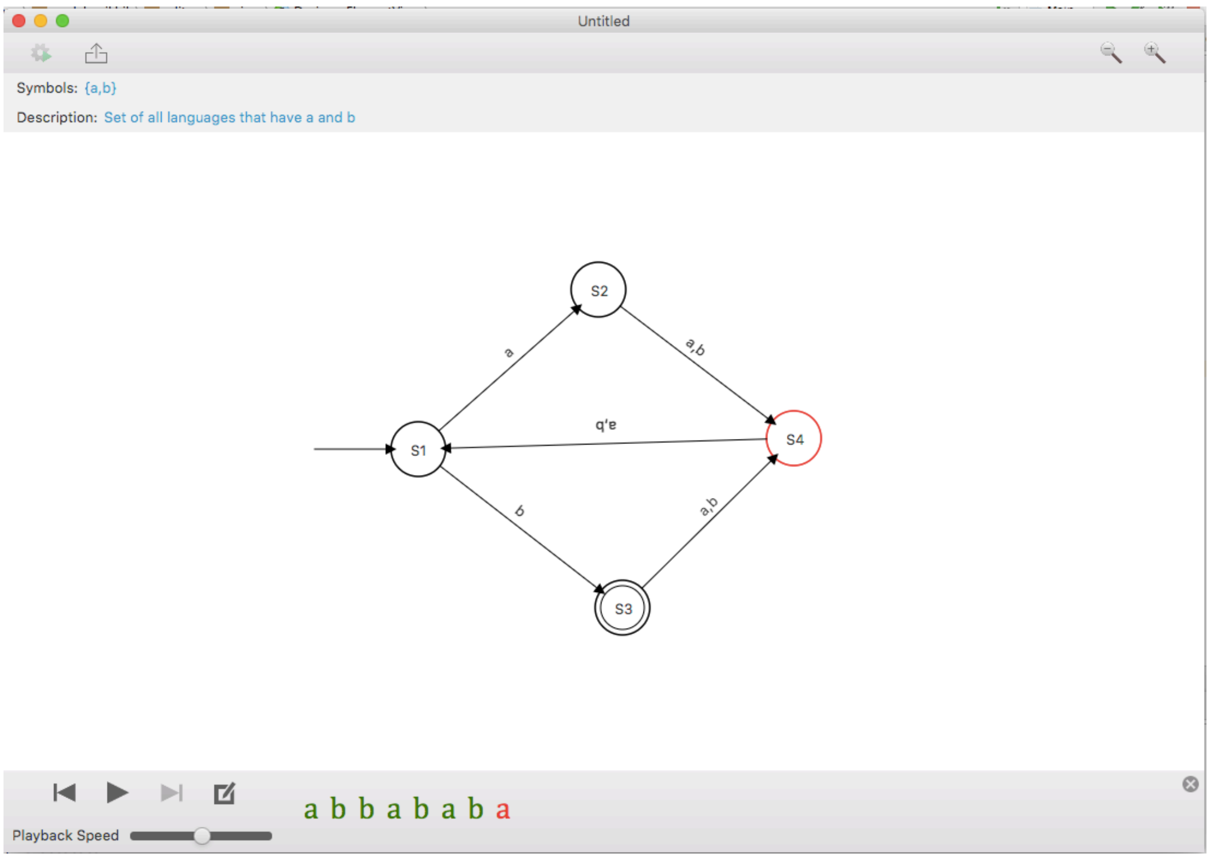
CSCI-641 Aspect Oriented Programming
Project Report
DFA Doer - A pedagogical tool in JavaFX
Nikhil Verma

Introduction

In this report, I will discuss the implementation of a pedagogical tool, emphasizing on some of the more pronounced uses of Aspect oriented programming in a GUI application.

State machines play a crucial role in the discussion of Computational Theory. As a visual representation of a mental model, they help in quickly clarifying the inner workings of a state based algorithm. "DFA Doer" is a GUI based pedagogical tool made in JavaFX. As an academic tool, it allows users to quickly design Discrete Finite State Machines and then test it on a given string. By testing a string, the system plays a transitioning animation and highlights the terminal state in red or green depending on whether it is final or not.

Functionality



The above screenshot shows what the application looks like. States are created by double clicking anywhere in the workspace. By holding shift and dragging towards another state, an empty transition is created. Double clicking a transition allows editing its symbols. If the states are moved (by dragging), their connected transition arrows reactively follow along. The panel at the bottom allows testing strings on the current DFA. It contains all the needful controls to step through each symbol of the input. While testing a DFA, editing is disabled.

States and transitions in the application can also be selected by either clicking on them or dragging a selection box around them. Selected states and transitions can then also be moved together or deleted.

Every operation in the application is undoable and redoable to multiple steps. The application also allows saving the document to an external JSON file from where it can be reloaded later. There are also hotkeys defined for several common tasks.

Implementation

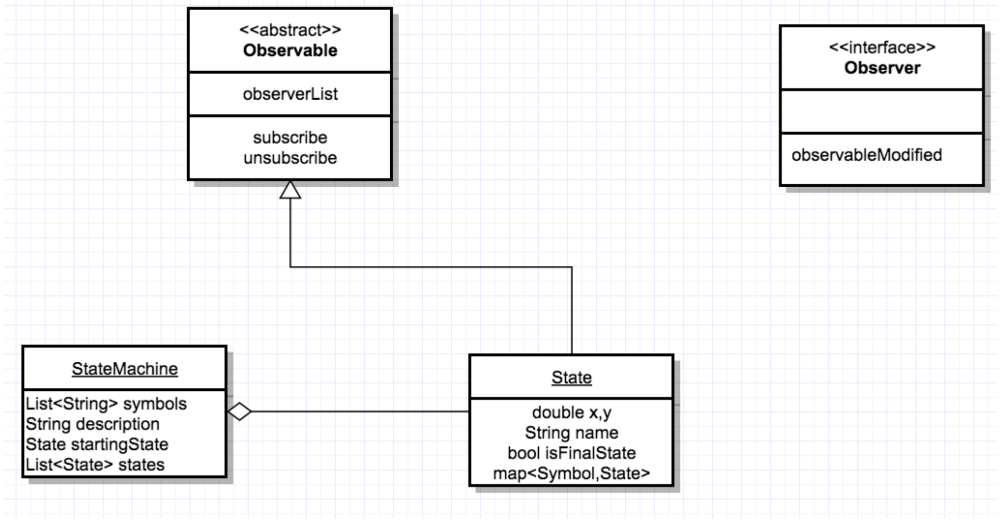
At its core, the application follows the standard Model View Controller approach. Several cross cutting concerns are handled in aspects which allow for a more modular codebase and better separation of concerns. Some of these include:

- Undo Redo
- Disabling mouse events during string testing
- Observer notification while moving states
- Performance tracking

Model

The core data classes in the application are **StateMachine** and **State**. These classes are strictly decoupled from the view and controller classes discussed ahead.

StateMachine
Holds all necessary details that define a DFA. These include symbols, description, starting state and list of states in the DFA.



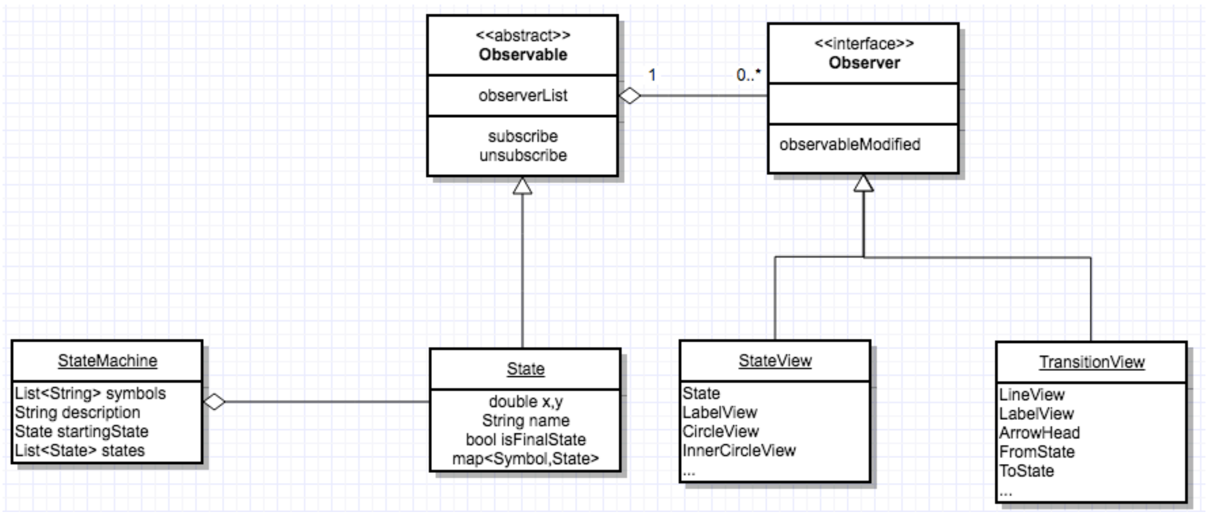
State
Defines a single state containing all the relevant details and a map that indicates outgoing transitions to other states. Keep in mind that the attributes (x,y) do not necessarily indicate the actual position in the view, but rather its position with respect to other states.

As can be seen from the diagram above, 'State' inherits an 'Observable' class. The use of observer pattern here becomes apparent when dealing with the reactive nature of its attributes with its view (discussed ahead).

View

Aside from all the usual UI elements provided by JavaFX, this application makes use of two custom View classes: **StateView** and **TransitionView**.

StateView
Represents a group of several graphical elements that make up a State in the workspace. It also holds the actual model 'State' being represented. By implementing the Observer Interface (as seen in diagram), 'StateView' receives notifications of any attribute change of its model 'State'

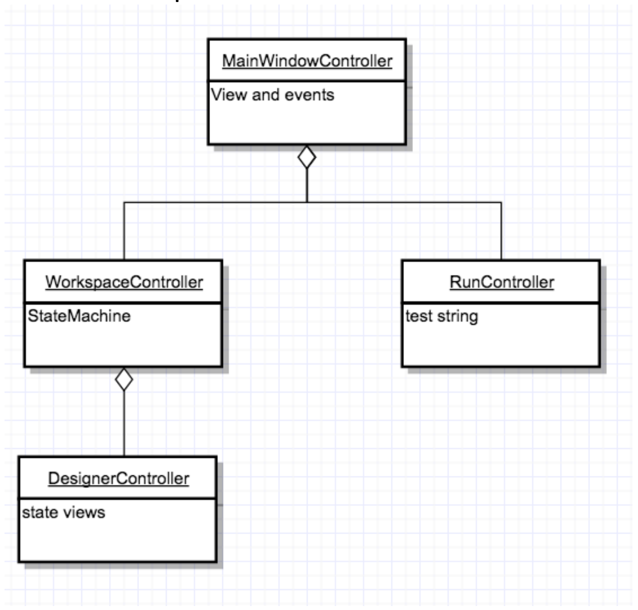


TransitionView
Represents a group of graphical elements that mark an outgoing edge from a state. They hold the 'from' and 'to' StateView that define the transition. As an observer, the transition view responds to any positional change to either of these model states.

Controller

This application makes use of an FXML file for its user interface. In JavaFX, an FXML file is a declarative file similar to HTML that allows setting up event handlers on elements. Event handlers are simply methods defined in a **MainWindowController** class.

The ‘MainWindowController’ then delegates the task to respective sub controllers that are responsible for various segregated functionalities in the application. The diagram below gives a basic overview of the concept.



WorkspaceController
Responsible for all high level features like saving and loading documents etc.

DesignerController
Handles all the necessary mouse and keyboard events that are used to create the DFA.

RunController
Responsible for handling all events in the test input phase

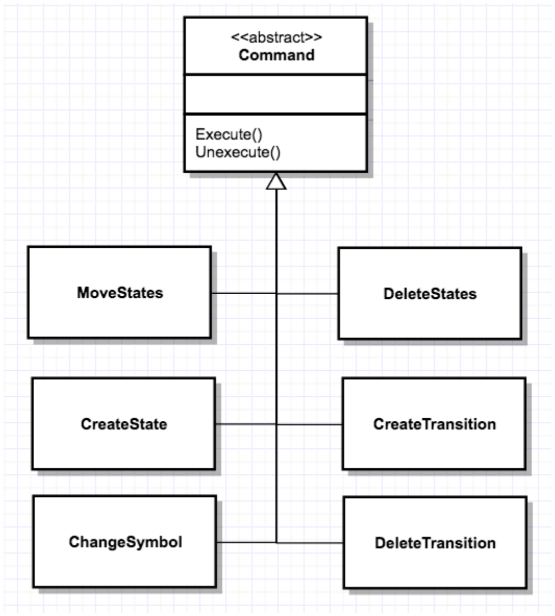
Cross Cutting Concerns

This section defines some of the cross cutting concerns in the application which were addressed using aspects.

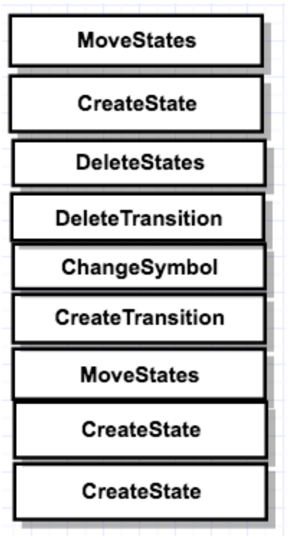
1.Undo Redo

As mentioned before, each operation in the application is undoable and redoable to multiple steps. Implementing such a features requires careful planning of every action performed in the application. Essentially, we want to update the state of the application and have a way to reverse that change when needed. This is where command pattern helps.

A command is defined by an abstract class that contains two abstract methods: *execute* and *unexecute*. As their name suggests, these methods are complementary in nature. Concrete subclasses of this abstract class are used to create objects that can bring the application to any desired state.

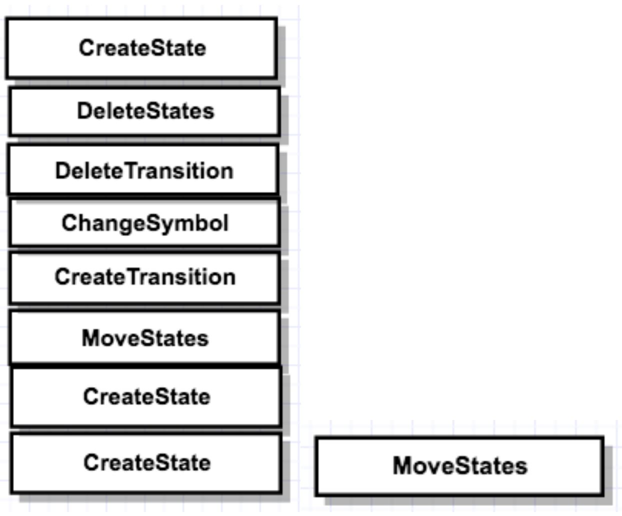


The diagram above depicts few of the many commands that are executed in the application. These command objects are created and used during necessary event calls. For example, '*MoveStates* is created and executed whenever state(s) are moved. After executing a command, it is put in a history stack which keeps track of all the commands executed up till this point.



History Stack

Undoing a command now is only a matter of popping out the top command from the history stack and "unexecuting" it. After doing so, the said command is then put on a **Future Stack** from where it can be redone later.

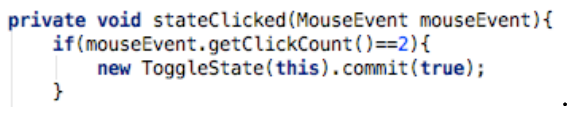


History Stack on left and Future Stack on right

Since such operations are scattered throughout the application's codebase, it is a cross cutting concern. It is imperative then that the process of pushing commands to the History stack be handled in a separate Aspect.

<u>UndoRedo.aj</u>

The future stack and history stack are maintained in this aspect.



```
private void stateClicked(MouseEvent mouseEvent){
    if(mouseEvent.getClickCount()==2){
        new ToggleState(this).commit(true);
    }
```

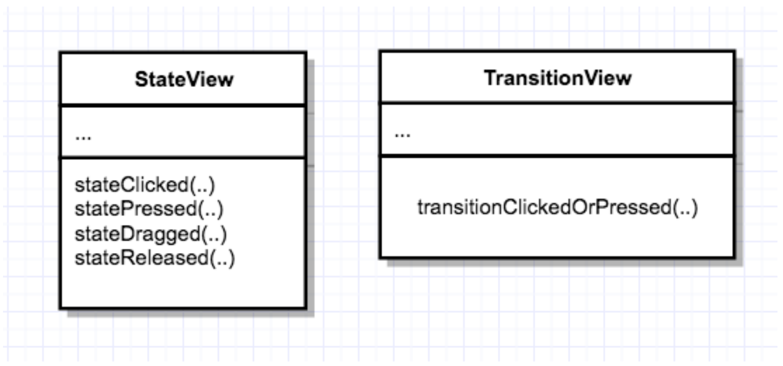
Double clicking a state toggles its final flag

It defines a pointcut advice that is executed whenever a command is committed. It also defines pointcuts for the undo and Redo event handlers (from the MainWindowController) which are performed here. The code below is a reduced version for the same.

```
public aspect UndoRedo {

    private Stack<Command> history=new Stack<>();
    private Stack<Command> future=new Stack<>();

    pointcut operationCommitted(Command command):(call(public void
Command+.commit(..))&&target(command));

    after(Command command):operationCommitted(command){
        //…push the command onto the history stack and reset future from
here
    }

    //defines a pointcut for the undo event callback from GUI
    pointcut undo(MainWindowController controller):(execution(private void
com.madebynikhil.editor.controller.MainWindowController.undo(..))&&
target(controller));

    after(MainWindowController controller):undo(controller){
        //…Performing undo event
    }

    //defines a pointcut for the redo event callback from GUI
    pointcut redo(MainWindowController controller):(execution(private void 
com.madebynikhil.editor.controller.MainWindowController.redo(..))&& 
target(controller));

    after(MainWindowController controller):redo(controller){
        //…Performing redo event
    }
}
```

2.Disabling mouse events on custom views while testing a string

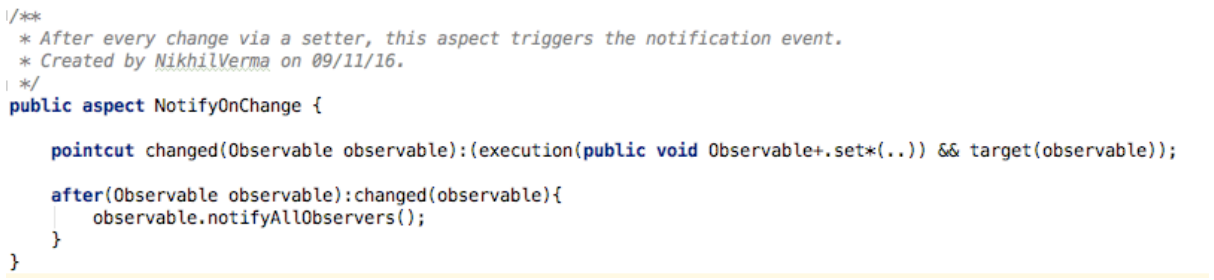
To prevent editing of the DFA during input testing, mouse events that are defined on StateView and TransitionView are disabled using an around advice. These views define handlers that are intercepted and bypassed by an aspect



DisableMouseEventsWhileTesting.aj
Contains an around advice that bypasses the event handlers during testing phase.

3.Moving transitions connected to a state (Observer Pattern)

As discussed earlier, to allow transition views to react to changes in their connected states position, observer pattern was employed. An aspect is used to send notifications to all the subscribers of an observer after any setter method is called. This aspect uses a wildcard syntax to define the pointcut.

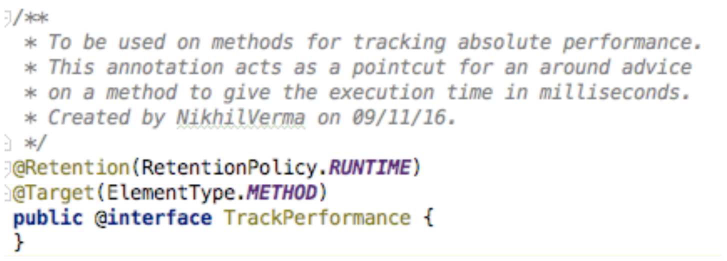


```
/**
 * After every change via a setter, this aspect triggers the notification event.
 * Created by NikhilVerma on 09/11/16.
 */
public aspect NotifyOnChange {

    pointcut changed(Observable observable):(execution(public void Observable+.set*(..)) && target(observable));

    after(Observable observable):changed(observable){
        observable.notifyAllObservers();
    }
}
```

4.Performance Tracking

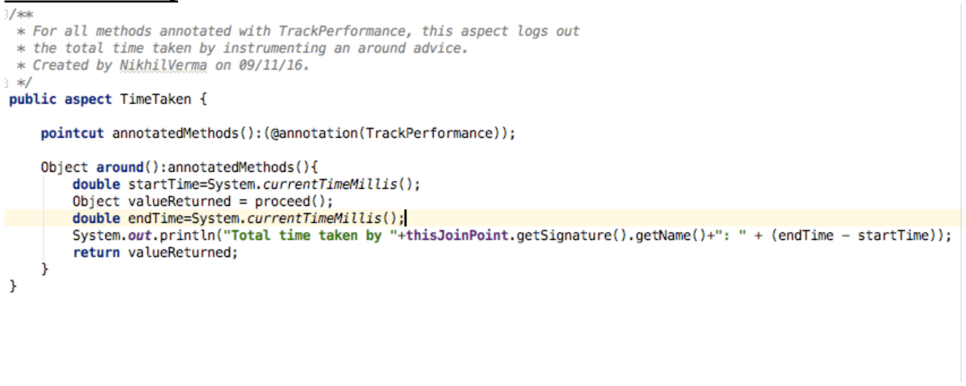
Tracking performance of heavy tasks such as an IO operation is performed with the help of an around advice that outputs the total time taken by recording the start and end time of an annotated method. For this, a custom annotation called TrackPerformance was created and used as a pointcut in an aspect named TimeTaken.

TrackPerformance.java



```
/**
 * To be used on methods for tracking absolute performance.
 * This annotation acts as a pointcut for an around advice
 * on a method to give the execution time in milliseconds.
 * Created by NikhilVerma on 09/11/16.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface TrackPerformance {
}
```

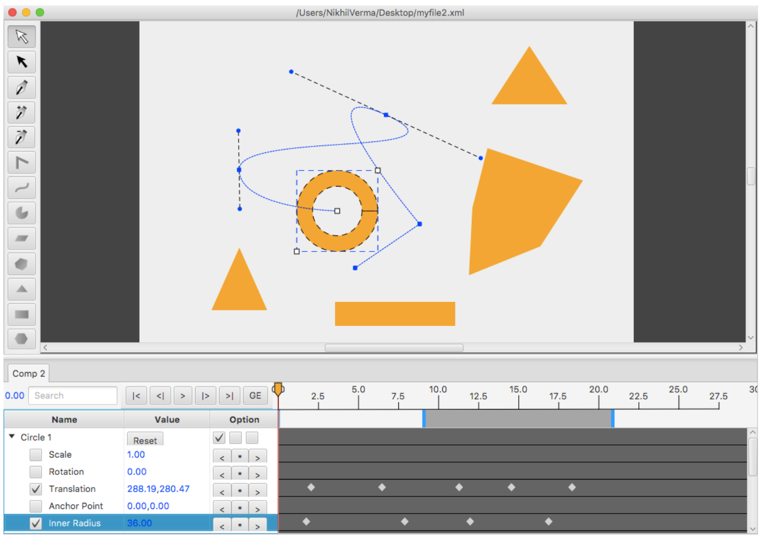
TimeTaken.aj



```
/**
 * For all methods annotated with TrackPerformance, this aspect logs out
 * the total time taken by instrumenting an around advice.
 * Created by NikhilVerma on 09/11/16.
 */
public aspect TimeTaken {

    pointcut annotatedMethods():(@annotation(TrackPerformance));

    Object around():annotatedMethods(){
        double startTime=System.currentTimeMillis();
        Object valueReturned = proceed();
        double endTime=System.currentTimeMillis();
        System.out.println("Total time taken by "+thisJoinPoint.getSignature().getName()+": " + (endTime - startTime));
        return valueReturned;
    }
}
```

Comparison with a non AOP project: Timeshift

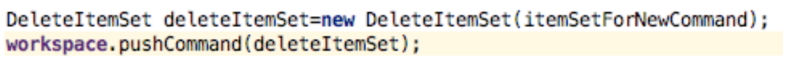
By making clever use of aspects, we were able to decouple some of the most intricate features of the application. To solidify this claim, I will briefly compare the development of another similar GUI project that I did in the past, which did not make use of aspect oriented programming.



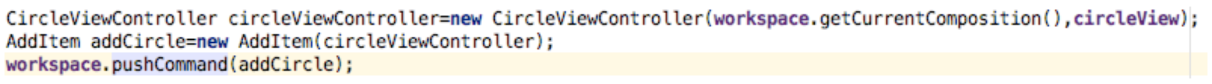
Name	Value	Option
Circle 1	Reset	✓
Scale	1.00	< * >
Rotation	0.00	< * >
✓ Translation	288.19,280.47	< * >
Anchor Point	0.00,0.00	< * >
✓ Inner Radius	36.00	< * >

Timeshift is an animation tool that allows the animation of various types of shapes. Shapes like rectangles, polygons, circles, triangles etc. contain several properties that can be tweaked to change their appearance. By ‘key framing’ these properties across a timeline, these shapes can be animated. This was a big project and I am purposely skipping many sophisticated features of the tool in the interest of reader's time. But the main part that demands attention here is the implementation of the Undo Redo feature.

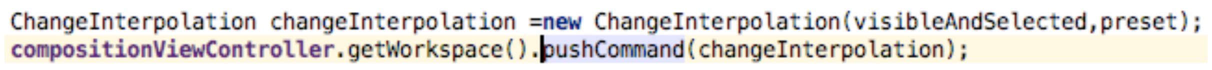
Timeshift also made use of the command pattern as described earlier. But it did not rely on aspects to push commands onto the history stack. Instead, every action performed in the application required a manual push onto the history stack. To make matters worse, the history stack was part of a special "Workspace" class (similar to WorkspaceController here). Therefore, in the controller hierarchy, every class had to hold 'back references' to their parent to facilitate getting access to the workspace class. Here are a few snippets from the application.



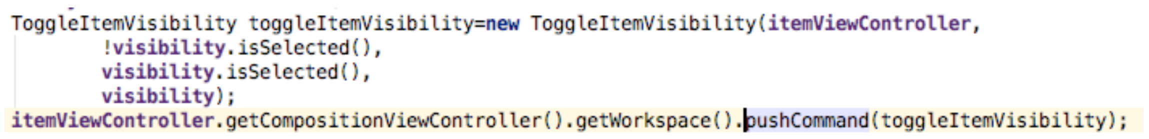
```
DeleteItemSet deleteItemSet=new DeleteItemSet(itemSetForNewCommand);
workspace.pushCommand(deleteItemSet);
```



```
CircleViewController circleViewController=new CircleViewController(workspace.getCurrentComposition(),circleView);
AddItem addCircle=new AddItem(circleViewController);
workspace.pushCommand(addCircle);
```

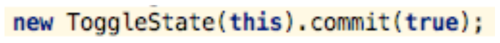


```
ChangeInterpolation changeInterpolation =new ChangeInterpolation(visibleAndSelected,preset);
compositionViewController.getWorkspace().pushCommand(changeInterpolation);
```

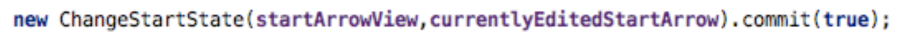


```
ToggleItemVisibility toggleItemVisibility=new ToggleItemVisibility(itemViewController,
        !visibility.isSelected(),
        visibility.isSelected(),
        visibility);
itemViewController.getCompositionViewController().getWorkspace().pushCommand(toggleItemVisibility);
```

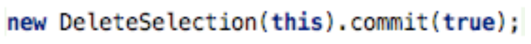
The current approach in this project does not require doing any such extra work. In fact, it resolves to just one liners



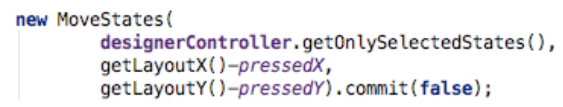
```
new ToggleState(this).commit(true);
```



```
new ChangeStartState(startArrowView,currentlyEditedStartArrow).commit(true);
```



```
new DeleteSelection(this).commit(true);
```



```
new MoveStates(
        designerController.getOnlySelectedStates(),
        getLayoutX()-pressedX,
        getLayoutY()-pressedY).commit(false);
```

Conclusion

Aspect oriented programming adds a nice layer in the development of elaborate software solutions. While AOP still only solves a niche category of problems, it still helps in establishing a more modular codebase.