the boolean value.

- `Intersected(other:Line)` method checks whether the **Circle** is intersected with **another Line**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether **another Line** is intersected with the **this Circle**, and return the boolean result.

- `Intersected(other:Rectangle)` method checks whether the **Circle** is intersected with **another Rectangle**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether **another Rectangle** is intersected with the **this Circle**, and return the boolean result.
- `Intersected(other:Circle)` method checks whether the **Circle** is intersected with **another Circle**, see details in 3.12 **REQ12**.
- `Intersected(other:Group)` method checks whether the **Circle** is intersected with **another Group**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether **another Group** is intersected with the **this Circle**, and return the boolean result.
- `move(inDx:double,inDy:double)` method moves the Circle horizontally by `inDx` and vertically by `inDy`, adding the `inDx` and `inDy` to the **x** and **y** coordinate of `center Vec` respectively.
- `getLeftBounding()` method returns the difference between the **x** coordinate of `center Vec` and `radius`.
- `getRightBounding()` method returns the addition of the **x** coordinate of `center Vec` and `radius`.
- `getTopBounding()` method returns the difference between the **y** coordinate of `center Vec` and `radius`.
- `getBottomBounding()` method returns the addition of the **y** coordinate of `center Vec` and `radius`.

2.2.6 Square Class

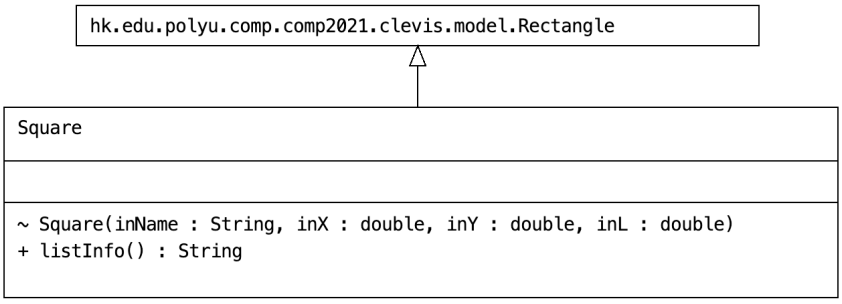

Figure 8: Square Class

`Square` is the class of **square** object in ClevisModel, extends from the `Rectangle` class (See Figure 8) for the `Square` is a special case of the rectangle with the equivalent length of width and height. In the constructor `Square (inName:String, inX: double, inY:double,`

`inL:double)`, it invokes its super class's constructor (i.e. `super (inName, inX, inY, inL, inL)`) to ensure the the equivalent length of width and height.

2.2.7 Group Class

Figure 9: Group Class

According to the project description, we need to group existing shapes and do operations on it as same as on individual Shapes.

Therefore we create a class `Group` for **Group** object in ClevisModel, implementing the `Shape` interface (See Figure 9). Essentially, it takes a `Shape` Array field `shapeList` to store and refer to the shapes grouped by a Group object.

For methods not in red box:

- the constructor take the `Shape[]` parameter `inShapeList`, and assigns it to the `Shape` Array field `shapeList`.

- `getShapeList()` method returns the field `shapeList`.
- `isIntersected(other:Shape)` method checks which `other` Shape is checked with `this` Shape, then invokes the corresponding `Intersected(other:OTHER_TYPE)` below and return the boolean value.
- `Intersected(other:Line)` method checks whether the **Group** is intersected with **another Line**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Shape in the `shapeList` of the Group is intersected with the **other Line**, and return the boolean result.
- `Intersected(other:Rectangle)` method checks whether the **Group** is intersected with **another Line**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Shape in the `shapeList` of the Group is intersected with the **other Rectangle**, and return the boolean result.
- `Intersected(other:Circle)` method checks whether the **Group** is intersected with **another Line**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Shape in the `shapeList` of the Group is intersected with the **other Circle**, and return the boolean result.
- `Intersected(other:Group)` method checks whether the **Group** is intersected with **another Line**. It invokes `isIntersected(other:Shape)` method from the Shape interface to check whether each Shape in the `shapeList` of the Group is intersected with the **other Group**, and return the boolean result.
- `move(inDx:double,inDy:double)` method moves the Group horizontally by `inDx` and vertically by `inDy`. It moves each Shape in the field `shapeList` as well as itself by invoking the

  `move(inDx:double,inDy:double)` in the `Shape` interface.
- `getLeftBounding()` method returns the **minimum** value from the left bounds of each Shape in the field `shapeList` invoking the `getLeftBounding()` in the `Shape` interface.
- `getRightBounding()` method returns the **maximum** value from the right bounds of each Shape in the field `shapeList` invoking the `getRightBounding()` in the `Shape` interface.
- `getTopBounding()` method returns the **minimum** value from the top bounds of each Shape in the field `shapeList` invoking the `getTopBounding()` in the `Shape` interface.
- `getBottomBounding()` method returns the **maximum** value from the bottom bounds of each Shape in the field `shapeList` invoking the `getBottomBounding()` in the `Shape` interface.

- `listInfo()` method uses `levelCount` field and invokes the `spaceGen`, `inFour` and `deFour` methods to list the information of the group including the shapes inside, using the indentation to indicate the containing relation.
- `resetLevel()` method resets the `levelCount` to 0 for the next invoking of `listInfo()` method.

2.2.8 BoundingBox Class

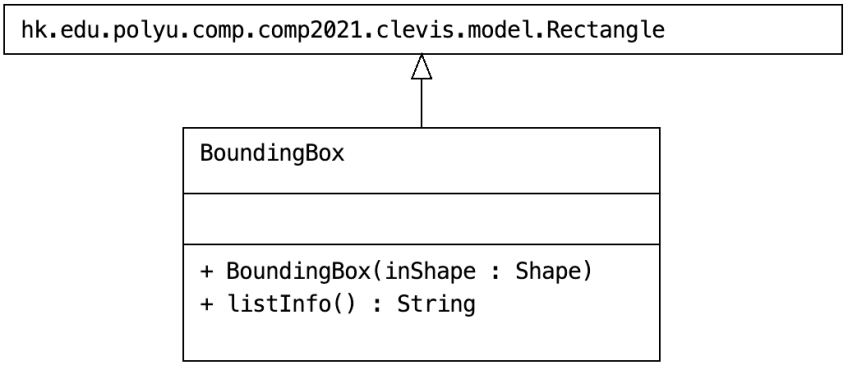

Figure 10: BoundingBox Class

According to the project description, we need to output the bounding box of a Shape, with top-left corner coordinates, width and height, which refers to a **Rectangle**. Therefore we create a `BoundingBox` class for **bounding box** object in ClevisModel, extends from the `Rectangle` class (See Figure 10). In the constructor `BoundingBox (inShape: Shape)`, it invokes its super class's constructor to create a `Rectangle` object with four bounds of the Shape as parameters (i.e.

```
super("Bounded " + inShape.getName(),

    //top-left corner coordinates
    inShape.getLeftBounding(), inShape.getTopBounding(),

    // width
    inShape.getRightBounding()-inShape.getLeftBounding(),

    // height
    Math.abs(inShape.getTopBounding()-inShape.getBottomBounding()));
```

).

2.3 ClevisModel Class (Controller Class)

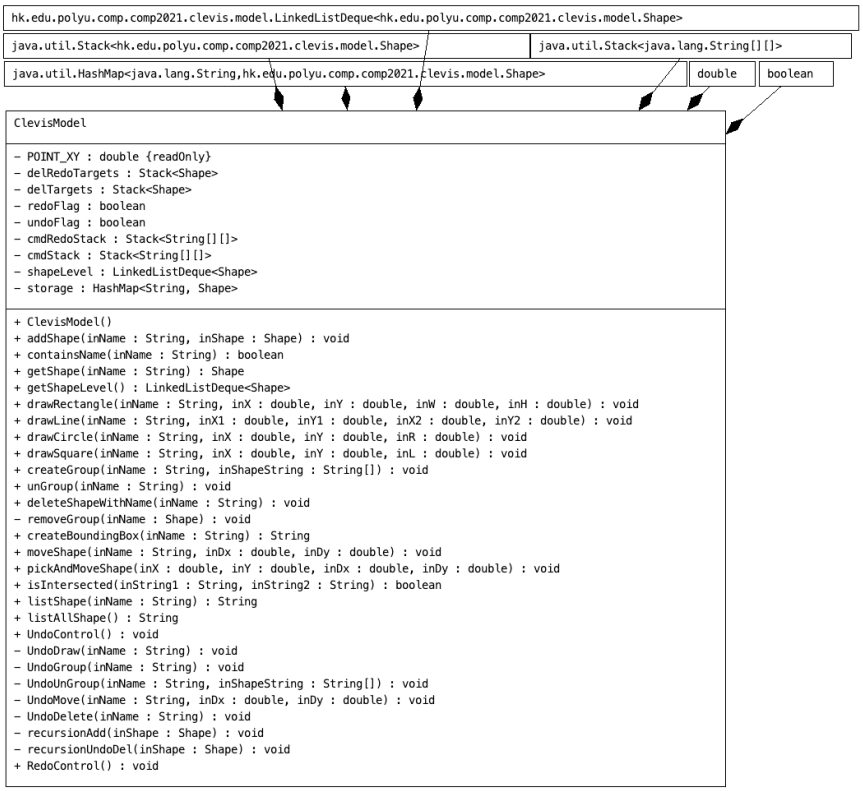

Figure 11: ClevisModel Class

`ClevisModel` class, as the controller, is the core class of the project, it processes the data from View class(i.e. CLI and GUI), generates corresponding objects, implements the required functions, and make internal storage, aggregating the previous developed `Shape` interface, and self-designed `LinkedListDeque` (see details in 2.5.3 **Deque by LinkedList**).

`ClevisModel` class can be divided into 3 parts:

1. Internal Storage Section
2. Basic Function Section
3. Undo & Redo Function Section

2.3.1 Internal Storage Section

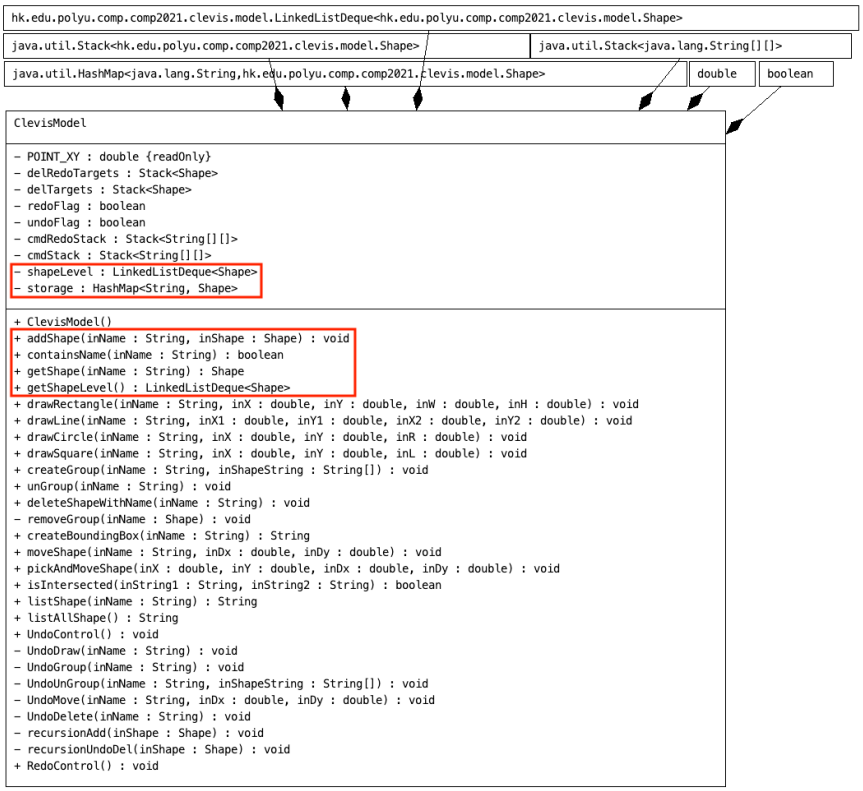

Figure 12: Internal Storage Section in ClevisModel Class

In order to store all kinds of data based on the user input (e.g. a Rectangle, BoundingBox,...) as well as their relations and orders. The project implements internal storage based on the `HashMap` and self-design `LinkedListDeque`. We use the `HashMap<String, Shape>` named `storage` to store the pair of shape name and corresponding Shape object, which can ensure the uniqueness of the shape name and improve the searching efficiency (see detail in 2.5.1 **HashMap**). In terms of the self-design `LinkedListDeque<Shape>` storage named `shapeLevel`, it stores the referencing relations between Shapes object (i.e. Grouping relation, Z-order relation).

Methods in red box for internal storage:

- `addShape(inName:String,inShape:Shape)` method adds the pair of name and Shape into the `storage`, and add the Shape into the `shapeLevel`.
- `containsName(inName:String)` method checks whether the name is already stored in the HashMap.
- `getShape(inName:String)` method returns the Shape object in the `storage` based on its name.
- `getShapeLevel()` method returns the `LinkedListDeque<Shape> shapeLevel`.

2.3.2 Basic Function Section

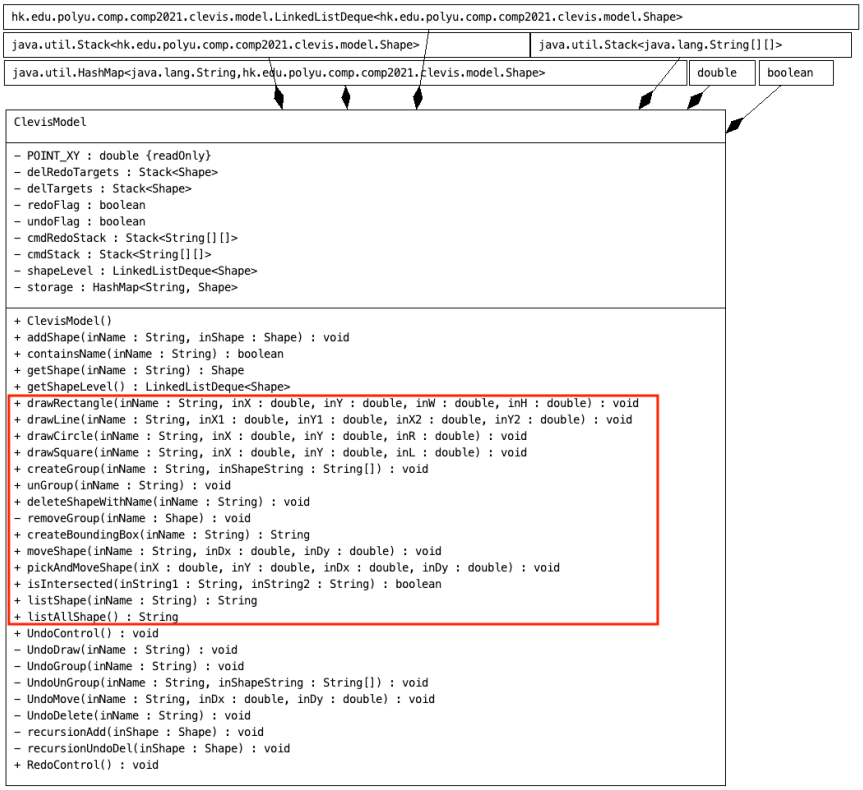

Figure 13: Basic Function Section in ClevisModel Class

In Basic Function Section, with 13 public methods and 1 private method, the project requirements **[REQ2] to [REQ14]** are implemented. Overall these methods take the data from View as their parameter and invoke methods of Shape objects in the internal storage.

- `drawRectangle()`, `drawLine()`, `drawCircle()`, and `drawSquare()` methods create `Shape` object based on the parameters and store them in `storage` and `shapeLevel`, invoking the `addShape(inName:String,inShape:Shape)` method.
- `createGroup(inName:String,inShapeString:String[])` method firstly creates a `Group` object named `inName`, with group shapes in Shape Array which is mapped with names in `inShapeString` based on the `storage` HashMap.

  Then it invokes `addShape(inName:String,inShape:Shape)` method to put the generated Group object into the internal storage.

  Finally, for attaching the **parent-child relation** to the grouped shapes, this method invokes `setParents(father:Shape)` (see details in 2.5.3 **Deque by LinkedList**) for each shapes in the `inShapeString`.
- `unGroup(inName:String)` method ungroups the Shape(Group) named `inName`.

First, it invokes `ungroup()` method (see details in 2.5.3 **Deque by LinkedList**) to remove the **parent-child relation** and **child-parent relation** of the Group and shapes inside.

Then it remove the inName-Shape pair out of the `storage` HashMap by invoking the `remove(inName:String)` method.

- `deleteShapeWithName(inName:String)` method deletes a Shape named `inName`.

  It invokes `removeRefer()` and `removeGroup()` methods (see details in 3.8 **REQ8**) to delete the shape from the internal storage.

- `createBoundingBox(inName:String):String` method creates a `BoundingBox` object

  taking the Shape named `inName` as the parameter. Then returns the information of the `BoundingBox` object by invoking the `listInfo():String` method of the `Shape` interface.

- `moveShape(inName:String,inDx:double,inDy:double)` method moves the Shape

  named `inName` horizontally by `inDx` and vertically by `inDy` by invoking

  the `move(inDx:double,inDy:double)` method of the `Shape` interface.

- `pickAndMoveShape (inX:double,inY:double,inDx:double,inDy:double)` method finds the shape containing the point(inX,inY) with the largest Z-order by checking the intersection between Shapes and given point (see details in 2.5.2 Quick Union & 3.11 REQ11), finally move the target Shape by invoking

  the `move(inDx:double,inDy:double)` method of the `Shape` interface.

- `isIntersected (inString1:String,inString2:String)` method checks and returns the boolean result of the intersection relation between two Shapes named `inString` and `inString2` by invoking the `isIntersected(other:Shape):boolean` method of the `Shape` interface.

- `listShape(inName:String):String` method return the String of the information about the Shape named `inName` by invoking the `listInfo():String` method of the `Shape` interface.

- `listAllShape():String` method traverses all Shapes from the `shapeLevel` storage, and appends each Shape's information to a `StringBuilder` object by invoking

  the `listInfo():String` method of the `Shape` interface on each Shape.

2.3.3 Undo & Redo Function Section

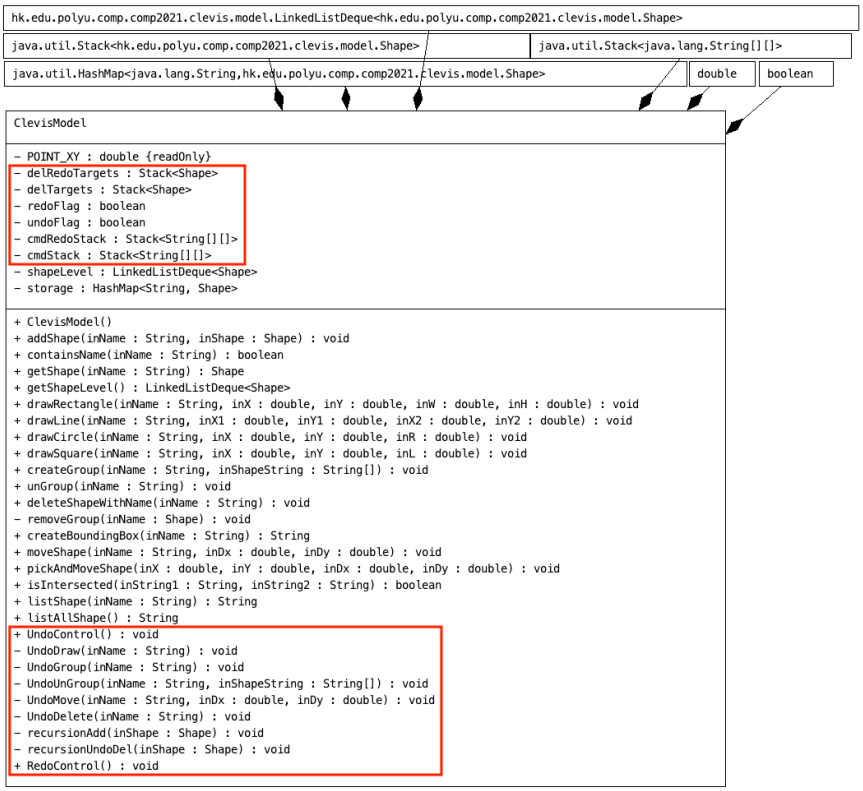

Figure 14: Undo & Redo Function Section in ClevisModel Class

In order to achieve undo and redo function:

- Use two 2D String Array `Stacks` (`cmdStack` and `cmdRedoStack`) as field for storing **commands to be undo** and **commands to be redo** respectively.
- and two boolean `undoFlag` and `redoFlag` as field to mark the states of undo and redo.
- two `Shape Stacks` `delTargets` and `delRedoTargets` store the deleted Shape for further undo and redo operation, which is the most complex situation when user call undo and redo.

Methods in red box:

- `UndoControl()` method takes controls of undo operation. It takes the last command stored in the `cmdStack`, then can call corresponding methods to implement undo:
  - Undo **draw** commands: call `Undodraw()`;
  - Undo **group** commands: call `UndoGroup()`;
  - Undo **ungroup** commands: call `UndoUnGroup()`;
  - Undo **delete** commands: call `UndoDelete()`;
  - Undo **move** and **pick-and-move** commands: call `UndoMove()`;
- `RedoControl()` method takes controls of redo operation.

  (See undo and redo implementation details in 2.5.2 Quick Union & 3.17 [BON2])

2.4 View Class

2.4.1 CLI view class

The Command-Line Interface(Clevis) provide the whole functions of project operations inside command line controller, contains `drawing four different shapes, group/ungroup, delete, making boundingbox, moving, intersection, list, undo/redo and Quit`.

When choose '0' in `application`, it will enter the CLI interface. It will give corresponding respond about whether successfully complete functions after inputting correct and significant commands in terminal.

The basic implementation of CLI is `Scanner` function. What the CLI does is to determine which function user prefer according to the `"System.in"` line by line. As correct format, function word is the first word of each line for judging the using function and input command step for step for user using each function. So we using `Scanner.next()` to detect each word with `split()` according to the space. When determine the first word in each line, CLI will judge with `if-else` condition until finding corresponding function word, then it will finish parameter and data of different function until searching nothing anymore for each line.

For each function in `if-else` condition, it will catch the corresponding exception and determine whether needs exception or not. If so, it will give the error with certain kind of exception; else, it will use the function in `ClevisModel` to finish the function and back to the command input interface while `quit()` function executing.

- For the requirement of recording user input string into **log.txt** file, we use `BufferedWriter()` to write `System.in` data into local directory txt with `Scanner.nextLine()` to store whole line string before judging the function word in line, then use `BufferedRead()` to read the content in log.txt with `readLine()` to store the string in `Scanner` as the input command for function calling.

- For the requirement of recording user input string into **log.html**, due to the html unlike txt with real-time storing, it uses `StringBuilder()` to store the input string line by line with `FileOutputStream()` and generate html file with `PrintStream` to print all information in `log.html` until `quit()` operate.

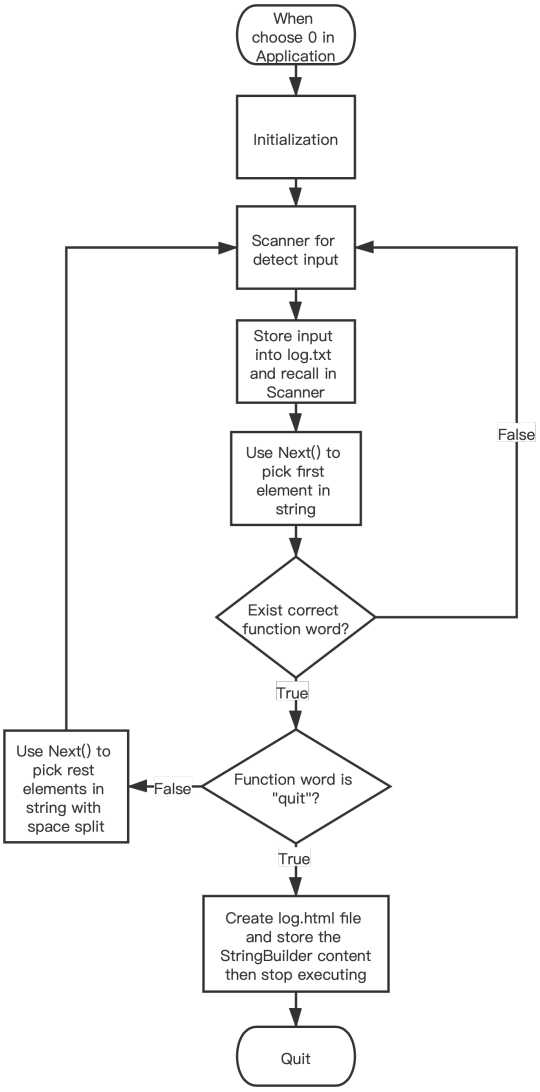

Figure 15: Flowchart of CLI structure

2.4.2 GUI view class

In GUI View class, our group mainly uses SWING package to make accomplishment, including overall layout, components and corresponding methods.

- The whole frame of GUI is based on a JFrame. Rather than using many JDialogs, we choose using different JPanels to show different function pages, which are all added in `this.getContentPane()`. And all the components, containing JButtons, JLabels, JTextArea, JTextfield are placed in each JPanel. Also, a special JPanel which is used as `Draw Area` will appear in the upper half of frame through the whole program.

- The JButton `Back` realizes jumping to different function interfaces.

  By using `setVisibile(false), setVisibile(true) and remove()` in

  the `Back's ActionListener lamda statement`, we realize the appearance of different interfaces.

- The accomplishments of all functions are based on the methods predefined in `ClevisModel.java and the built-in functions in Swing`.
  - To realize draw function, we have to `override` the paint method, and when we need to paint, we should invoke `repaint()`. So in overrode paint method, we define each draw methods to draw different graphics, such as `drawRect, drawLine and drawOval`.
  - After user inputing different parameters in `Textfields`, and pressing the corresponding function button, the parameters will be sent and the corresponding method will be called with the help of the lamda statements under each buttons. For example, to realize drawShape of drawing a rectangle. After typing eachparameters and press `drawShape` button, each parameter will be sent to `ClevisModel.drawRectangle()` initially until `repaint()` is called. After that, the repaint() will call the overrode method `paint()`. Firstly, we will identify the shape type of inputing shape, using `shape.getSHAPE_TYPE().equals()`, then the method `g.drawRect()` will be invoked to realize painting the Rectangle.
  - In order to make not `only one` graphic is painted in draw area, after adding a new graphic into specified shapeList, the method `repaint()` should be invoked once undoubtedly.

- For each `function button` action listener, no matter which exception happens, `catch` statement will catch it with `Exception e1`. And set the content of the label into "Illegal input, please try again!", which shows the error message to warn users.

- For a better human-computer interaction experience, we define a `refreshsize() method` to refresh size of frame when the `Draw Area` is not big enough to paint. By adding `shape.getBottomBounding()` and `shape.getRightBounding()` to the current the maximum value of the Y-axis and X-axis respectively, we can get a bigger size of `Draw Area` as well as the Frame size.

- For the other buttons actionlistener events, all are based on and invoked the predefined methods in `ClevisModel` class.

2.5 Data Structure Used

2.5.1 HashMap

In this project, retrieving data from the storage is frequently requested, for example, when user input `list LineB` command for the information of shape named "LineB", the program need to find the corresponding Shape object with that name, and do some operations. Therefor, to efficiently implement data retrieving, we use `HashMap` as the internal storage to store the pair of `(String,Shape)`, whose average cost of search and delete is amortized constant time $\Theta(1)$ [1].

2.5.2 Quick Union Without Path Compression

In this project, for efficiently implementing **pick and move**, **undo and redo**, we give each Shape with a parent instance field which connects to its parent. This will make a Shape figure out in which group it is. For example, if a specific Shape **A** is in a group **E**, then its parent should be the group shape **E**. Further if group **E** is also in another higher level group **H**, then **E's** parent should be group **H** and A**'s** ancestor should be group **H**, and we know that a Shape can not have multiple parents at the same time, so, all in all, the dependency relation is maintained by a tree-like structure, and this inspires us to try Quick Union, which is widely used in connectivity problem [2]. Since a Shape may not be in any group, so a single independent node is legal, and if a specific **Shape is in a group then its parent should be itself.** The cost of the operation of determining whether a specific Shape is already in a Group is constant time (i.e. check whether its parent is itself), and that of finding an ancestor of a specific Shape is $O(logn)$, where n is the total number of nodes.

We do not use the optimization of Path Compression because we want to maintain the dependency relation between `Shape` and `Group` [3].

Therefore, the structure is like Figure 16:

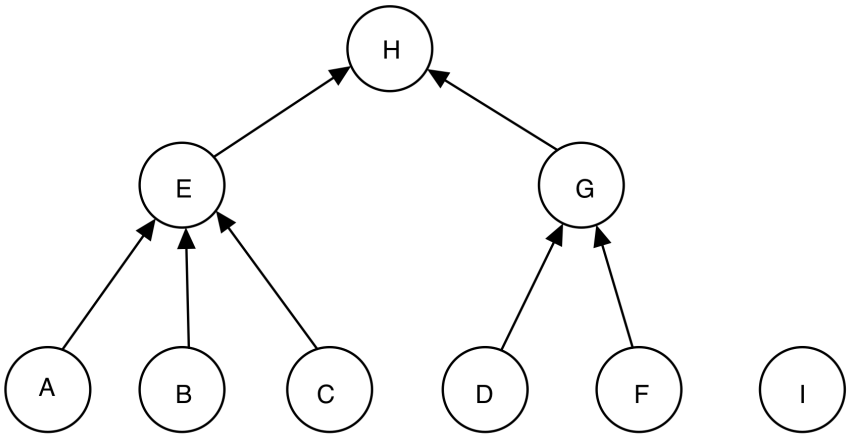

Figure 16: Quick Union

2.5.3 Deque by LinkedList

To maintain the z-order of each Shape, we utilise the linked list. We push back all the newly created Shape (a single independent Shape or a group) to the end of the list. However, the original linked list provided by `java.util.LinkedList` can not elegantly support traversing on the path of Quick Union and LinkedList (e.g., $\mathbf{B} \rightarrow \mathbf{D} \rightarrow \mathbf{G}$), and when implementing the **undo** for deleting a Shape, especially when it is a group, it is hard for us put back the Group together with its children into their previous position in the linked list, but if we have better topological structure (i.e. each node has field **left**, **right** and **parent**), **undo delete** will be easier when we first put back the group and then put back its children. Therefore, we create our own `LinkedListDeque`, where we imitate the industrial LinkedList with a circular node, to support those functions [4]. For saving memory, we combine the structure of Deque and Quick

Union into a new mixed structure - DeqUnion (See Figure 17).

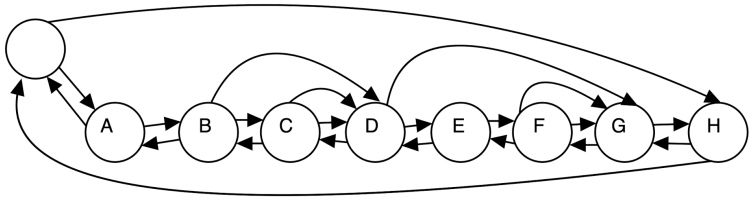

Figure 17: DeqUnion

In implementation, we add `Shape` type field `right`,`left` and `parent` to each class who implements Shape interface, as well as relative methods in `Shape` interface.

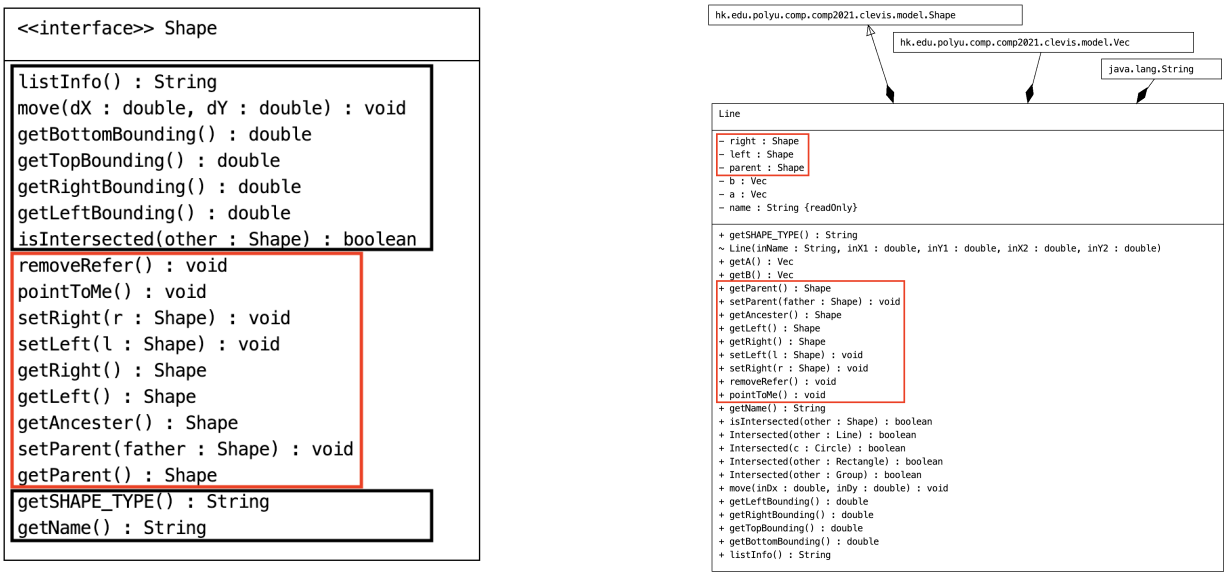

Figure 18: Shape Interface & Line Class

For three fields in red box:

- `right` : refer to the right node of `this`;
- `left` : refer to the left node of `this`;
- `parent` : refer to the parent node of `this`, default value is `this`;

For methods in red box:

- `removeRefer()` removes `this` out of the `LinkedListDeque` (see details in 3.8 REQ8);
- `pointToMe()` set its parent to itself.
- `setRight`, `setLeft` and `setParent` set three fields respectively;
- `getRight`, `getLeft` and `getParent` return three fields respectively;
- `getAncestor()` returns the top parent of `this` by iteratively get the parent:

```
public Shape getAncestor() {
    Shape ancestor = this;
    while(!ancestor.getName().equals(ancestor.getParent().getName()))
    {
        ancestor = ancestor.getParent();
    }
    return ancestor;
}
```

3 Requirements & Bonus

All requirements and bonus are implemented successfully on the CLI and GUI platform we designed. The implementation is encapsulated in `ClevisModel` Class and achieved by calling relative methods from `ClevisModel` Class. Creating `ClevisModel` object:

```
private ClevisModel clevisModel = new ClevisModel();
```

Below are requirements and bonuses implementation details.

3.1 REQ1 log file

1. The requirement is implemented successfully.
2. **For HTML format file**, we use `StringBuilder` to store the html structure string and `PrintStream` to print html code into a new log.html file in local directory. Then for each time we input command in line, it will record the data in scanner with `BufferReader` and write it in `StringBuilder` with `BufferWriter` to store the record with one line, until users input **quit** for asking store all the input command and upload `StringBuilder` to log.html eventually.
   For TXT format file, we also use `BufferReader` and `BufferWriter` to store the scanner data, however, it will upload to log.txt after recording the input command immediately. And user could check the content in local directory no matter which process is doing, even have not asked for quit.
   Last, we take parameters from `main(String[] args)` method, and set args[1] and args[3] as the name of the html and txt file, when the launching parameter is `"-html log.html -txt log.txt"`.
3. The view class set default file name to "log.html" and "log.txt" when the launching parameter is missing. If the file did not exist before, it will create a new file to store the information; if the file already exist, it will override the file with new data.

3.2 REQ2 rectangle n x y w h

1. The requirement is implemented successfully.

2. Invoking the `drawRectangle(inName:String, inX:double, inY:double, inW:double, inH:double)` of `clevisModel` object, taking the user inputs as parameters.

3. Throw exception with `IllegalArgumentException` if there already have the shape with same name to the new rectangle.

3.3 REQ3 line x1 y1 x2 y2

1. The requirement is implemented successfully.

2. Invoking the `drawLine(inName:String, inX1:double, inY1:double, inX2:double, inY2:double)` of `clevisModel` object, taking the user inputs as parameters.

3. Throw exception with `IllegalArgumentException` if there already have the shape with same name to the new Line.

3.4 REQ4 circle n x y r

1. The requirement is implemented successfully.

2. Invoking the `drawCircle(inName:String, inX:double, inY:double, inR:double)` of `clevisModel` object, taking the user inputs as parameters.

3. Throw exception with `IllegalArgumentException` if there already have the shape with same name to the new circle.

3.5 REQ5 square n x y l

1. The requirement is implemented successfully.

2. Invoking the `drawSquare(inName:String, inX:double, inY:double, inL:double)` of `clevisModel` object, taking the user inputs as parameters.

3. Throw exception with `IllegalArgumentException` if there already have the shape with same name to the new square.

3.6 REQ6 group n n1 n2 ...

1. The requirement is implemented successfully.

2. Invoking the `createGroup(inName:String, inShapeString:String[])` of `clevisModel` object, taking the user inputs as parameters including a String Array of Shape names input from the user.

3. Throw exception with `IllegalArgumentException` if there already satisfy following conditions:

(a) Trying to group a Shape **more than once** (e.g. `group mygroup lineA lineA lineB`).

(b) Trying to group shapes not contained in the storage.

(c) Trying to create a new group with a name having been contained in the storage.

(d) Trying to group shapes having been grouped in another group.

3.7 REQ7 ungroup n

1. The requirement is implemented successfully.
2. Invoking the `unGroup(inName:String)` of `clevisModel` object, taking the user inputs as parameters including a String of Shape name input from the user.
3. Throw exception with `IllegalArgumentException` if there already satisfy following conditions:

   (a) Trying to ungroup a group not contained in the storage.

   (b) Trying to ungroup a non-group Shape.

   (c) Trying to ungroup a group having been grouped in another group.

3.8 REQ8 delete n

1. The requirement is implemented successfully.
2. To remove a Shape from our storage system, we first need to remove it from our **LinkedListDeque**, and then need to remove it from our HashMap **storage**.

   For removing a Shape from **LinkedListDeque**:

   - If it is a single independent Shape, we let its left node's right pointer point to its right node, and let its right node's left pointer point to its left node. (See Figure 19)

```
removeRefer():
    getLeft().setRight(getRight())
    getRight().setLeft(getLeft())
```

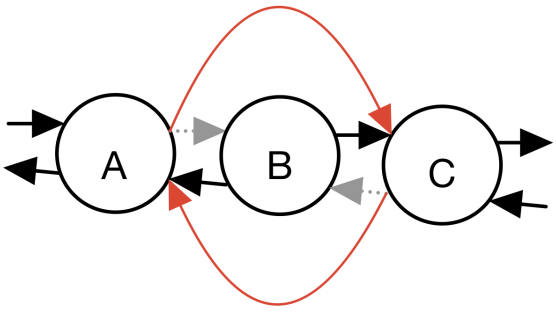

Figure 19: Example of B.removeRefer()

- If it is a Group, we use DFS postorder. Since a Group is a tree-like structure, it is unnecessary to use the memorization to optimize this procedure:

```
removeRefer():
    for (Shape a in shapeList):
        a.removeRefer()
    getLeft().setRight(getRight())
    getRight().setLeft(getLeft())
```

  We use DFS postorder because in the **undo delete** operation for a Group (add it to its previous z-order position in our LinkedListDeque), it is more convenient for us to first add back the Group, and then follow the references provided by its **shapeList** to add back its child. Therefore, the order we delete the Group and its child is crucial. This will keep the correctness of state transition.

For removing a Shape from **storage**:

- If it is a single independent Shape, we simply remove it from storage.
- If it is a Group, we use DFS postorder:

```
removeGroup(Shape name):
    if (name is an instance of Group):
        for (Shape item : name.shapeList):
            removeGroup(item)
    storage.remove(name)
```

3. Throw an IllegalArgumentException when:

   - Trying to delete a shape not contained in the storage.
   - Trying to delete a shape having been grouped in another group(i.e. that shape's parent is not itself)

3.9 REQ9 boundingbox n

1. The requirement is implemented successfully.
2. Invoking the `createBoundingBox(inName:String)` of `clevisModel` object, taking the user inputs as parameters to create a bounding box for a shape named `inName`.
3. Throw an IllegalArgumentException when:

   - Trying to create a bounding box for a shape not contained in the storage.
   - Trying to create a bounding box for a shape having been grouped in another group.

3.10 REQ10 move n dx dy

1. The requirement is implemented successfully.

2. Invoking the `moveShape(inName:String, inDx:double, inDy:double)` of `clevisModel` object, taking the user inputs as parameters to move a shape named `inName` by `inDx` and `inDy`.

3. Throw an IllegalArgumentException when:

   - Trying to move a shape not contained in the storage.
   - Trying to move a shape having been grouped in another group.

3.11 REQ11 pick-and-move x y dx dy

1. The requirement is implemented.

2. According to the requirement of the `pick-and-move` command [REQ11], a Shape containing that point indicates that the minimum distance between the point and the Shape outline is smaller than 0.05. Therefore we implement that point as a `Circle` object with radius of 0.05, and coordinates of `inX,inY`.

   ```
   private final double POINT_XY = 0.05d;
   Circle xyPoint = new Circle("xyPoint",inX,inY,POINT_XY);
   ```

   In the [REQ11], we have to pick the shape which contains that point, and which has the largest z-order within all shapes that are intersected with that point (if any), and move it, so what we have to do is to find the right-most Shape, that is intersected with the point, on the **LinkedListDeque**. If the right-most Shape is a Group, we need to move all its children as well. Knowing that no matter what method we use, we still need to traverse every **bottom-level non-group Shape**. Initial idea is to traverse the **bottom-level non-group Shape**, and once the intersected **Shape A** is found, we go to its **Ancestor** (may be **Shape A** itself) and cache the Ancestor to **finalShape**, and then go ahead. There may be other intersected Shape exist between **Shape A** and its **Ancestor**, but z-order of them is smaller than that of **Shape A's Ancestor**, so we ignore them for the purpose of pruning. An example is given as Figure 20. In this example, if **Shape A, E, H** are intersected with **xyPoint**, our traversal path, which skips **Shape E**, is **A** $\rightarrow$ **D** $\rightarrow$ **G** $\rightarrow$ **H**, and the expected one we need to find is Shape **H**. The time complexity of finding **H** is $O(n)$ with small constants.

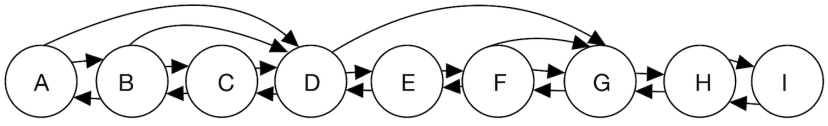

Figure 20: Pick and Move Traversal

The following is the implementation:

```
// the final picked Shape
Shape finalShape = xyPoint;

//start from the first shape(right node of the Sentinel)
Shape inShape = shapeLevel.getSentinel().getRight();

    while (inShape != shapeLevel.getSentinel()) {
        if (inShape.isIntersected(xyPoint) &&
            ! (inShape instanceof Group)) {
            finalShape = inShape.getAncester();
            inShape = inShape.getAncester().getRight();
        }
        else {
            inShape = inShape.getRight();
        }
    }
```

Finally, move that target shape by invoking

the `move(inDx:double,inDy:double)` method of the `Shape` interface.

3. If pick nothing, which means no shape contains that point, throw an IllegalArgumentException.

3.12 REQ12 intersect n1 n2

1. The requirement is implemented successfully.

2. Determining whether **Line AB** formed by point **A** (a_1, a_2) and point **B** (b_1, b_2), and **Line CD** formed by point **C** (c_1, c_2) and point **D** (d_1, d_2) are intersected: First of all, we can conduct the rapid exclusion test to quickly judge some situations that must not intersect and optimise the **intersect** operation. **False** $\rightarrow$ not intersected; **True** $\rightarrow$ intersected.

   Rapid exclusion test: return **False** when:

   - $max(a_1, b_1) < min(c_1, d_1)$, or
   - $max(a_2, b_2) < min(c_2, d_2)$, or
   - $max(c_1, d_1) < min(a_1, b_1)$, or
   - $max(c_2, d_2) < min(a_2, b_2)$

   If **AB** and **CD** are intersected, point **C** and **D** must be on both sides of **AB**, and point **A** and point **B** must be on the both sides of **CD**. In order word, for guaranteeing the intersection of **AB** and **CD**, if the rotation of vector $\overrightarrow{CD}$ to vector $\overrightarrow{CA}$ is clockwise, then

the rotation of vector $\overrightarrow{CD}$ to vector $\overrightarrow{CB}$ should be counterclockwise (i.e. the opposite of $\overrightarrow{CA}$, so $(\overrightarrow{CD} \times \overrightarrow{CA}) \cdot (\overrightarrow{CD} \times \overrightarrow{CB}) < 0$. The same situation holds for the counterpart of vector $\overrightarrow{AB}$.

Therefore, return **False** when:

- $(\overrightarrow{CD} \times \overrightarrow{CA}) \cdot (\overrightarrow{CD} \times \overrightarrow{CB}) > 0$
- $(\overrightarrow{AB} \times \overrightarrow{AC}) \cdot (\overrightarrow{AB} \times \overrightarrow{AD})) > 0$

If it is not the case mentioned above, then return **True**.

3. Determining whether Line **AB** formed by point **A** and point **B**, and Circle **O** with radius **r** are intersected can be classified to the following cases:

   - If both **A** and **B** are inside the Circle **O**, that is $|\overrightarrow{OA}| < r$ and $|\overrightarrow{OB}| < r$, return **False**.
   - If **A** or **B** are just on the Circle, that is $|\overrightarrow{OA}| = r$ or $|\overrightarrow{OB}| = |r|$, return **True**.
   - If one inside and one outside, that is $|\overrightarrow{OA}| < r$ and $|\overrightarrow{OB}| > r$ **OR** $|\overrightarrow{OA}| > r$ and $\overrightarrow{OB} < r$, return **True**.
   - If **A** and **B** are outside of **O**, then **AB** will **not** intersect with **O** when $\angle OAB > 180°$ or $\angle OBA > 180°$. This is because if one of them is obtuse, then $|OH| > r$ (See Figure 21). Therefore, if $\overrightarrow{AO} \cdot \overrightarrow{AB} < 0$ or $\overrightarrow{BO} \cdot \overrightarrow{BA} < 0$, then return **False**.
   - If **A** and **B** are outside of **O**, and are not the above cases, then we need to compute $|OH|$ and compare it with r. Therefore, if intersected, we have equation 1. Further, to reduce the loss of precision in floating-point number computation caused by division, we can transform the equation 1 into equation 2. Finally, if the equation holds, return **True**, otherwise return **False**.

$$\frac{|\overrightarrow{OA} \times \overrightarrow{OB}|}{|\overrightarrow{AB}|} = h \leq r \tag{1}$$

$$|\overrightarrow{OA} \times \overrightarrow{OB}| - r \cdot |\overrightarrow{AB}| \leq 0 \tag{2}$$

4. Determining whether Circle **A** with radius r_1 and center O_1 and Circle **B** with radius r_2 and center O_2 are intersected is to check both equation 3 and equation 4 holds. If holds, then return **True**, otherwise return **False**.

$$|O_1 \cdot O_2| - |r_1 - r_2| \geq 0 \tag{3}$$

$$|O_1 \cdot O_2| - |r_1 + r_2| \leq 0 \tag{4}$$

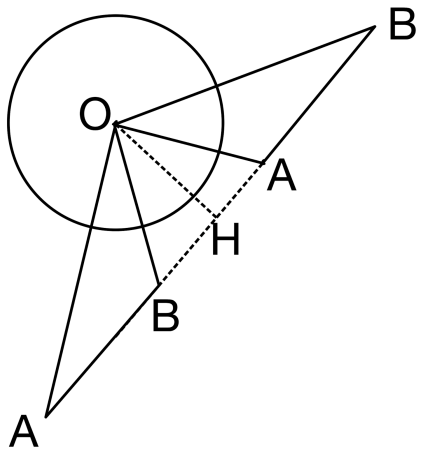

Figure 21: Obtuse ΔOAB

5. Throw an IllegalArgumentException when:

   - Trying to check intersection of two shapes not contained in the storage.
   - Trying to check intersection of two shapes having been grouped in another group.

3.13 REQ13 list n

1. The requirement is implemented successfully.
2. Invoking the `listShape(inName:String)` of `clevisModel` object, taking the user inputs as parameters to list out the information of a shape named `inName`.

   - Trying to list information of a shape not contained in the storage.

3.14 REQ14 listAll

1. The requirement is implemented successfully.
2. Invoking the `listAllShape()` of `clevisModel` object, to list out the information of all shapes in the storage.

3.15 REQ15 quit

1. The requirement is implemented successfully in Clevis(CLI) and GUI.
2. - In Clevis(CLI): Once user enter "quit", Clevis will be terminated (see details in 4.1 Clevis-CLI).
   - In ClevisGUI: Once user click "Quit" button, ClevisGUI will be terminated after the second confirmation (see details in 4.2 GUI).

3.16 BON1 GUI

1. The requirement is implemented successfully.
2. See implementation details in 2.4.2 ClevisGUI Class.
3. See error conditions in 4.2 GUI User Manual.

3.17 BON2 undo & redo

1. The requirement is implemented.
2. In order to achieve undo and redo function, it is essential to:
   (a) **store commands** input by use in a specific order;
   - as described in 2.3.3 Undo & Redo Function Section, we store commands input from the user into the `cmdStack`(see Figure 22), including the command name(e.g. "rectangle" for draw a rectangle) and parameters (e.g. rectangle name, x, y, w, h); commands to redo in `cmdRedoStack`; deleted Shapes into the `delTargets` Stack; and redo delete shape to into `delRedoTargets` Stack.

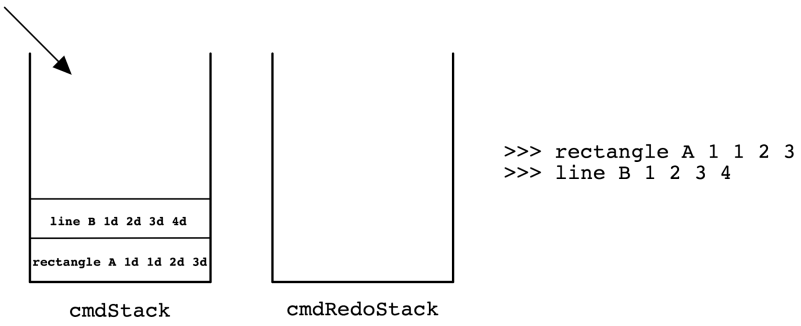

Figure 22: cmdStack & cmdRedoStack Stack

- Example for storing **rectangle** command:

```
String[][] cmdStr = new String[6][1];
cmdStr[0][0] = "rectangle";
cmdStr[1][0] = inName;
cmdStr[2][0] = String.valueOf(inX);
cmdStr[3][0] = String.valueOf(inY);
cmdStr[4][0] = String.valueOf(inW);
cmdStr[5][0] = String.valueOf(inH);
cmdStack.push(cmdStr);
```

- Example for storing **delete** command:

```
String[][] cmdStr = new String[3][1];
cmdStr[0][0] = "delete";
cmdStr[1][0] = inName;
delTargets.push(storage.get(inName));
```

(b) **execute corresponding actions** (forwards or backwards);

- Before implementing the undo and redo action, it is essential to be clear with the undo redo performance of each command:
- Undo (act backwards):
  - `rectangle`, `line`, `circle`,and `square`:
    delete the created shape, by invoking the `deleteShapeWithName()` method in `ClevisModel` class.
  - `group`:
    ungroup the created group, by invoking the `unGroup()` method in `ClevisModel` class.
  - `ungroup`:
    group again the ungrouped group, by invoking the `createGroup()` method in `ClevisModel` class.
  - `move` and `pick-and-move` :
    move back the moved shape, by invoking the `move()` method in `ClevisModel` class.
  - delete:
    add back the deleted shape back to its original position(to keep Z-order unchanged)
    Note that, in one of the internal storage `storage` HashMap, we **fully remove** the pair of name and shape,
    however, in another `shapeLevel LinkedListDeque`, we did not remove the Shape fully, **only changed its left and right nodes** to let it be invisible (see in 3.8 REQ8).
    Therefore, to "add" the "deleted" shape back to its original place, we only need to rearrange the "deleted" shape's left and right nodes to let it be visible again without changing its Z-order, and directly put the deleted Shape (and its grouped shapes if the shape is a group) to the `storage` HashMap because no order in HashMap.
    Implementation:

```
if (!(inShape instanceof Group)) {
// directly put the deleted Shape to the storage
    storage.put(inName, inShape);
// rearrange shape's left and right nodes
    inShape.getLeft().setRight(inShape);
```

```
            inShape.getRight().setLeft(inShape);
        }
        else { // for group shape
        // recursively do the above steps for shapes grouped
            recursionUndoDel(inShape);
            recursionAdd(inShape);
        }
```

- Redo (act forwards):
  Simply do the command again.

(c) **clear unreachable commands in cmdRedoStack**.

It is important that once user enter **normal command** (not undo and redo) immediately after **undo or redo command**, user can not call redo command because of the command ambiguity. Therefore, in this situation, `cmdRedoStack` should be cleared for it is unreachable any more.

Implementation:

```
if ((undoFlag || redoFlag) &&
        (cmdStack.isEmpty()||
        (!cmdStack.peek()[0][0].equals("undo") &&
        (!cmdStack.peek()[0][0].equals("redo"))))) {
        cmdRedoStack.clear();
        undoFlag = false;
        redoFlag = false;
    }
```

The above code can also ensure the operation called by undo and redo will NOT be recorded in the `cmdStack`.

3. Detailed undo implementation(see Figure 23):

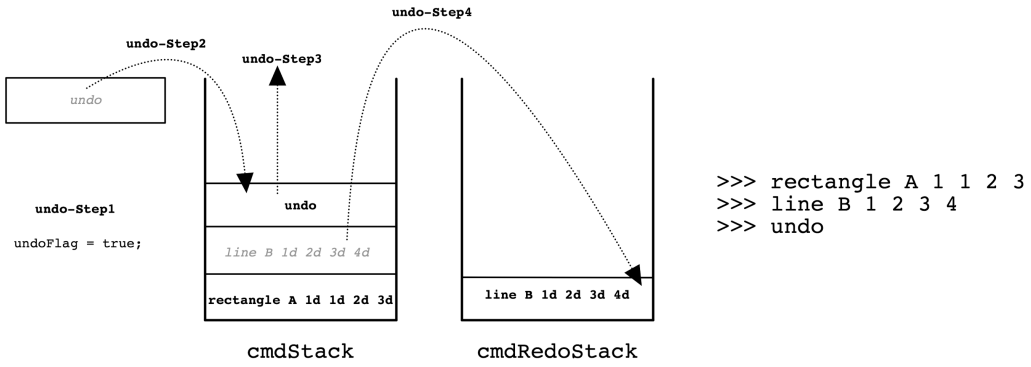

Figure 23: cmdStack & Undo steps

(a) Step1: set `undoFlag` to true for indicating undo command have been performed;

(b) Step2: push "undo" command into `cmdStack` temporarily for indicating latest command is "undo";

(c) Step3: according to the second-top element in the `cmdStack`(skip "undo" to get the top normal command need to undo), do the backwards operation (based on above **execute corresponding actions** section). Then check whether need to clear unreachable commands in cmdRedoStack. Finally pop the temporary "undo" command out.

(d) Step4: pop the peak command from the `cmdStack`, and push it to the `cmdRedoStack` for further redo operations.

4. Detailed redo implementation(see Figure 24):

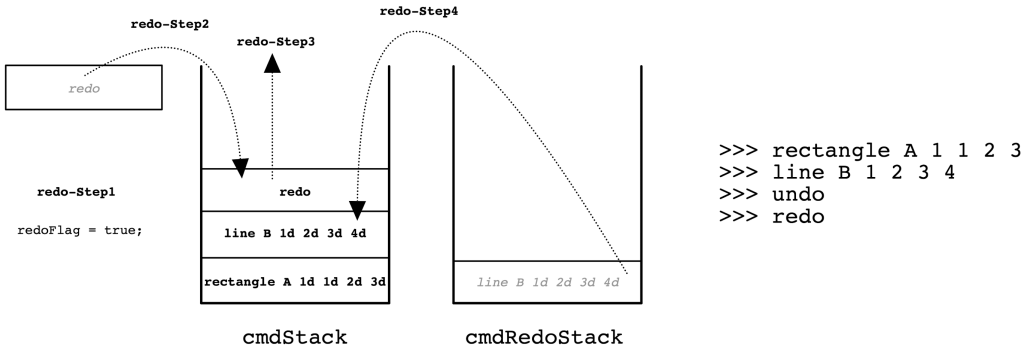

Figure 24: cmdStack & Redo steps

(a) Step1: set `redoFlag` to true for indicating redo command have been performed;

(b) Step2: push "redo" command into `cmdStack` temporarily for indicating latest command is "redo";

(c) Step3: according to the peek element in the `cmdRedoStack`, do the forwards operation (based on above **execute corresponding actions** section). Then check whether need to clear unreachable commands in cmdRedoStack. Finally pop the temporary "redo" command out.

(d) Step4: pop the peak command from the `cmdRedoStack`, and push it to the `cmdStack` for further undo operations.

5. Throw an IllegalArgumentException when:

- Trying to **undo** while no command can be undid any more, which means the `cmdStack` is empty.
- Trying to **redo** without any **undo** command called immediately before it, which means the `cmdRedoStack` is empty.

4 User Manual

To launch:

Run `Application` class. Type **0** to select Clevis(CLI), or **1** to select ClevisGUI. Press enter to launch.

4.1 Clevis(CLI)

After choosing to start the `Clevis` program, it will show the title and storage document information. The storage document is stored into local root directory of the `Clevis` with users input commands in terminal no matter whether the codes run or not. When `"Please type your code:"` exists, which means typing some commands to start the graphics functions.

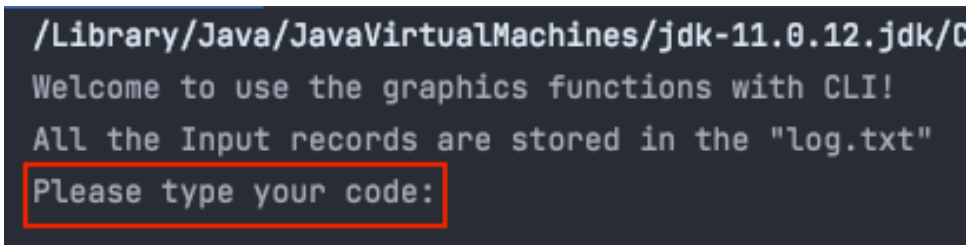

Figure 25: Welcome page of CLI

To draw Rectangle, input the command:

rectangle [name] [x of top-left corner] [y of top-left corner] [width] [height]

Press enter, see the specific information about the new rectangle information.

Figure 26: Successful case for Rectangle Draw

To draw Line, input the command:

line [name] [x of end A] [y of end A] [x of end B] [x of end B]

Press enter, see the specific information about the new line information.

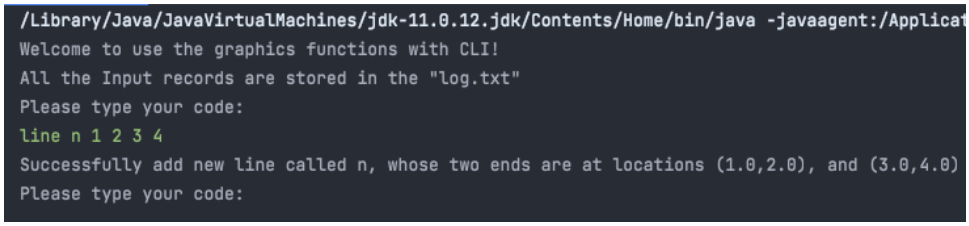

Figure 27: Successful case for Line Draw

To draw Circle, input the command:

circle [name] [x of center] [y of center] [radius]

Press enter, see the specific information about the new circle information.

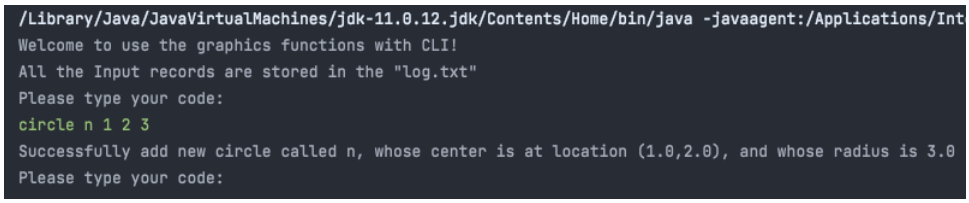

Figure 28: Successful case for Circle Draw

To draw Square, input the command:
square [name] [x of top-left corner] [y of top-left corner] [side length]
Press enter, see the specific information about the new square information.

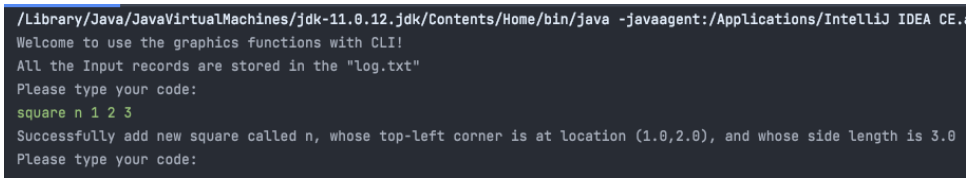

Figure 29: Successful case for Square Draw

With graphics created, modification functions can be performed on them.
To Group, input the command:
group [group name] [name of shape need to group]...
Press enter, see the prompt for successfully group.

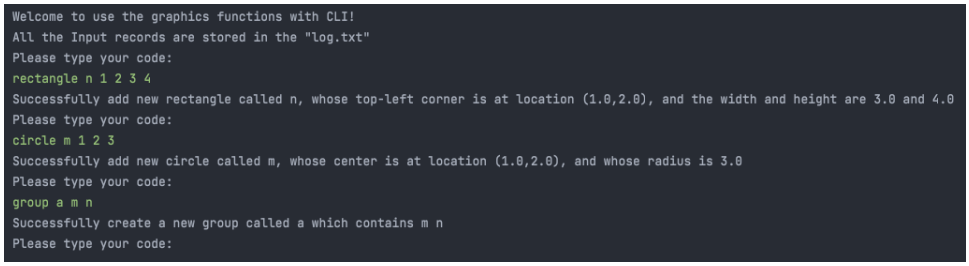

Figure 30: Successful case for Group function

To Ungroup, input the command:
ungroup [group name]
Press enter, see the prompt for successfully ungroup.

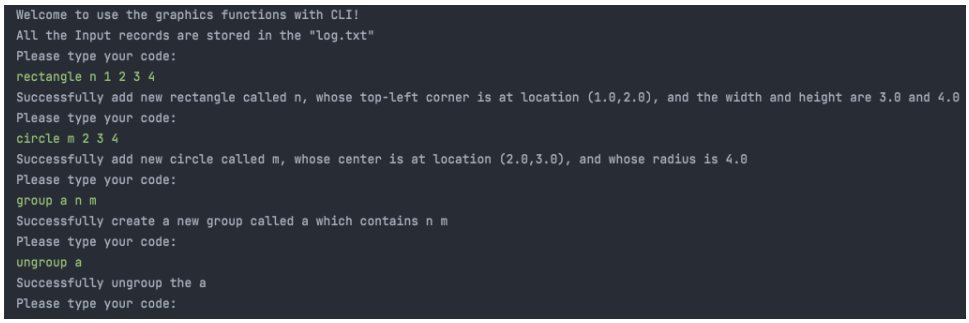

Figure 31: Successful case for Ungroup function

To Delete, input the command:
delete [shape name]
Press enter, see the prompt for successfully delete.

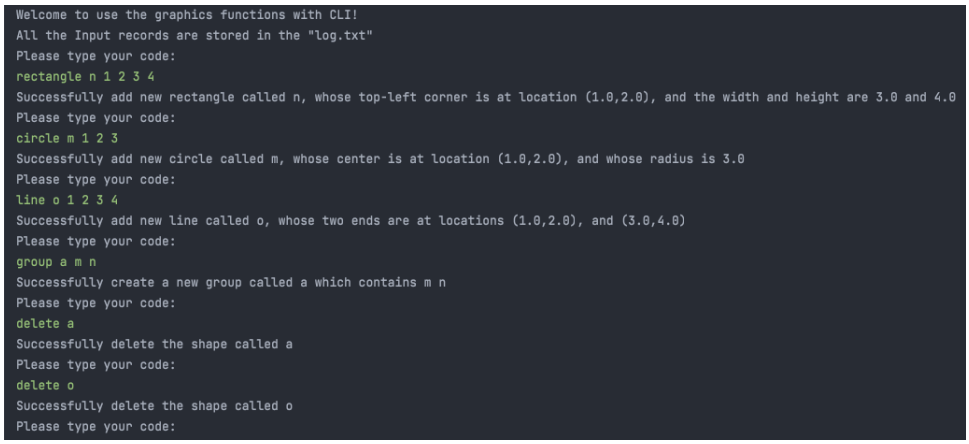

Figure 32: Successful case for delete

To get boundingbox, input the command:
boundingbox [shape name]
Press enter, see the specific information about the new boundingbox information.

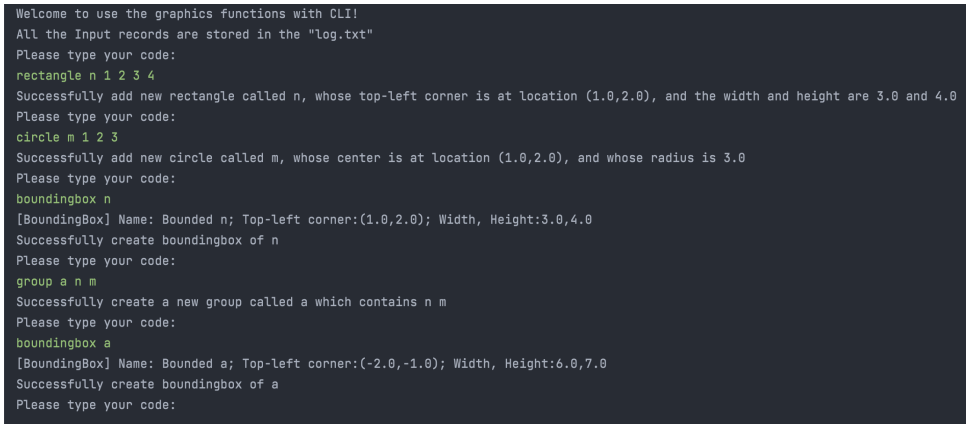

Figure 33: successful case for boundingbox

To Move, input the command:
move [shape name] [moving distance on x] [moving distance on y]
Press enter, see the specific moving information.

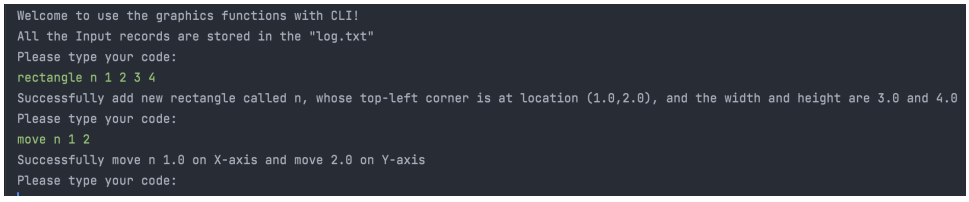

Figure 34: successful case for move

To Pick point and move, input the command:

pick-and-move [x of picked point] [y of picked point] [moving distance on x] [moving distance on y]

Press enter, see the specific pick-and-move information.

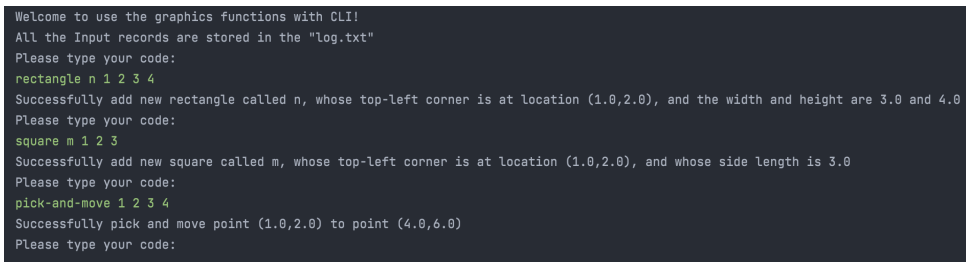

Figure 35: successful case for pick-and-move

To check intersection, input the command:

intersect [shape1 name] [shape2 name]

Press enter, see the specific information about intersection checking result.

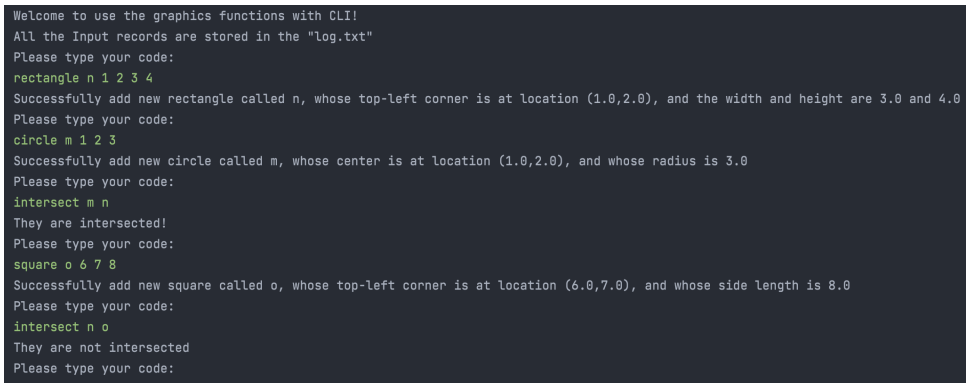

Figure 36: successful case for intersect

To list information, input the command:

list [shape name]

Press enter, see the specific information about shape.

Figure 37: successful case for list

To list all information, input the command:

listAll

Press enter, see the specific information of all exist shapes.

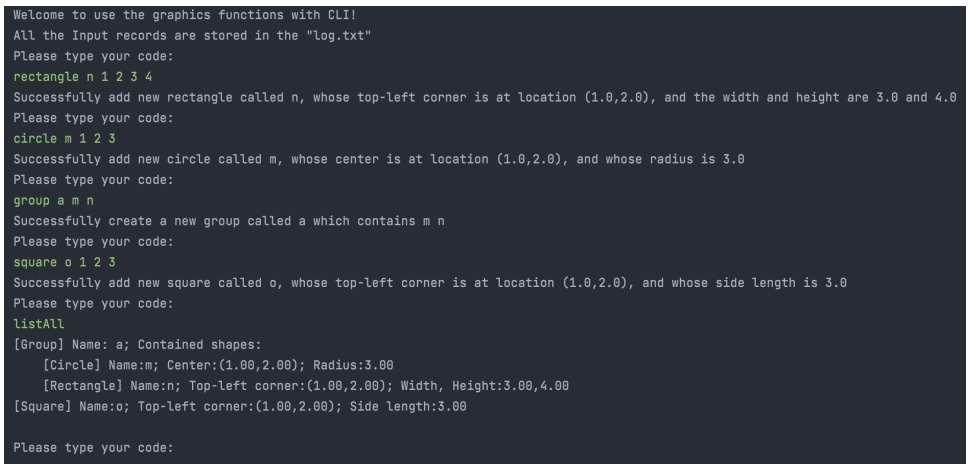

Figure 38: Successful case for listAll

To Undo, input the command:
undo
Press enter, see the prompt of successfully undo.

Figure 39: Successful case for undo

To Redo, input the command:
redo
Press enter, see the prompt of successfully redo.

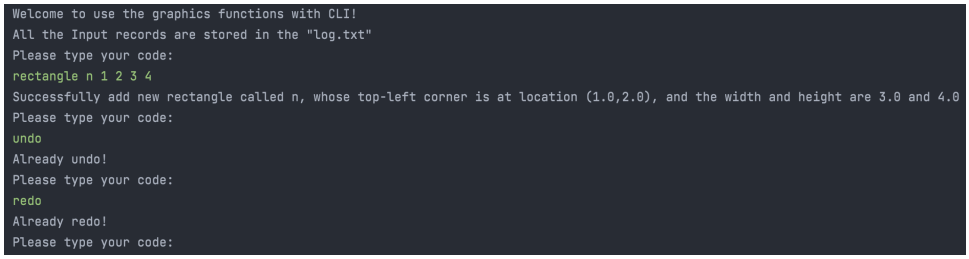

Figure 40: Successful case for redo

Error situations for input:

- Only input function word will throw `NoSuchElementException` and ask user to remember parameter and data and input again. (Contains other fundamental function except `listAll, undo, redo, quit`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle
Error for java.util.NoSuchElementException. You need input data to realize the function!
Please type your code:
```

- Lack or redundant parameters for functions will throw `IllegalArgumentException` and ask user to input again. (Contains other fundamental and extra functions except `quit`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle n 1 2 3 4 5
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
rectangle n 1 2 3
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
```

  - `quit` function no matter whether parameters behind but only need to detect the "quit" as function word.
  - `delete, boundingbox and list` functions could only delete/boundingbox/list one shape/group shape at the same time.

- Capitalize the function word and incorrect format will ask user to input again. (Contains other fundamental and extra functions except `quit()`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
Rectangle n 1 2 3 4
Please type your code:
rectangle n 1 2 3 4
Successfully add new rectangle called n, whose top-left corner is at location (1.0,2.0), and the width and height are 3.0 and 4.0
Please type your code:
pick and move 1 2 3 4
Please type your code:
```

  - Input of `listAll` and `quit` functions will ignore the upper and lower case which only needs input correct function word for calling functions.
  - When using `pick-and-move` function, user must input the dash among the function word with lower case.

- Already exist the parameter(name) among shapes will throw `IllegalArgumentException` and ask user to input again. (Contains functions of `rectangle, line, circle, square, group`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle n 1 2 3 4
Successfully add new rectangle called n, whose top-left corner is at location (1.0,2.0), and the width and height are 3.0 and 4.0
Please type your code:
rectangle n 2 3 4 5
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
```

- Could not find the parameter(name or coordinate) among exist shapes will throw `IllegalArgumentException` and ask user to input again. (Contains function of `group`, `ungroup`, `delete`, `boundingbox`, `move`, `pick-and-move`, `list`, `intersect`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle n 1 2 3 4
Successfully add new rectangle called n, whose top-left corner is at location (1.0,2.0), and the width and height are 3.0 and 4.0
Please type your code:
group a n m
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
```

  - The parameter of `ungroup` function is the group shape. If the shape have not been grouped, it could not been ungrouped.
  - If can not find any shape/group shape contains target coordinate, `pick-and-move` function will throw exception.

- The shape use more than once in function will throw `IllegalArgumentException` and ask user to input again. (Contains function of `group`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle n 1 2 3 4
Successfully add new rectangle called n, whose top-left corner is at location (1.0,2.0), and the width and height are 3.0 and 4.0
Please type your code:
group a n n
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
```

  - `intersect` function allows the shape use more than once which means shape intersect with itself and always return true.

- Operation on dependant shape which is including in group will throw `IllegalArgumentException` and ask user to input again. (Contains functions of `group`, `ungroup`, `delete`, `boundingbox`, `move`, `intersect`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle n 1 2 3 4
Successfully add new rectangle called n, whose top-left corner is at location (1.0,2.0), and the width and height are 3.0 and 4.0
Please type your code:
group a n
Successfully create a new group called a which contains n
Please type your code:
group b n
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
```

- Use `undo or redo` function when there is no process to undo or redo will throw `IllegalArgumentException` and ask user to input again. (Contains function of `undo`, `redo`)

```
Welcome to use the graphics functions with CLI!
All the Input records are stored in the "log.txt"
Please type your code:
rectangle n 1 2 3 4
Successfully add new rectangle called n, whose top-left corner is at location (1.0,2.0), and the width and height are 3.0 and 4.0
Please type your code:
undo
Already undo!
Please type your code:
undo
Error for: java.lang.IllegalArgumentException. Please find error situation in User Manual
Please type your code:
```

4.2 ClevisGUI

The graphical user interface(**GUI**) provides the whole functions of project, containing **draw, group/ungroup, delete, make boundingbox, move, intersection, list, undo/redo and Quit**. The upper half of frame which is the draw area with a default size of **600*300** where all the operations results will be showed.

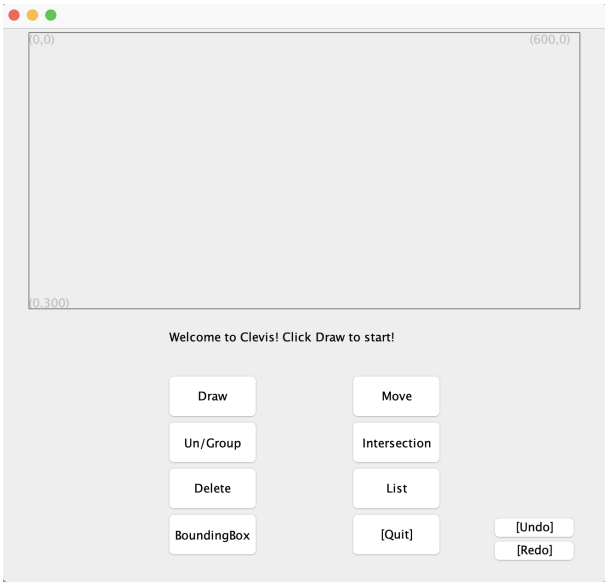

Figure 41: Main page of GUI

To draw graphics:

Click the **Draw** button on the main page to go to draw page. Choose which graphic want to draw. Input corresponding parameters. Press **DrawShape** button to print out.

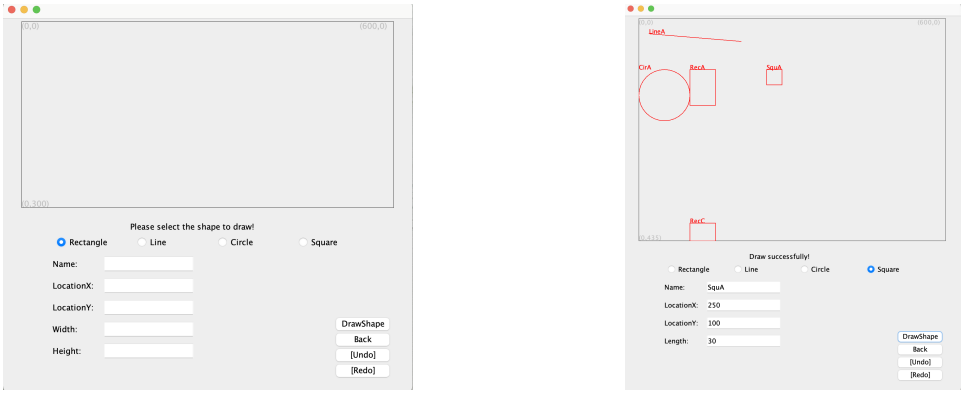

Figure 42: Draw

Error situation:

- When inputting the **same name** rather than a new name or input **none-integer** parameters, the label will show the error message.

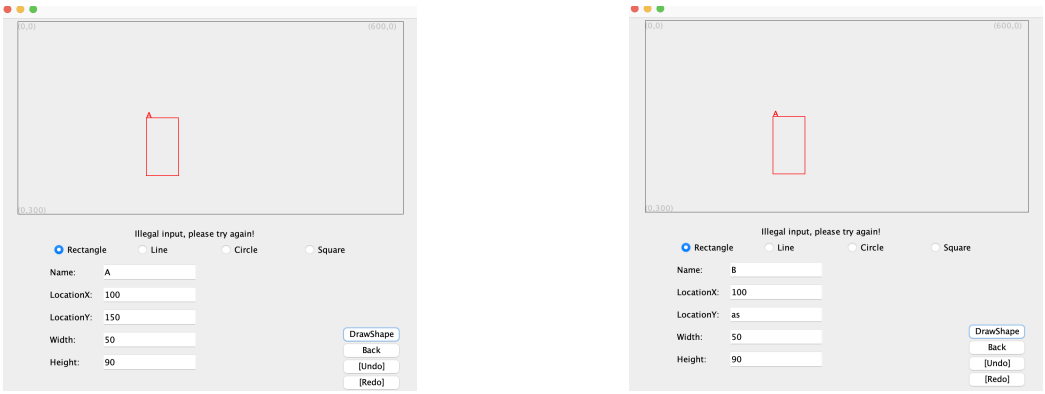

Figure 43: Draw error

To group/ungroup:

Click the **Un/Group** button on the main page to go to Un/Group page.

To Group: Input name of the new group, name(s) of shape to be grouped. Press **Group** to Group.

To UnGroup: Input the group name. Press **Ungroup** to ungroup.

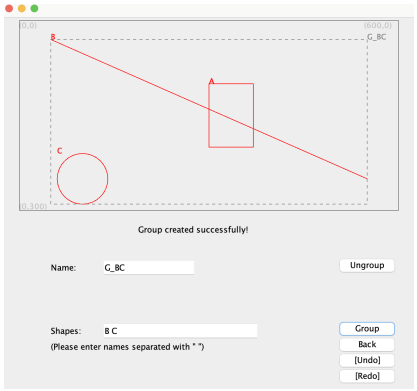

Figure 44: Un/Group

`Error situation`:

- When inputting **without a blank** between graphic name, the error message will be shown. Also, the **wrong** group name or **no** group name will make error when doing ungroup.

Figure 45: Unsuccessful Group & Ungroup

To delete graphics:

Click the **Delete** button on the main page to go to Delete page.

Enter the name of the shape to delete and press **Delete**.

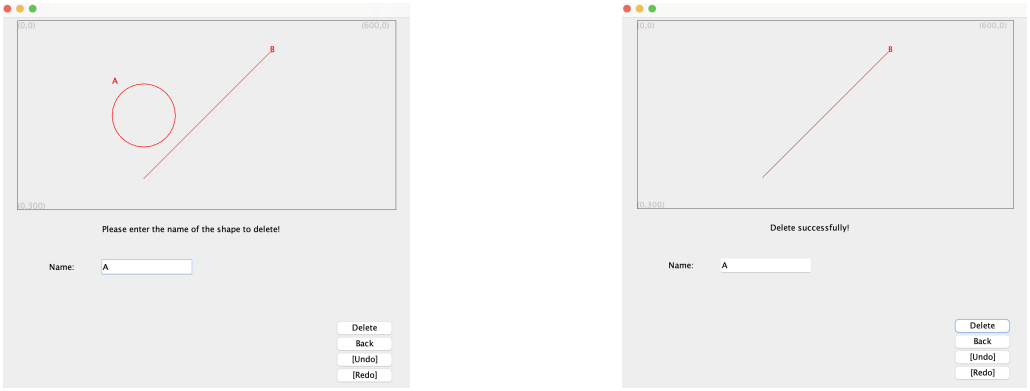

Figure 46: Successful Delete

- Grouped shapes can not be deleted.

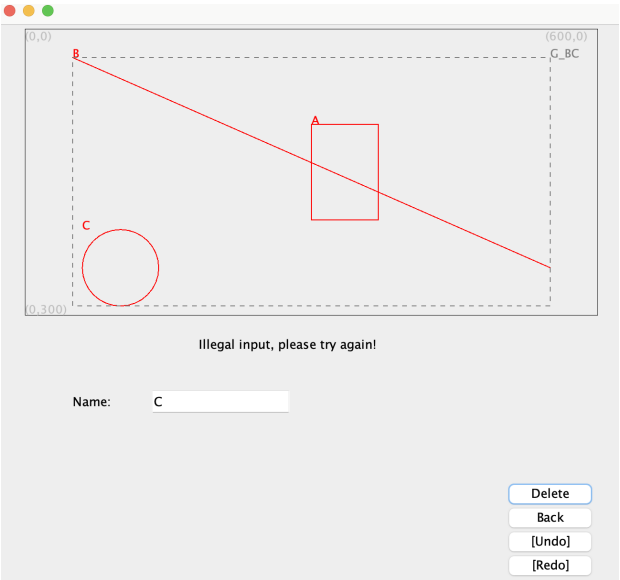

Figure 47: Unsuccessful Delete

To make bounding box:

Click the **BoundingBox** button on the main page to go to BoundingBox page.

Enter name of the shape and press **Bounding** to make bounding box.

The bounding box will be shown as a blue rectangle.

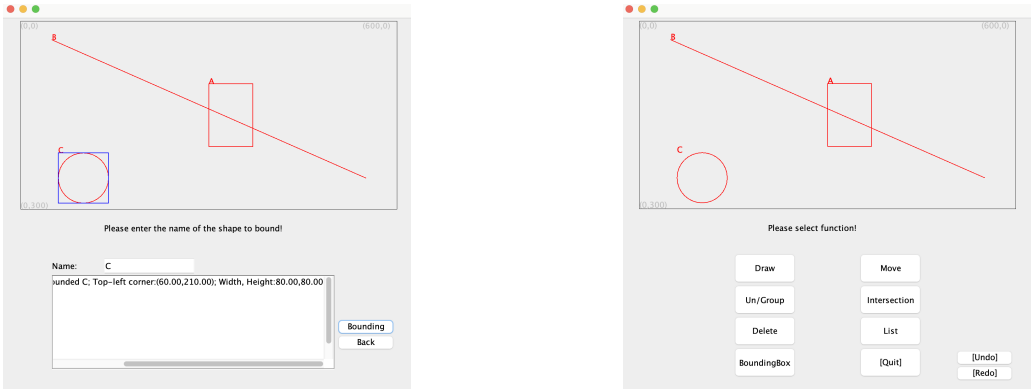

Figure 48: BoundingBox

Grouped shapes can not be bounded

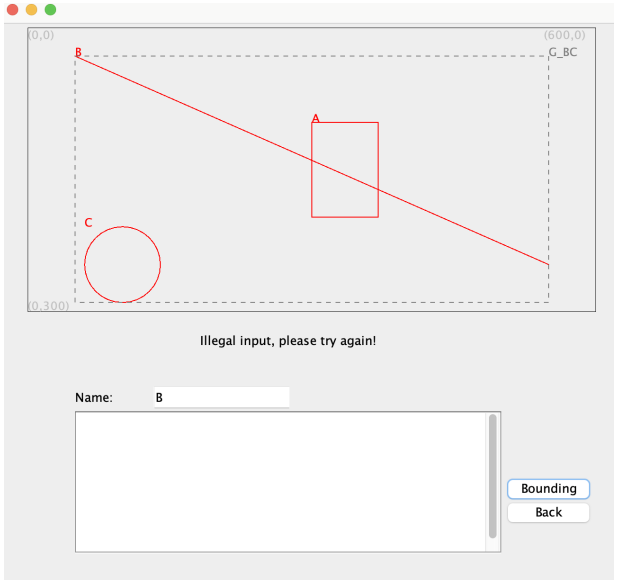

Figure 49: Unsuccessful Bounding Box

To move graphics:

Click the **Move** button on the main page to go to Move page.

Input name, dX and dY, press **Move** to move.

Or, input a **Point** with dX and dY, press **Pick&move** to pick and move.

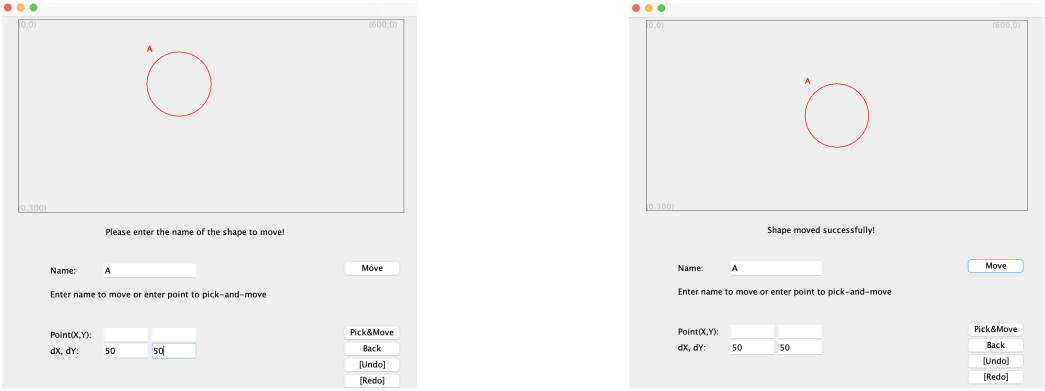

Figure 50: Successful move

`Error situation`:

- Grouped shapes can not be moved.

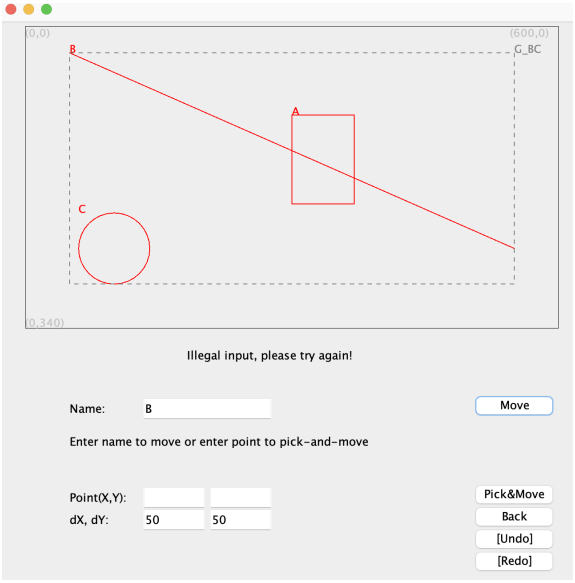

Figure 51: Unsuccessful Move

- **If the point does not locate in any graphic, there is no graphic will be picked**, and the label after draw area will show the error message.

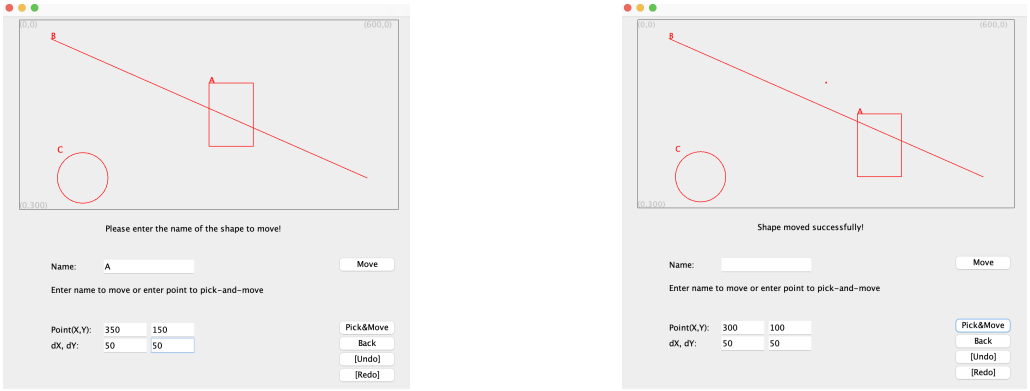

Figure 52: Unsuccessful and Successful pick&move

- If the point does not locate on the **border** of graphic, the error will occur.

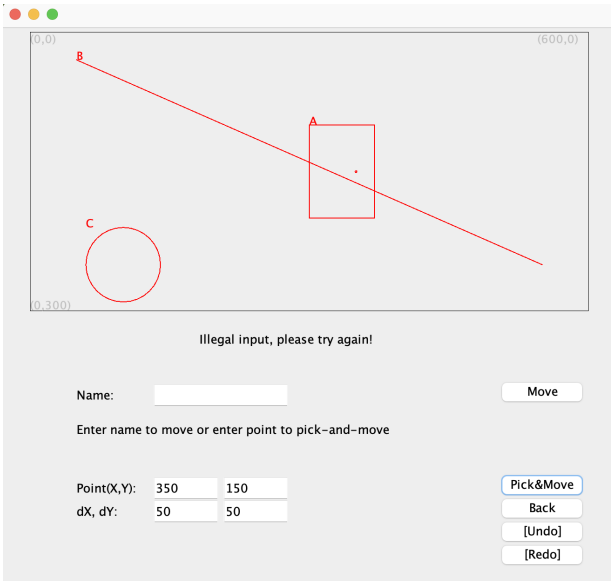

Figure 53: Unsuccessful pick &move

- Grouped shapes is able to pick &move

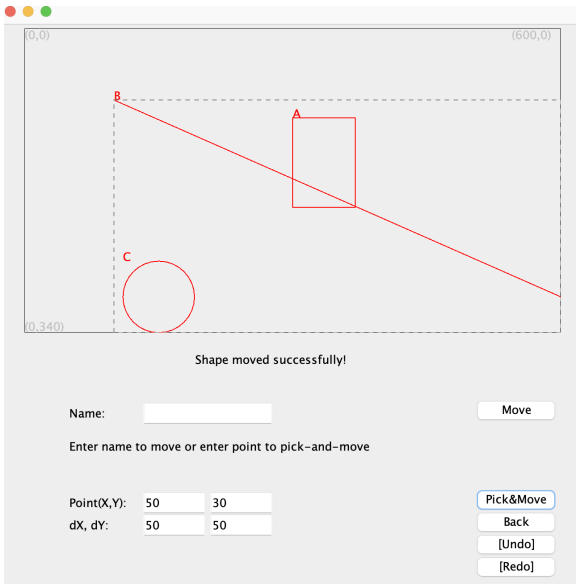

Figure 54: Successful pick &move for a group

To check graphics intersection :
Click the **Intersection** button on the main page to go to Intersection page.
Input names and press **Check** to check intersected or not.
`Error situation`:

- **If the shapes are in a group, the intersection can not be accessed**. Unless it has been **ungrouped**.

Figure 55: Successful & Unsuccessful Intersection

To list graphic information:

Click the **List** button on the main page to go to List page.

Input the name and press **list** to list. Or pressing **listAll** to all graphics.

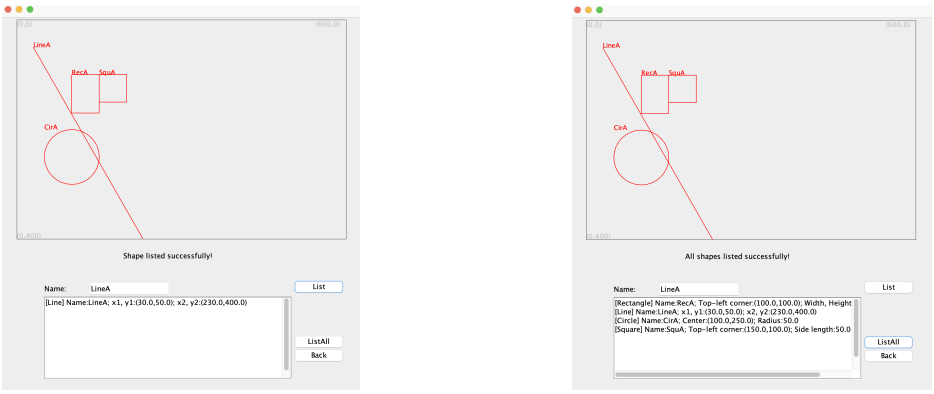

Figure 56: List

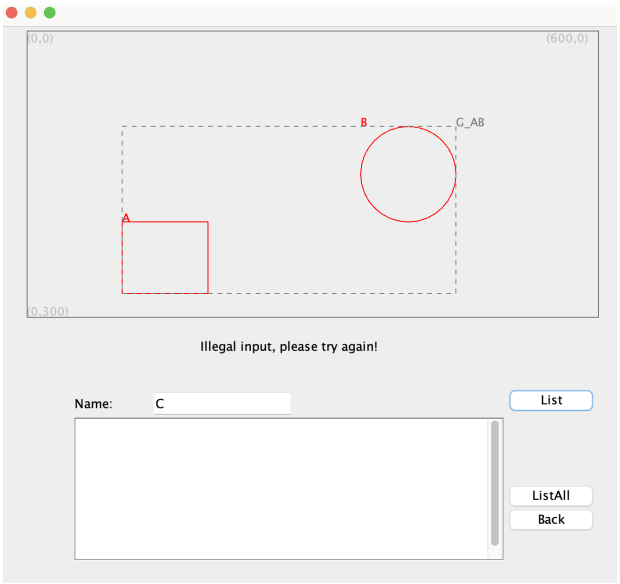

Figure 57: Unsuccessful list

For **most function**, **no name or wrong name** input will cause error.

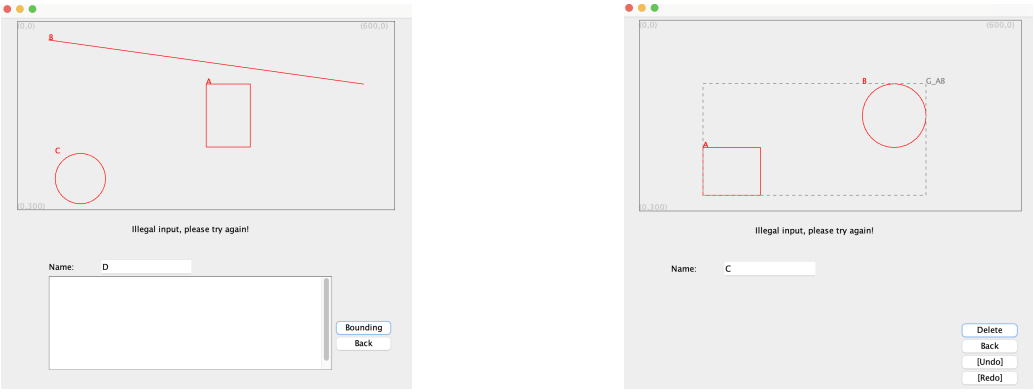

Figure 58: Error

To Undo/Redo:

Press [**Undo**] to undo/ Press [**Redo**] to redo.

Available for all commands except boundingbox,intersect, list, listAll, and quit.

Figure 59: Redo &Undo for moving RecA

To Quit:

Press [**Quit**] and press [**QUIT**] on the confirmation dialog to terminate, or press **CANCEL** to cancel quit.

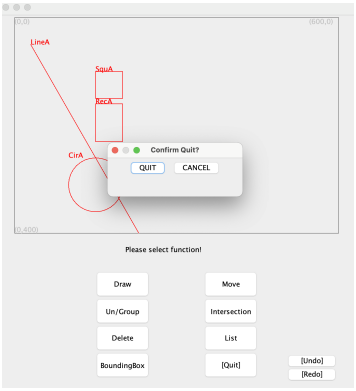

Figure 60: Confirm page of quit

References

[1] D. Liu, Z. Cui, S. Xu, and H. Liu, "An empirical study on the performance of hash table," in *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*. IEEE, 2014, pp. 477–484.

[2] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 319–344, 1991.

[3] R. Sedgewick and K. Wayne, *Algorithms*. Addison-wesley professional, 2011.

[4] J. Hug, *Sentinel Nodes*. University of California, Berkeley, 2021. [Online]. Available: https://sp21.datastructur.es/materials/lectures/lec4/lec4