Review of Providers' Code

Design critique

Overall, the interfaces were very flexible and easy to understand and reuse. After our first read-through of the code, we were easily able to understand how the views worked and how our model could adapt to it with simple steps. The design of the functionalities implemented was clear and flexible. The only issue was functionalities that were missing, which were necessary to make the program function at all. Our providers' view design relied on clear, essential functions of the model: providing shapes and their lists of keyframes. Providing this type of information to the views was the only necessary requirement to integrate our implementations. We found no calls to model or controller interface methods that seemed too specific to their implementations, so we were impressed by their interface design.

Implementation critique

Aside from the missing functionalities in our providers' code, our only critique on implementation would be in the way they worked with lists of keyframes. This was also one of our biggest difficulties when creating our adapters. The views worked with lists of keyframes by requesting them from the model and mutating (sorting) them in a view class, then requesting them again from the model (in the view) in order to get keyframes at a proper tick. I believe that this type of implementation should be handled in the model (e.g. using a getShape(int tick) method) to avoid mutating the model outside of itself or the controller. Even if this method isn't added to the model, I'd suggest retrieving a copy of the keyframes, sorting that copy and using it where it is necessary. This way, we wouldn't be mutating the model from the view and relying on that mutation in order for the view to function.

Documentation Critique

We had no huge problems with the code documentation of our providers. Some javadoc could have been more specific, but the purpose of most methods were pretty clear from their implementations. However, some vague javadoc caused a lack of clarity. For instance, the getInputKf() method at line 53 of NewVisualView.java was a bit unclear. Without documentation of how the keyframe fields are taken from the user (and without access to the KeyFrame constructor), we didn't know which numbers in the array of integer inputs referred to which values of a keyframe. Luckily, we figured it wouldn't matter much, since we couldn't find anywhere that displayed to the user in what order to input the values. It would also have been nice to be provided documentation regarding what functionalities of the code were used and/or functional. We also had to fix many styling issues for Java style points. This included moving class variables that were only used once.

Design/Code Limitations

Some implementations of the code proved limiting to us as consumers. One major confusion involved how a NewVisualView and IController must work together. On one hand, the view took the IController as an argument in its constructor. On the other hand, it is necessary that the controller has access to the view as well, but the IController interface (which we were provided) did not have a method to take in a view. This would imply that the view is passed into the controller's constructor, but that causes a circular problem where neither object can be initialized. We worked around this by adding a setController() method to our IController adapter interface, which would initialize the view and pass itself to its constructor. Other issues regarding

this implementation caused limitations as well. Since their controller was passed to the view as an argument, there was no equivalent to a setListener(IController c) method in the view, which would pass the controller to the panel and set it as an action listener. Furthermore, the setListener method in the panel was never used, and so we had no access to the panel to invoke it. We worked around this problem with another adapter class for the view, but our implementation felt a bit hacky. We therefore would suggest providing a method in the interface that allows public access to setting the listener of the inner animation panel. This would also solve the circular issue in the constructors so that the view can instead take in a controller via this new method. Overall, the design of the code was clear, thoughtful, flexible, and easy to adapt to. Our main request would be to add the necessary functionalities of the view.

Review of our Experience

We learned a lot about the design and purpose of adapters from this experience. We especially learned about what cases are necessary for different types of adapters. For instance, most classes within our model (shapes, keyframes, etc.) used a delegating adapter pattern, since our main function already creates instances of our original classes, and their methods are easily translated. However, our model uses inheritance because it needs to implement both model classes (since our original model interface needs to be passed to our controller). This way, we wouldn't have to overwrite all of our methods by simply calling the corresponding methods on the delegate model. We realized that inheritance in the adapter pattern is more convenient when instances of the class will be passed as both the original and the adaptee interfaces.

Another main takeaway from this experience was about communication. Although we didn't need to make updates for our consumers and they didn't request any code or information in addition to the code we initially provided, we realize we could have done more to communicate in order to make their experience easier. Our code had possible implementation leaks that could have been difficult to adapt to.

We also could have communicated better with our providers. Throughout our process, we made requests for more code (e.g. their controller interface), but we could have attempted to communicate more about their code's functionality. We spent a lot of time trying to get their view to work with our code and waited too long to ask if it was actually functioning. We eventually found out that their view does not function. Their panels are never painted in the view, so our new hybrid view shows a blank window. Our providers were unable to make the necessary changes to their code. However, they were quick to respond to our requests. Overall, collaborating with a provider was a beneficial experience that gave us insight into the unique thought process of students facing the same task.