Christopher Pole

SE450 – Object Oriented Software Development
Final Report

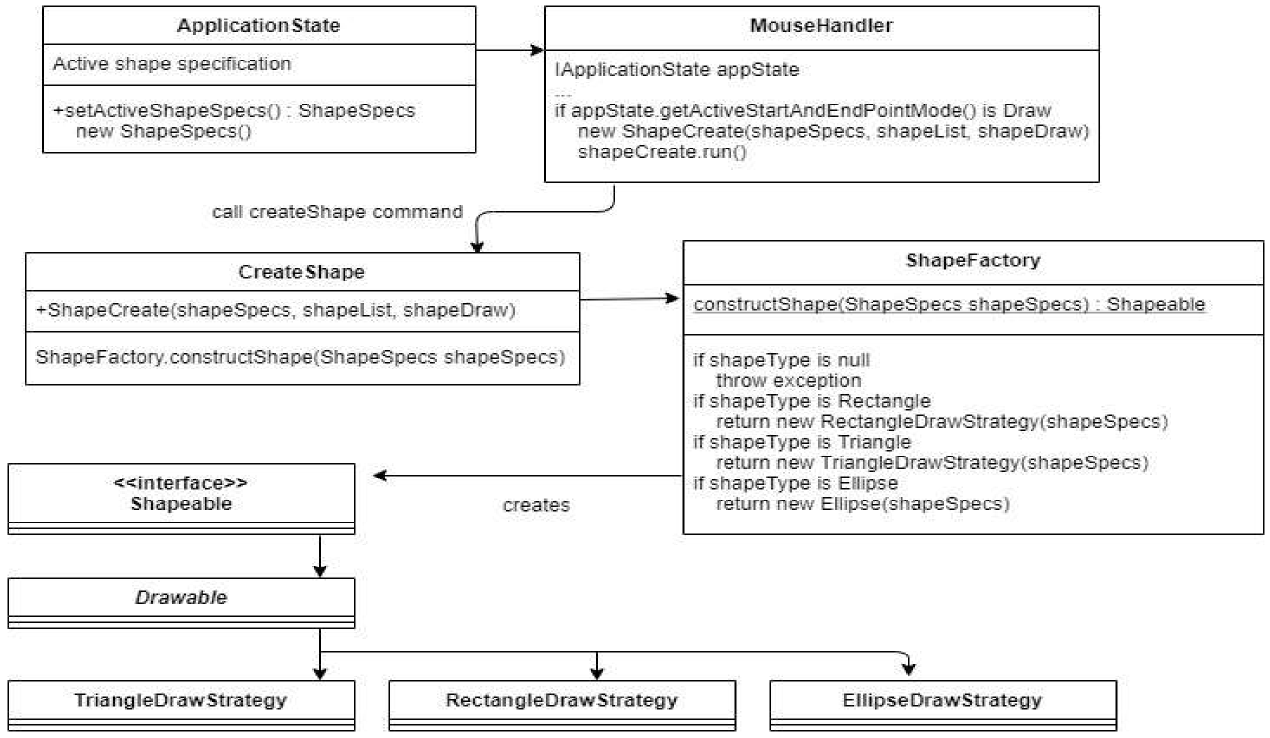
1. **Missing features, bugs, extra credit and miscellaneous notes**

- Full functionality is largely working as intended.
- Having issues with the order of Undo, Redoing of certain actions, particularly Group and Ungroup.
- Selecting shapes or groups of shapes will use the rectangular bounding box determined by the shape's or composite's maximum height/width and corner dimensions.
- There are six (6) unique design patterns: Factory Method, Composite, Builder, Command, and Template Method.

2. **Design Patterns**

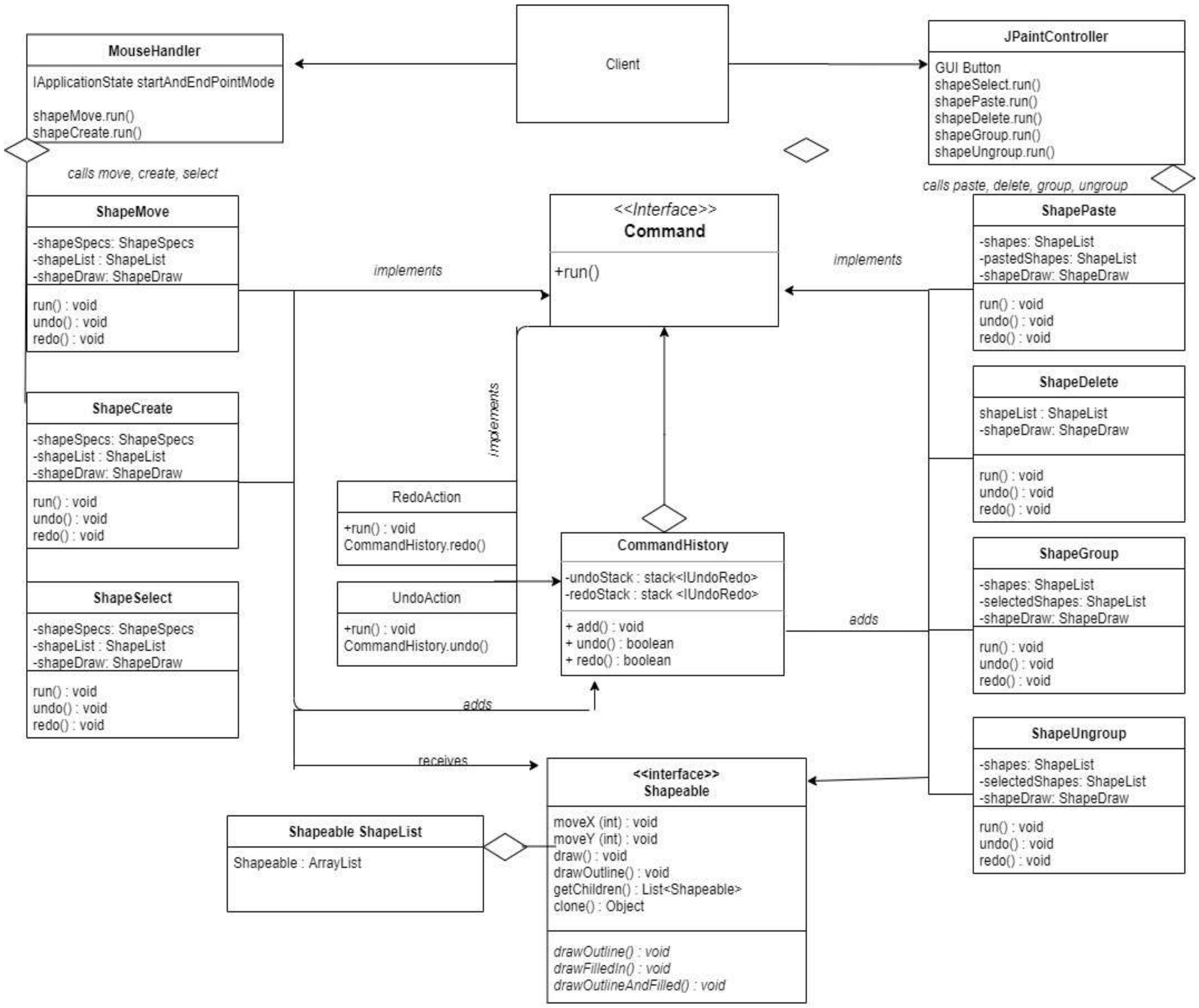
Factory Method Pattern

The factory method pattern was utilized in the ShapeFactory class as a means to delegate the responsibility of determining and constructing the necessary concrete shape. By doing this, the calling code (ShapeCreate) is decoupled from and isn't concerned with the instantiating (calling new) concrete shape implementations such that we are able to eventually add new shapes without modifying the ShapeCreate class.


ApplicationState
Active shape specification
+setActiveShapeSpecs() : ShapeSpecs
new ShapeSpecs()
MouseHandler
IApplicationState appState
...
if appState.getActiveStartAndEndPointMode() is Draw
new ShapeCreate(shapeSpecs, shapeList, shapeDraw)
shapeCreate.run()
call createShape command
CreateShape
+ShapeCreate(shapeSpecs, shapeList, shapeDraw)
ShapeFactory.constructShape(ShapeSpecs shapeSpecs)
ShapeFactory
constructShape(ShapeSpecs shapeSpecs) : Shapeable
if shapeType is null
throw exception
if shapeType is Rectangle
return new RectangleDrawStrategy(shapeSpecs)
if shapeType is Triangle
return new TriangleDrawStrategy(shapeSpecs)
if shapeType is Ellipse
return new Ellipse(shapeSpecs)
creates
<<interface>>
Shapeable
Drawable
TriangleDrawStrategy
RectangleDrawStrategy
EllipseDrawStrategy

Command Pattern

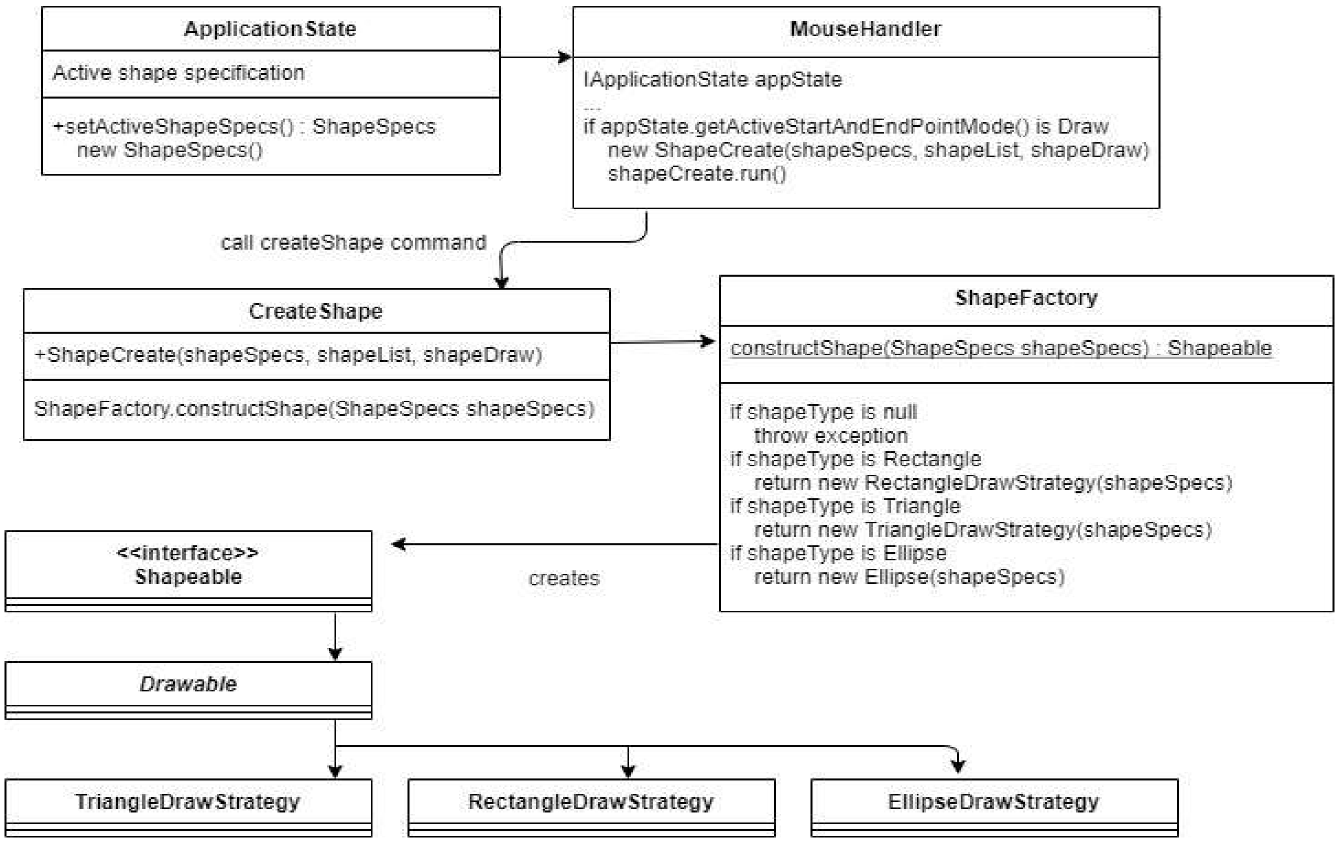
The Command Pattern is employed in each of the actionable classes in the controller package by implementing the ICommand interface (i.e. *run*). The invokers (MouseHandler and JpaintController) simply need to create and run the command object each time the user interacts with the canvas. This allows for concise calling code, confines the code performing the actual work to each command object, and, most importantly, these command objects can be stored in the CommandHistory stack. In this way, each command implementing the IUndo and IRedo interfaces can be undone or redone by popping from or adding to the CommandHistory stack.



Strategy Pattern

The strategy pattern, particularly useful for checking conditions and yielding appropriate behavior at run-time, is applied twice in my project.

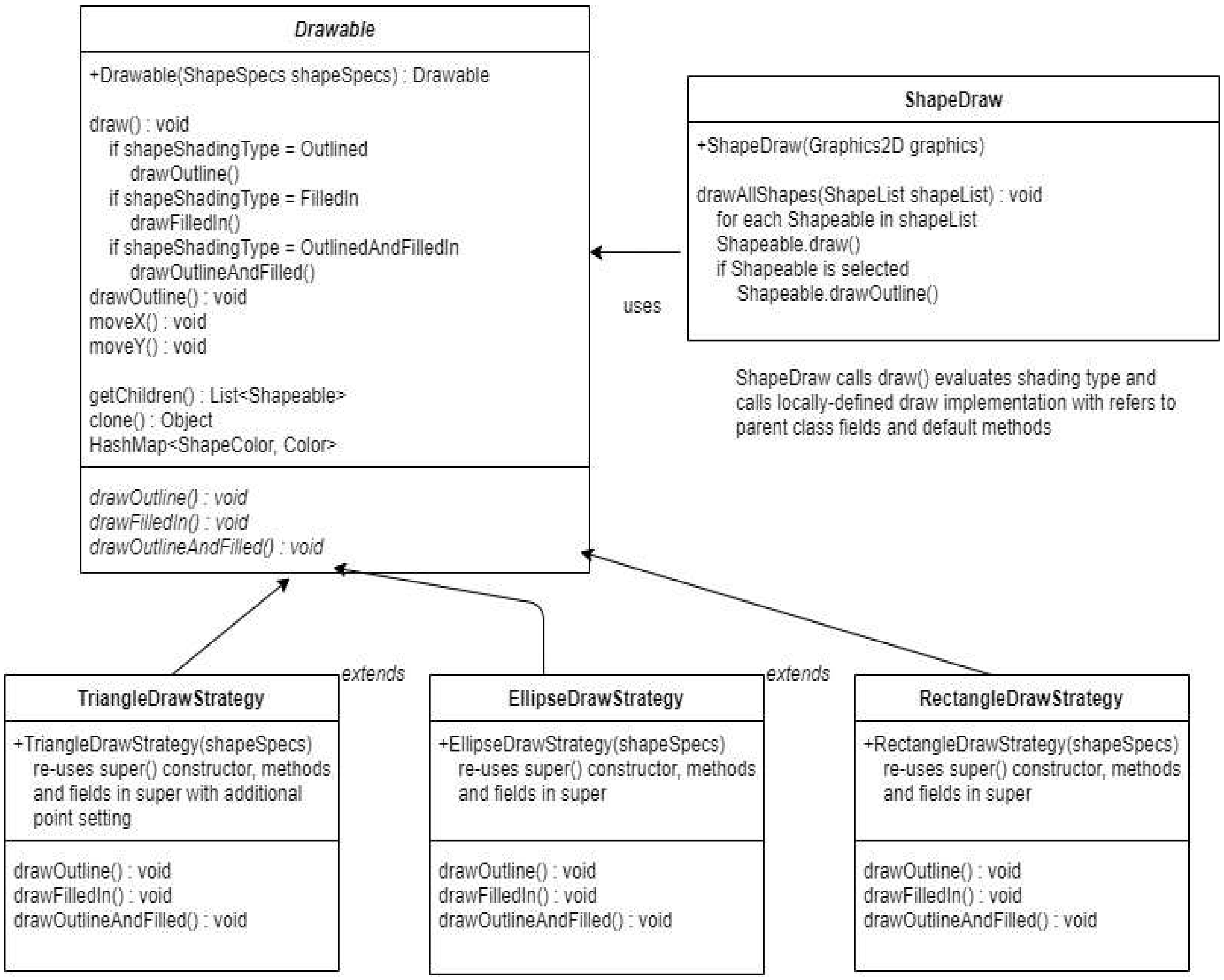
I use the strategy pattern in **creating** the shape objects. The ShapeFactory *constructShape* method which accepts the shape specifications, checks the **ShapeType** reflected in the specifications, and instantiates/returns an abstract Drawable object of the respective actual type. This logic is confined to the ShapeFactory *constructShape* method instead of the ShapeCreate command object to avoid tight coupling.


ApplicationState
Active shape specification
+setActiveShapeSpecs() : ShapeSpecs
new ShapeSpecs()
MouseHandler
IApplicationState appState
...
if appState.getActiveStartAndEndPointMode() is Draw
new ShapeCreate(shapeSpecs, shapeList, shapeDraw)
shapeCreate.run()
call createShape command
CreateShape
+ShapeCreate(shapeSpecs, shapeList, shapeDraw)
ShapeFactory.constructShape(ShapeSpecs shapeSpecs)
ShapeFactory
constructShape(ShapeSpecs shapeSpecs) : Shapeable
if shapeType is null
throw exception
if shapeType is Rectangle
return new RectangleDrawStrategy(shapeSpecs)
if shapeType is Triangle
return new TriangleDrawStrategy(shapeSpecs)
if shapeType is Ellipse
return new Ellipse(shapeSpecs)
creates
<<interface>>
Shapeable
Drawable
TriangleDrawStrategy
RectangleDrawStrategy
EllipseDrawStrategy

Same diagram as Factory Method Pattern above.

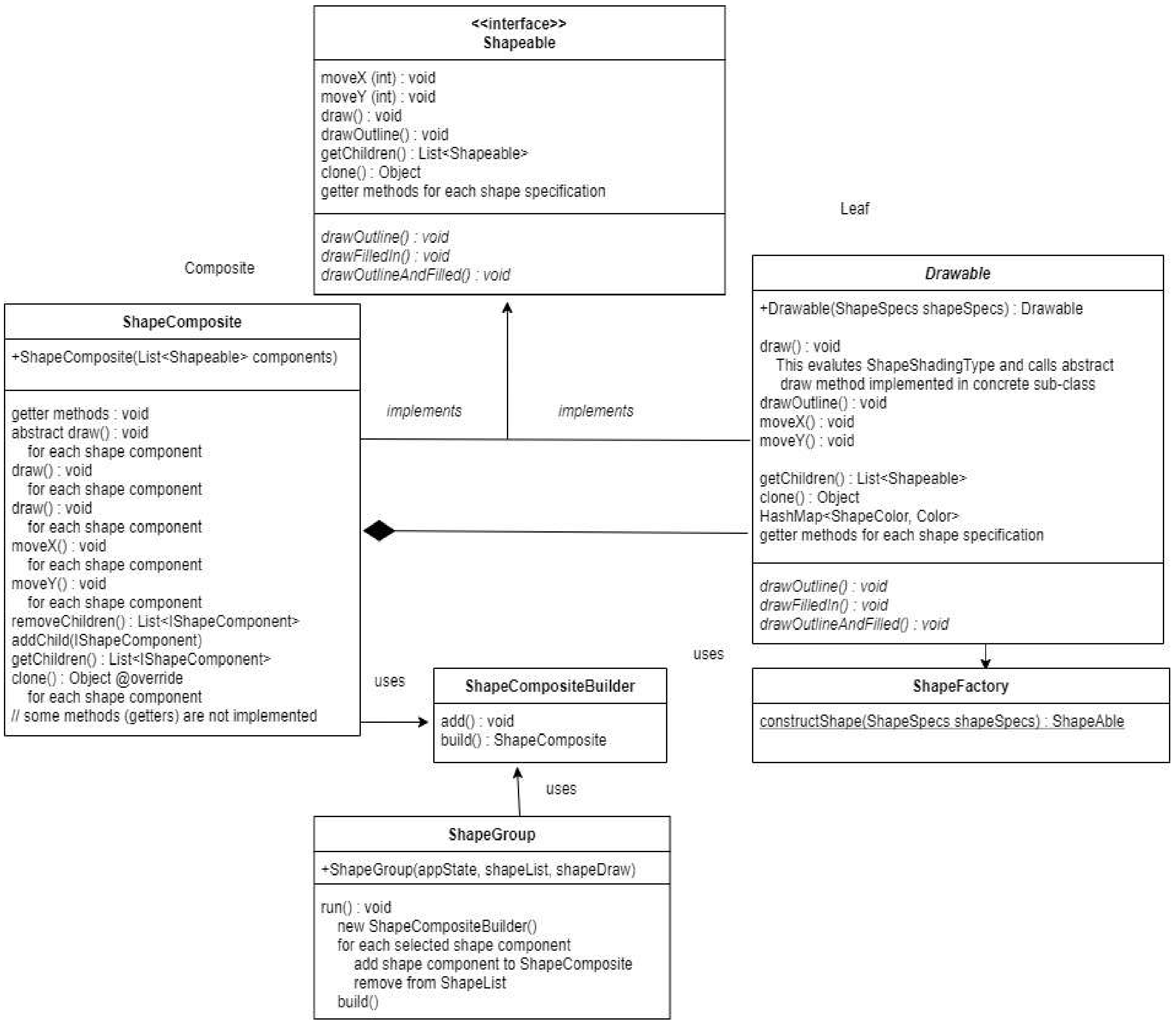
Template Method Pattern

The template method pattern is used to **draw** the shape objects by using inheritance. The Drawable *draw* method checks the local **ShapeShadingType** variable, and draws the object by calling the local draw implementation according to the ShapeShadingType and referencing local variables and default method implements within the parent class. By using dynamic dispatch, the *draw* method will call the necessary concrete draw implementation.


Drawable
+Drawable(ShapeSpecs shapeSpecs) : Drawable
draw() : void
if shapeShadingType = Outlined
drawOutline()
if shapeShadingType = FilledIn
drawFilledIn()
if shapeShadingType = OutlinedAndFilledIn
drawOutlineAndFilled()
drawOutline() : void
moveX() : void
moveY() : void
getChildren() : List<Shapeable>
clone() : Object
HashMap<ShapeColor, Color>
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void
uses
ShapeDraw
+ShapeDraw(Graphics2D graphics)
drawAllShapes(ShapeList shapeList) : void
for each Shapeable in shapeList
Shapeable.draw()
if Shapeable is selected
Shapeable.drawOutline()
ShapeDraw calls draw() evaluates shading type and calls locally-defined draw implementation with refers to parent class fields and default methods
extends
extends
TriangleDrawStrategy
+TriangleDrawStrategy(shapeSpecs)
re-uses super() constructor, methods and fields in super with additional point setting
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void
EllipseDrawStrategy
+EllipseDrawStrategy(shapeSpecs)
re-uses super() constructor, methods and fields in super
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void
RectangleDrawStrategy
+RectangleDrawStrategy(shapeSpecs)
re-uses super() constructor, methods and fields in super
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void

Composite Pattern

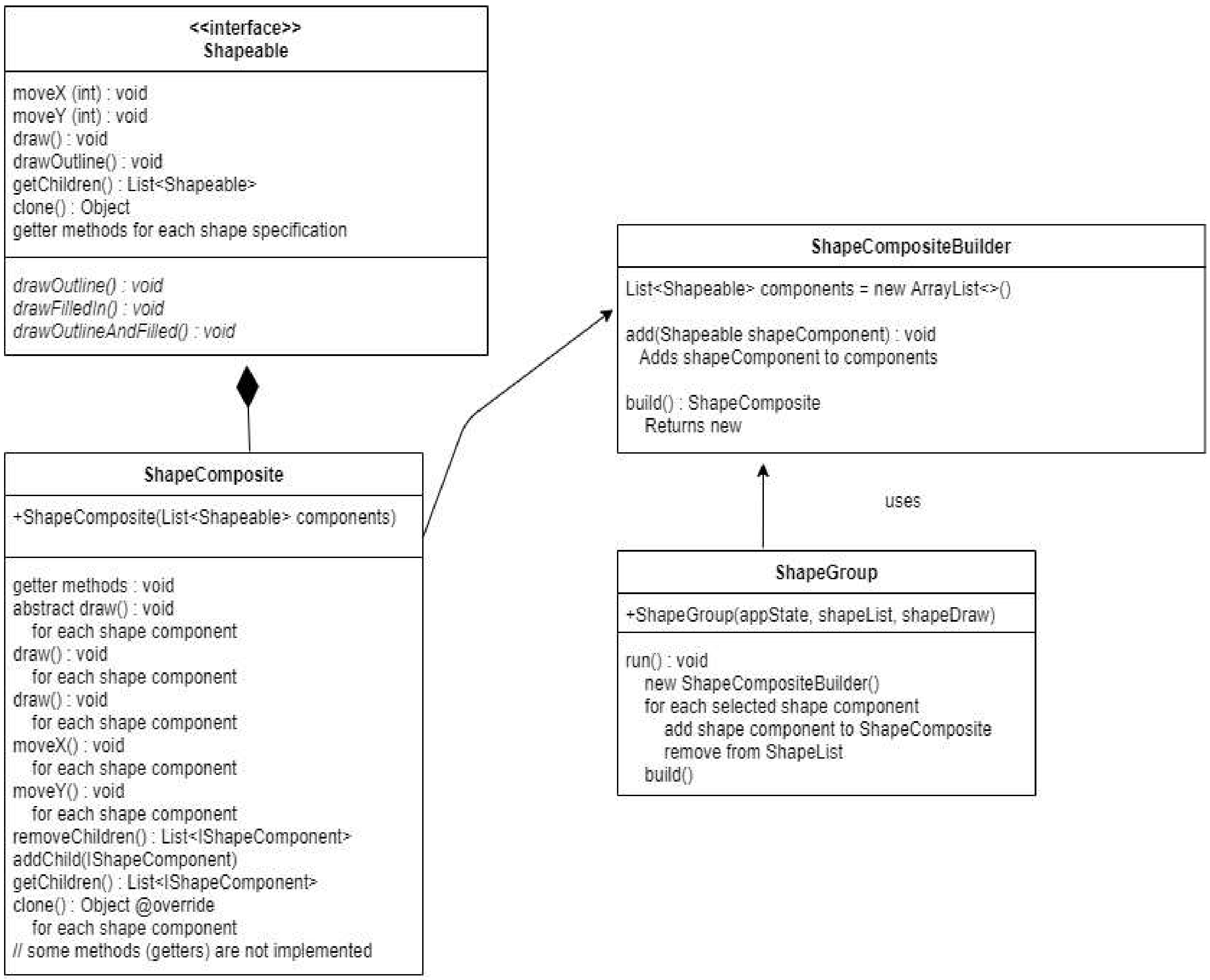
The composite pattern seen in the ShapeComposite class was indispensible in implementing the Group and Ungroup functionality. By implementing the Shapeable interface into my abstract Drawable class (pertaining to drawing and manipulating shapes) and the ShapeComposite class (pertaining to managing a group of Shapeable objects), this offers a tree structure wherein each node represents a group (or ShapeComposite) and each leaf represents a shape (or Drawable). The Shapeable *draw*, *drawOutline*, *moveX*, and *moveY* methods were rather easy to implement as I was simply iterating through the respective list such that the code will ultimately call the Drawable implementation. The *getHeight*, *getWidth*, *getModStart*, and *getModEnd* were also relatively trivial implementations of to assist with selection/collision detection.



<<interface>>
Shapeable
moveX (int) : void
moveY (int) : void
draw() : void
drawOutline() : void
getChildren() : List<Shapeable>
clone() : Object
getter methods for each shape specification
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void
Composite
Leaf
ShapeComposite
+ShapeComposite(List<Shapeable> components)
getter methods : void
abstract draw() : void
for each shape component
draw() : void
for each shape component
draw() : void
for each shape component
moveX() : void
for each shape component
moveY() : void
for each shape component
removeChildren() : List<IShapeComponent>
addChild(IShapeComponent)
getChildren() : List<IShapeComponent>
clone() : Object @override
for each shape component
// some methods (getters) are not implemented
implements
implements
Drawable
+Drawable(ShapeSpecs shapeSpecs) : Drawable
draw() : void
This evalutes ShapeShadingType and calls abstract
draw method implemented in concrete sub-class
drawOutline() : void
moveX() : void
moveY() : void
getChildren() : List<Shapeable>
clone() : Object
HashMap<ShapeColor, Color>
getter methods for each shape specification
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void
uses
uses
ShapeCompositeBuilder
add() : void
build() : ShapeComposite
ShapeFactory
constructShape(ShapeSpecs shapeSpecs) : ShapeAble
uses
ShapeGroup
+ShapeGroup(appState, shapeList, shapeDraw)
run() : void
new ShapeCompositeBuilder()
for each selected shape component
add shape component to ShapeComposite
remove from ShapeList
build()

Builder Pattern

The builder pattern is used by the ShapeGroup and ShapeUngroup classes to build (and de-build) shapes incrementally as shapes are added. It is the most suitable pattern to manage the ShapeComposite shape groups because it allows us to iterate through a selection and either group the selected elements, or disband the selected composite(s) and return children to the Master list.



<<interface>>
Shapeable
moveX (int) : void
moveY (int) : void
draw() : void
drawOutline() : void
getChildren() : List<Shapeable>
clone() : Object
getter methods for each shape specification
drawOutline() : void
drawFilledIn() : void
drawOutlineAndFilled() : void
ShapeComposite
+ShapeComposite(List<Shapeable> components)
getter methods : void
abstract draw() : void
for each shape component
draw() : void
for each shape component
draw() : void
for each shape component
moveX() : void
for each shape component
moveY() : void
for each shape component
removeChildren() : List<IShapeComponent>
addChild(IShapeComponent)
getChildren() : List<IShapeComponent>
clone() : Object @override
for each shape component
// some methods (getters) are not implemented
ShapeCompositeBuilder
List<Shapeable> components = new ArrayList<>()
add(Shapeable shapeComponent) : void
Adds shapeComponent to components
build() : ShapeComposite
Returns new
uses
ShapeGroup
+ShapeGroup(appState, shapeList, shapeDraw)
run() : void
new ShapeCompositeBuilder()
for each selected shape component
add shape component to ShapeComposite
remove from ShapeList
build()

3. **Successes and Failures**

- The MouseHandler turned out well and is easily managed. The primary responsibility is to log mouse clicks on the canvas by using the ApplicationState class to generate a set of shape specifications (or simply the drawn dimensions). Other common canvas functionality (erase, zoom, etc.) can easily be added in the future, but to limit responsibility it would be best to delegate the decisionmaking to a separate class after at a certain point.
- The Command classes are successful in maintaining a narrow set of responsibilities and properly delegating as needed. One major failure is how the command classes are all repainting the canvas with drawAllShapes if only for the reason of sloppy, duplicate code. This is simply fixed by repainting the canvas whenever a shape is added or removed. The particular ShapeList implementation was avoided intentionally in the beginning but I ran out of time to refactor the design.
- The ShapeList was relatively straightforward and successful. In hindsight, this also would've presented a good opportunity to implement the Singleton pattern given that we're operating on a single master list reference.
- Shape instantiation using the abstract Drawable class has proceeded pretty smoothly. The shape specifications are stored in the ShapeSpecs data class. The shape type decisionmaking is contained in ShapeFactory, the shading type decisionmaking is contained in Drawable, and the actual draw implementation is contained in the concrete shape draw strategy object. This designs means that we could relatively easily add new shape specifications, shape types, shape shading types, and so on
- The clone method (included in the Shapeable interface) turned out to be very effective once properly implemented. I had some trouble early on making proper deep copy clones, likely due to not properly implementing clone in the Point class.
- ShapeComposite has progressed smoothly together with the ShapeCompositeBuilder. Most of the methods simply recursively iterate through a list and execute the Drawable implementation, while some of the methods from the Shapeable interface are not necessary (the Shapeable interface could be cleaned up to avoid this).
- ShapeGroup and ShapeUngroup have presented a ton of issues, particularly when Undo and Redoing the commands. This is something I'm currently working on as it's not a really intuitive design as it stands
- The Shapeable interface could be refined (given that some methods aren't necessary) .
- Overall, I consider my project a success and like how my design has turned out. If there are two things I would change – it would be a redesign of the ShapeDraw (the command classes are looking very cluttered!) and cleaning up the ShapeGroup/ShapeUngroup run() functionality.