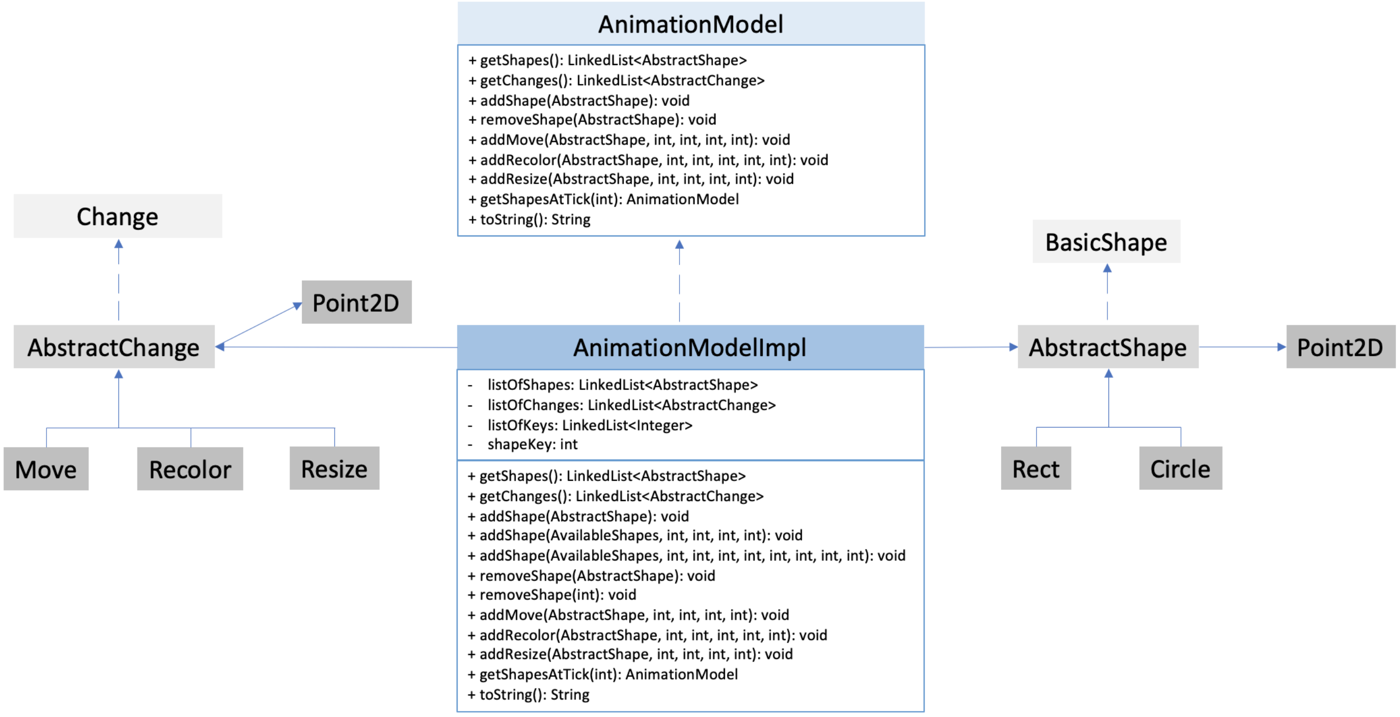
EasyAnimator

This is a simple animator software made for CS5004

Overview

The model operates with a ledger of shapes and changes to those shapes. Respective classes of shapes and changes manage the behaviors of the animation. Rect and Circle are concrete shapes extending the AbstractShape class that implements the BasicShape interface for shapes. Move, Resize, and Recolor are all concrete changes that can occur to shapes as the animation progresses. The model's main job is to store and update the shapes according to the changes based on ticks and inputs managed upstream by the controller.

Main Model

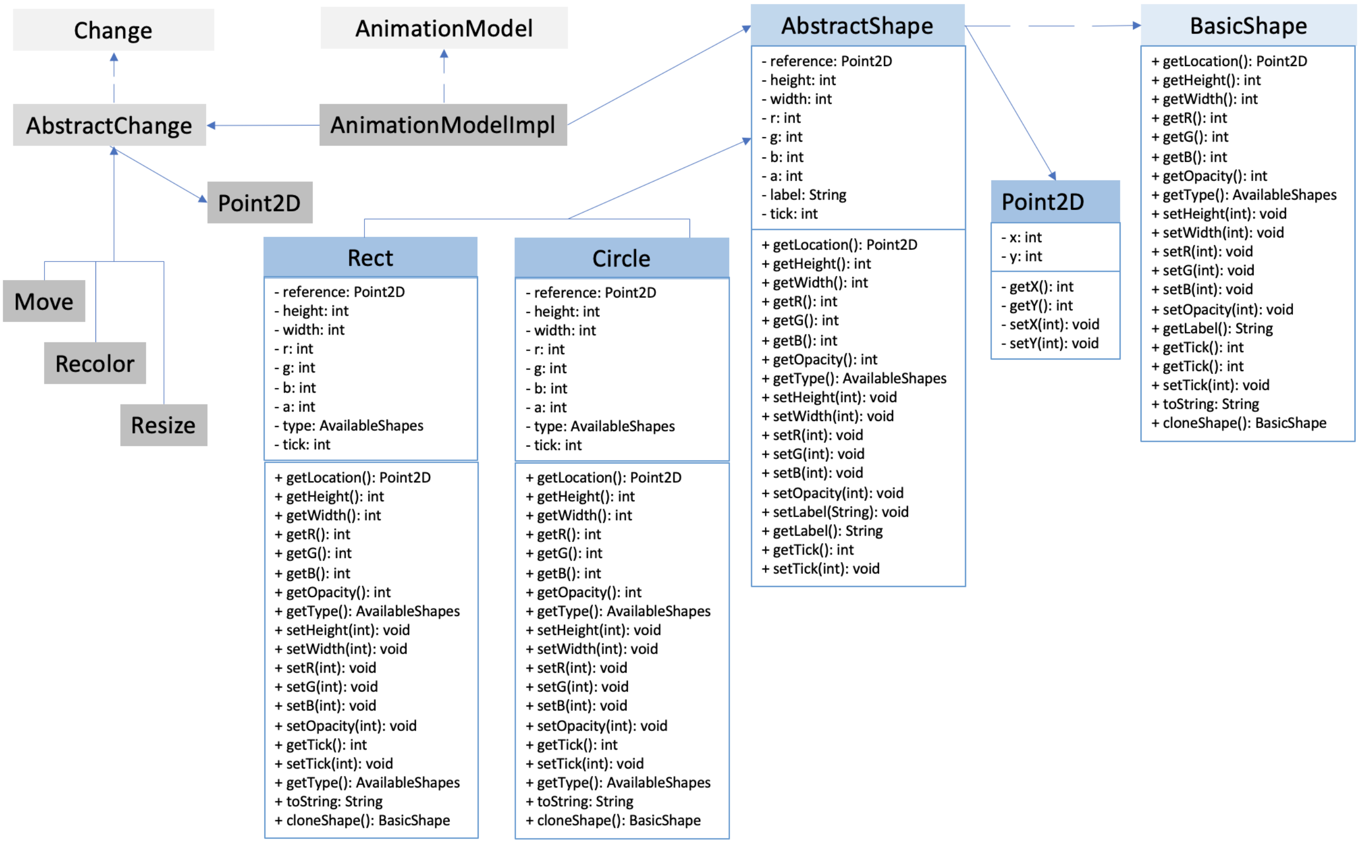

AnimationModel
+ getShapes(): LinkedList<AbstractShape>
+ getChanges(): LinkedList<AbstractChange>
+ addShape(AbstractShape): void
+ removeShape(AbstractShape): void
+ addMove(AbstractShape, int, int, int, int): void
+ addRecolor(AbstractShape, int, int, int, int, int): void
+ addResize(AbstractShape, int, int, int, int): void
+ getShapesAtTick(int): AnimationModel
+ toString(): String
Change
BasicShape
Point2D
AbstractChange
AnimationModelImpl
AbstractShape
Point2D
- listOfShapes: LinkedList<AbstractShape>
- listOfChanges: LinkedList<AbstractChange>
- listOfKeys: LinkedList<Integer>
- shapeKey: int
+ getShapes(): LinkedList<AbstractShape>
+ getChanges(): LinkedList<AbstractChange>
+ addShape(AbstractShape): void
+ addShape(AvailableShapes, int, int, int, int): void
+ addShape(AvailableShapes, int, int, int, int, int, int, int): void
+ removeShape(AbstractShape): void
+ removeShape(int): void
+ addMove(AbstractShape, int, int, int, int): void
+ addRecolor(AbstractShape, int, int, int, int, int): void
+ addResize(AbstractShape, int, int, int, int): void
+ getShapesAtTick(int): AnimationModel
+ toString(): String
Move
Recolor
Resize
Rect
Circle

AnimationModel: interface

AnimationModelImpl: implementing concrete class

The AnimationModel class is the interface for the AnimationModelImpl that sits at the core of the model. The implemenation holds listOfShapes for created shapes in a base state, listOfChanges holds the changes that will take place over the course of the animation, and the keys are held in a list and updated incrementally as a simple way to access shapes if needed that way just within the model. When the animation runs, modified copies of the of shapes are returned via getShapesAtTick() where changes in listOfChanges dictate alterations to versions of shapes held in listOfShapes. Shapes and changes can be added and removed here via outlined methods.

Shapes in the Model



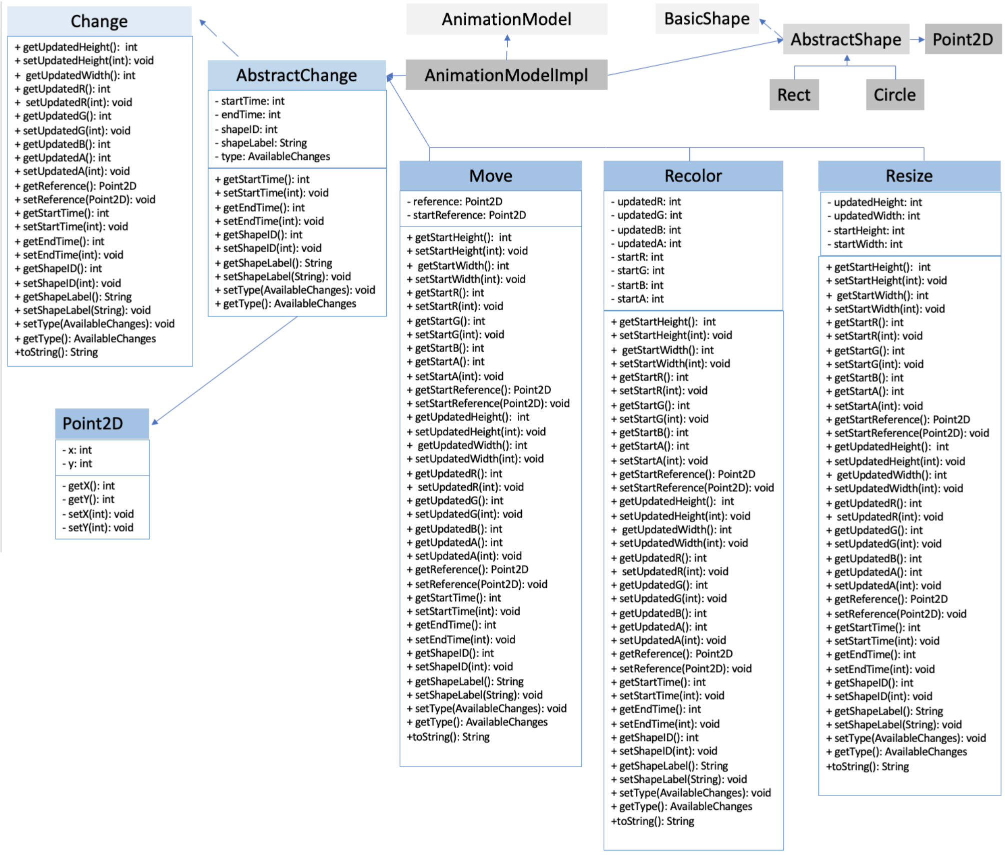
BasicShape: interface

AbstractShape: abstract class with a Point2D

Rect, Circle: concrete inheriting classes

Shapes themselves all have positions (Point2D composition), dimensions, color values, a string label to help provide another way to ID shapes, and a tick for when the animation runs (used by the controller as time passes). These attributes can be set and get via common methods in AbstractShape. Concrete shapes also have overloaded constructors so there are multiple ways to create shapes, have individual toString() methods, and can retrieve their respective enum types. Because the model will have to create copies of shapes, cloneShape() will help to get copies of the base case shapes in the model.

Changes to the Model



Change: interface

AbstractChange: abstract class with a Point2D

Move, Recolor, Resize: concrete inheriting classes

The model also keeps track of changes which each get starting attributes, ending attributes, start time, end time, a type, and a label and ID's for the shape that the change they are associated with (both are there for the sake of flexibility for the controller). While individual concrete classes, like Move, change only a particular type of attribute, Point2D in the case of Move, AbstractChange handles common attributes, like timestamps, ID, label, a type (which gets set in the concrete classes via a "super" call). These changes keep record of what change is supposed to occur, and over what time period, helping the model to generate appropriate copies of existing shapes at the appropriate times.

note: enum classes are not shown in the UML diagrams (types of changes and shapes are enumerated)

View

The scheme for the view has one interface that designates the behaviors for three implementing concrete classes: SVGView, TextView, and VisualView. The SVGView and TextView both generate a text file, with the SVGView being formatted specifically to meet SVG requirements so that it can render the animation. TextView simply generates a readable description of the animation. The VisualView generates a JFrame with a "canvas" panel component on which Graphics2D can be painted and repainted at particular ticks as the animation runs. The VisualView takes the model (like each other view) and uniquely converts "tweened" shapes at the current tick to graphics. Each view has an enum type and can run (IView specifies other behaviors but these are not functionally universal in their implementation).

In the main EasyAnimator class that takes the command line arguments for the animation, the arguments are parsed and then used to build a model and other simple parameters needed for the views. These parameters are passed to the ViewMaker that can generate the appropriate view type given these inputs. If there is an error in the execution of main() in EasyAnimator, the program will generate a warning panel.

Changes to the Model

From the original implementation, the model had some minor updates. Most notably, the data structures for storing changes and shapes were changed from lists to maps, reducing run time which was particularly important for visual view when running larger animations. As a result, the model can now also quickly getShape() and check if it hasShape() for searching. Moreover, a model now has a custom canvas class that determines the size/location of the canvas for view. Of course, the builder class was also implemented in the model to build a model given an input text file. Some superfluous methods and fields were originally added to leave flexibility for the controller/views, but these were removed. The extra parts of the model, now removed, were an overloaded addShape() method, and the ShapeID field and its associated getter/setter.