JPaint

Project Report

Andrew McShane

SE 450

11/15/2018

Contents:

Features:

Draw Shape:

- Select the Draw mode In the Point to Point mode, then drag anywhere on screen and release to draw a shape in that space. Additional shape options can be chosen on the top bar.

Select Shape(s):

- Set the Point to Point mode to Select, then drag **around the entire shape** to select that shape. Dragging around multiple shapes will select all those shapes.

Move Shape(s):

- Select the range of shapes while in Select mode, then switch to the Move Point to Point mode. Clicking, dragging, and releasing *anywhere on the screen* will move the selected range by the distance the mouse has been dragged. The Selected range will move as long as the user is clicking, dragging, and releasing while in Move mode.

Copy/Paste:

- Select the range of shapes while in Select mode, then press copy to duplicate the contents of that selection to the clipboard. To paste the shapes, click the Paste button, which will place the new selection at an offset proportional to the size of the selected range.

Undo/Redo:

- Pressing the Undo and Redo button will respectively undo and redo the actions made. Making a new change will remove any prior ability to redo an action.

Delete:

- With a selected range made in Select mode, click the delete button to remove that selection from the screen.

Bugs:

No known bugs are present at the current state of implementation.

Extra Implementations:

Unit Tests have been provided for the following Classes:

- BoundingBox
- PointToPointVectorBuilder
- IShape implementations: makeClone() method.
- ShapeList

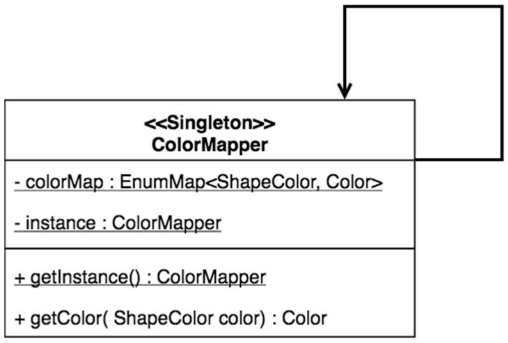
5 Design patterns were utilized which will be gone over in more detail later:

- Singleton
- Builder
- Factory
- Command
- Observer
- Facade[1]

[1] Though we did not go over Façade in class, I felt that the implementation of it in my design allowed a great deal of flexibility for future iterations. Although, I felt this was not distinguishing enough of an implementation feature to include a class diagram of in the analysis.

Design Analysis:

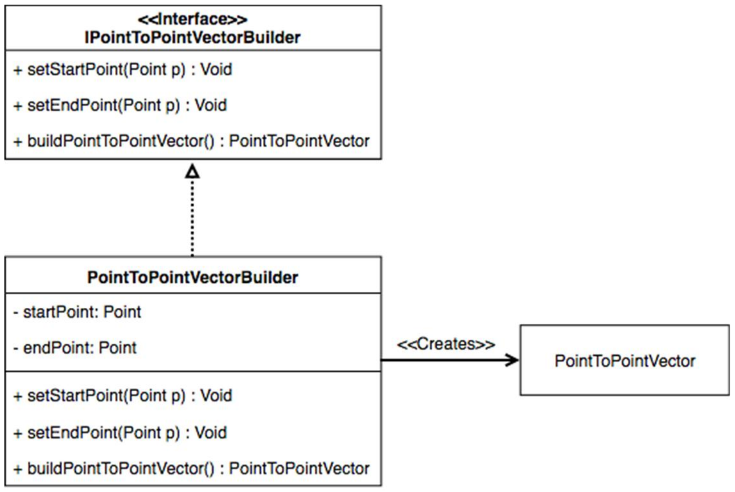
Singleton Pattern:



<<Singleton>>
ColorMapper
- colorMap : EnumMap<ShapeColor, Color>
- instance : ColorMapper
+ getInstance() : ColorMapper
+ getColor(ShapeColor color) : Color

In order to translate the enumerated value *ShapeColor* into a *Java.awt.Color* class, I felt that it was most appropriate to create a single class which is responsible for the management of this that can be accessed anywhere to keep dependency injection light and help the project stay simple. When Drawing a shape onto the screen, the *ShapeColor* value is passed into the *ColorMapper* instance and the appropriate *Java.awt.Color* is returned.

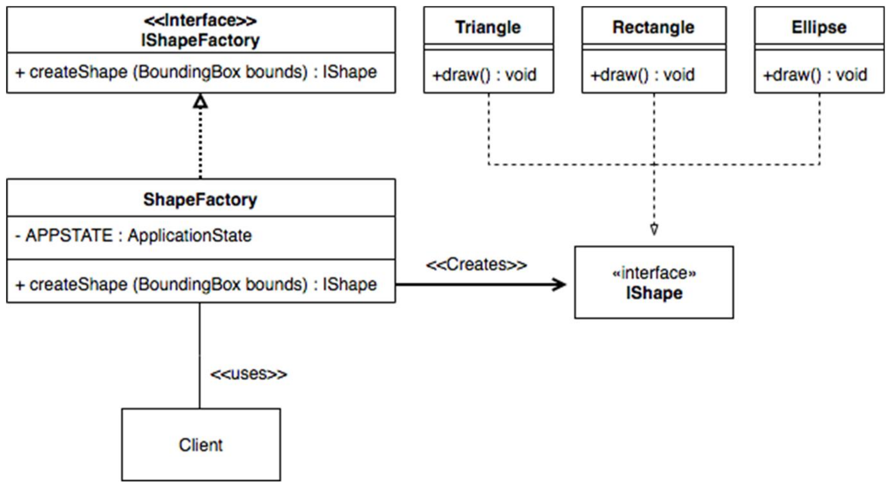
Builder Pattern:



<<Interface>>
IPointToPointVectorBuilder
+ setStartPoint(Point p) : Void
+ setEndPoint(Point p) : Void
+ buildPointToPointVector() : PointToPointVector
PointToPointVectorBuilder
- startPoint: Point
- endPoint: Point
+ setStartPoint(Point p) : Void
+ setEndPoint(Point p) : Void
+ buildPointToPointVector() : PointToPointVector
<<Creates>>
PointToPointVector

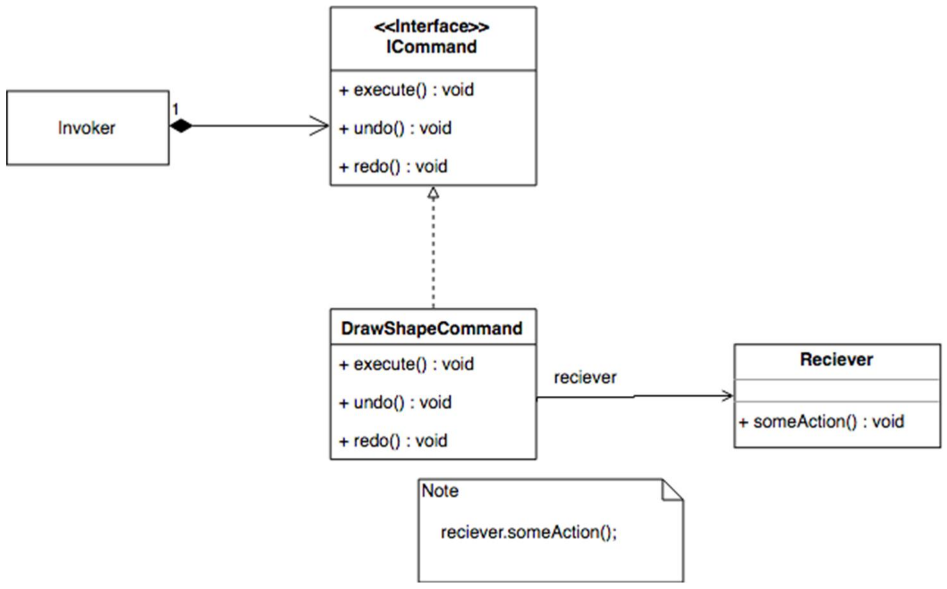
When attempting to get mouse pointer data on the canvas, I wanted a way to cleanly obtain a *PointToPointVector*, which is at its root a 2D displacement vector; this makes it an ideal class to handle move commands and be translated into a *BoundingBox*. However, out of a necessity to not have to hold onto 2 Point member variables in the class as well as handling its construction, I moved the responsibility off to a builder class, which the *MouseAdapter* class utilizes to construct canvas data.

Factory Pattern:



The Factory pattern was a logical choice to use for shape instantiation, as the shape to create is contingent on what the current application state is i.e., the type of shape to use, the shading type, colors, and so on. Offloading this work into a factory class allows the *MouseHandlerFacade* to easy create new shapes, and allows more shapes to be easily added by simply extending the *ShapeFactory* class.

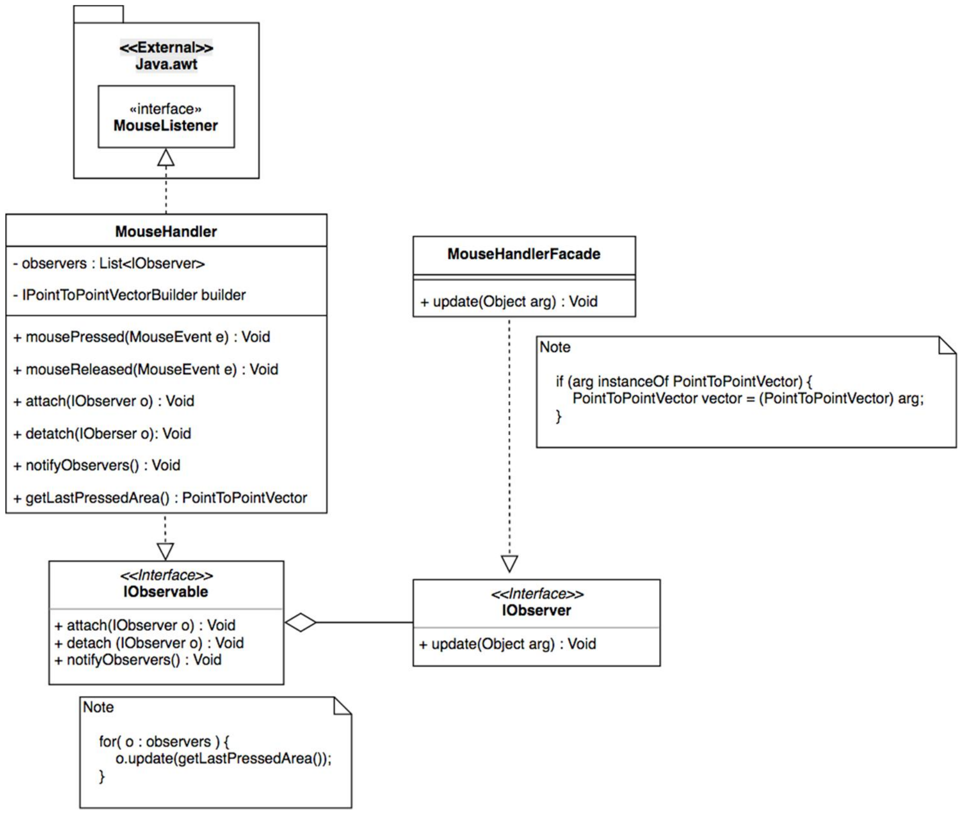
Command Pattern:



This particular example is showing how the *DrawShapeCommand* was implemented, though the rest of the commands which implement ***ICommand*** also implement this pattern.

The Command pattern was ideal for the drawing, deleting, copying, pasting, undoing, and redoing of shape drawings as it allowed the interface to be able to call on these commands with minimal knowledge of how the command is implemented, as to allow for the greatest amount of flexibility in the implementation, and allow more commands to easily be implemented into the design.

Observer Pattern:



<<External>>
Java.awt
«interface»
MouseListener
MouseHandler
- observers : List<IObserver>
- IPointToPointVectorBuilder builder
+ mousePressed(MouseEvent e) : Void
+ mouseReleased(MouseEvent e) : Void
+ attach(IObserver o) : Void
+ detatch(IOberser o): Void
+ notifyObservers() : Void
+ getLastPressedArea() : PointToPointVector
MouseHandlerFacade
+ update(Object arg) : Void
Note
if (arg instanceOf PointToPointVector) {
PointToPointVector vector = (PointToPointVector) arg;
}
<<Interface>>
IObservable
+ attach(IObserver o) : Void
+ detach (IObserver o) : Void
+ notifyObservers() : Void
<<Interface>>
IObserver
+ update(Object arg) : Void
Note
for(o : observers) {
o.update(getLastPressedArea());
}

Though the Observer pattern was used additionally for the *ShapeList* class to communicate to the *DrawShapeHandler*, I felt it more appropriate to focus on its usage in the *MouseHandler* class, as it has the most potential to provide additional flexibility in its implementation. By separating the *MouseHandler* and the *MouseHandlerFacade*, I allow the option of adding additional listeners to the *MouseHandler* for future iterations where, for example, I decide to display mouse coordinates in the bottom left corner.

Successes:

The area that I noticed the most success in throughout the project I felt was the mouse handler and its interactions with the mouse handler façade as I felt it effectively separated the concerns of the two classes while leaving the application relatively open for further implementations. In addition to this, I was able to handle the drawing of the shapes onto the canvas and storing of their data with ease, which I feel is greatly due to the extra ground work done by building a BoundingBox class in addition to a Point and PointToPointVector; by giving each IShape implementation a BoundingBox instead of 2 Points to represent its encompassing areas, I was able to easily include other helper methods such as BoundingBox.envelopes which compares two bounds for geometric comparisons. This came in handy chiefly with selecting shapes as it was only a simple call to the envelopes function, as well as with the drawing of shapes such as the Triangle, where I was able to calculate the midpoint of the bounds to get a quick data point that represented where the triangle should be drawn to.

Areas of Improvement:

The most notable area in need of improvement I noticed late in the design process was, ironically, the MouseHandler class. Once I had completed many of the core requirements of the project, I began to think of additional implementations that could be made later down the line, such as displaying a shape while the mouse is being dragged, and displaying cursor position data and so on. The current implementation uses the Builder pattern to send a BoundingBox to the listeners. However, I feel it could become much more modular and flexible by instead creating a ClickPhase enumerated class which would notify any listeners of the current click phase, leaving them to do what they will with that information. I feel this would allow much more flexibility of implementation by allowing the listening classes to each decide what to do with the data. One example would be a listener which takes the on mouse drag phase, and displays the coordinates on the screen, while another listener would still retain responsibility for executing draw shape commands, however with additional flexibility to also draw shapes while dragging.