Changes to the Model

1. IReadableShapeAnimation
   1. This interface was added to allow the model to return a list of readable animatable shapes. This is necessary so that the view can read this information.
2. IReadableTransformation
   1. This interface was added so that the view can read information from transformations.
3. Builder
   1. The builder was added to the model to allow the file reader to work.
4. checkTransformationOverlap()
   1. This method in the model was added and is called whenever a transformation is added. It is to ensure that two transformations of the same type don’t occur at the same time. Helper methods were implemented in the transformation classes and interfaces to allow this functionality.
5. Type changes
   1. Width/height and Points were changed to floats.
6. maxTime()
   1. Returns how long the animation is. This is necessary for the timer to work so it knows how long to run for.
7. getAllShapes()
   1. This method was added so the view can receive all the information about what shapes there are. This was added because there was no mechanism for the view to get this information before.
8. getAllTransformations()
   1. This method was added so the view can receive all the information about what transformations there are. This was added because there was no mechanism for the view to get this information before.
9. getDescription() removed
   1. getDescription is no longer the job of the model, but rather the view. Therefore it was removed/moved to the view.
10. Changed ShapeType to an enum
    1. This was changed from a string. It was changed to an enum because it is better design so all the possibilities are enumerated and there is more structure when using type in a switch statement.

DESIGN OVERVIEW

The model keeps track of all the shapes and what transformations act on those shapes. Shapes are distinguished by their IDs. The controller/view can read from the model in two ways: it can get the states of the shapes at a given time, or get a list of all the shapes in the animation in order of creation and a list of all transformations in order of start time. The former is designed for a view that “plays” the animation, while the latter is for something that will format all the information.

Once again, shapes are wrapped into a ShapeAnimation, which essentially is an animatable shape that keeps track of when the shape appears and its name.

The top level controller is the interface IController. It is what all controllers should inherit from, and allows you to set and get a FPS (frames per second). The most important method is run(). The concrete controller will implement run in an appropriate way for the concrete view.

TextController is the controller for text views. This controller is used for both the Description and SVG views. This controller implements run by passing the list of shapes and transformations to the view.

GraphicalController is the controller for visual views. The controller is used for the VisualView. This controller is meant for views that “play” through the animation, displaying it frame by frame. This controller implements run by using a timer and passing the shapes at each frame from the model to the view to be drawn at each frame.

IView is the top level view interface. It implements three methods. The first is drawShapes. This method is used to draw a state of the animation, and should be implemented only by visual views. The next two is outputText and setOutput. These are meant for views that output text.

AbstractTextView is the abstract view that implements IView that text views inherit from. Its job is to initialize the output for the view and implement shared behaviors like formatting time. SVG and DescriptionView extend this class. These views both use the visitor pattern to create output.

The two texts views are given a list of readable shapes and readable transformations, but the issue is that they can not easily resolve their types. Because of this, visitors were implemented.

There are two visitor interfaces, one for shapes and one for transformations. In every readable shape and readable transformation, there is an accept method for their respective visitors. The visitors are used by the view to output concrete-specific details. For example DescriptionShapeVisitor is able to visit shapes to create concrete shape-specific descriptions, and DescriptionTransformationVistitor does the same for transformations.

SVG uses SVGFormatter, which is essentially a wrapper for another shape visitor. The reason for this is that the visitor for shapes for SVG must also have access to its transformations, so we must ensure that the visitor is also passed the correct list of transformations and is not used incorrectly outside the formatter. Additionally, because transformations are also dependent on the concrete shape type, two transformation visitors had to be made; one for each shape.

The visitor pattern is the best choice for two reasons. The first is it’s the most elegant way to resolve type. Without it, we would need to worry above safe-casting. As the transformations themselves also have fields whose types need to be resolved and the SVG view relies on the concrete type of both shape and transformation at the same time, this quickly becomes very cumbersome. Second, it is more open to extensibility. In order to make a new view, you simply need to make a new visitor for shapes and transformations that formats the information into the format you would like. Once the visitors are written, writing the view is trivial.

The final view is the visual view. This view’s heavy lifting is done by the model and controller. All the visible view has to do is draw whatever shapes it is given exactly as they are.