In terms of the AnimatorCommand interface and the following implementations:

Pro side:
Code structure is easy to follow. Thus, is easy to be modified in response to future changes (if any).
If you someday want to add a new implementation of AniCommand, you will just have to add the new get methods into the AniCommand interface; add the new get methods into the AbstractAniCommand class; and write a class representing the new command which will override the new get methods. You won't have to modify the existing command classes. Which is awesome.

Area of improvement:
Speaking from the practical angle. If a ColorAniCommand wants to call something like getStartPos() or getStartWidth(), should it return a null or should it throw an exception? I believe it is the later one. Since 'null' represents lack of information. Whereas throwing an exception represents a wrong operation. And seeking position or scale information from a color command is regarded more as a wrong operation than a lack of information.
For going there, you may go into your AbstractAniCommand, and rather than write 'return null' as the default for those get methods, please consider writing 'throw IAE….'' As the default for those get methods.

In terms of AniShape and its implementation:

Pro side:
Even though you only have one class implementing the AniShape interface, I believe that it's still a good practice to create an interface and a child class rather than a single class. Since it opens the room for future modification, if it eventually turns out to have new feature to be added.

Area of improvement:
For the child class AnimatedShape. On one side, the documentation said that the getEndTime() method should return -1 if the shape doesn't have an end time. Yet on the other side, the implementation of getEndTime() is simply 'return etime.' And in the constructor of AnimatedShape, it is not supported that the 'etime' may be -1. Since stime must >= 0, and it's not allowed for etime to be smaller than stime, according to your code. Thus, how will the documented functionality be implemented? Your code won't return -1 if the shape doesn't end.

I've taken a look at your getShapeAtTime(int time) method. Its idea is to apply all the animations on the startingShape up until the specified time. This method works well.

I'm not sure if my recommendation works with your code. But here is a thing to notice:

If I firstly announce getShapeAtTime(1), and then announce getShapeAtTime(1.1), you'll have to do the same thing twice, and do a little bit more thing for the second time due to the extra .1 second. This might not be efficient. What's efficient is:
At the first time I call getShapeAtTime(1), you calculate the things from 0 to 1.
At the second time I call getShapeAtTime(1.1), you only need to refer to the things done on the above line and add on the extra calculation for the things from 1 to 1.1.

There are ways for achieving this. You may consider using an interval tree. In which while given a time, you go into the tree to find the corresponding shape information with the given time. If you wanna go this way, firstly you will have to store into the interval tree with shape stages at each time interval. It's a little big of the workload, but in this way you only need to do the calculations once rather than doing it at every time you call getShapeAtTime.

In terms of AnimatorModelState interface; AnimatorModel interface; and SimpleModel class:

Pro side: You have AnimatorModel interface extending the AnimatorModelState interface, which offers more flexibility. The logic is this: if someday in the future you have to add another model class and implementation which is different from SimpleModel, you may simply have another interface extending the AnimatorModelState interface which incorporates different methods inside than those in the AnimatorModel interface, and have the new implementation following that new interface.

Area of improvement:

Consider to move getPossibleShapesAndDefinitions() from AnimatorModelState into AnimatorModel. Since there might be in the future that you need another implementation of modeling which does not have to use this feature called getPossibleShapesAndDefinitions(). As a result, getPossibleShapesAndDefinitions() is more specific to SimpleModel implementation, thus should be moved into the interface AnimatorModel.

The method getShapes(int time) returns List<Shape> corresponding to the given time.
The method getShapesInAnimation() returns List<String> representing all shapes in the animation. Yet please consider to rename this method for better representing its purpose. For instance, you may rename it to getShapesAsString().
Otherwise, users might be confused of why does getShapes(int time) return List<Shape> while getShapesInAnimation() returns List<String>.

Other implementations in this area are fine. No redundant methods detected. Later on let's see how you use those methods to support the view.

In terms of your Shape class:
I see that you set all the field as private. Yet, later on in that class, you wrote public get methods which return each field as an output. So are those fields private or public? Now, people can see those fields you denounced as 'private' using the public get methods you wrote. Which defeats the encapsulation purpose. I assume that most of those public get methods are for testing purposes. Thus, you may consider to rewrite your test to test the same thing in different way such that you won't need to know the fields of Shape class for the testing.

In terms of your TextualAnimatorView; SimpleAnimatorView; AbstractTextAnimatorView; SVG AnimatorView; and TextAnimatorView:
Firstly, I noticed that you have your SimpleAnimatorView extending TextualAnimatorView. Even though this implementation does not cause any problem, there is a slight area of improvement. SimpleAnimatorView interface represents all the views: SVG; text; visual; and interactive. Thus, I believe that it should be on the top, and have other interfaces extending it. However, now you're putting TextualAnimatorView on the top, which only represents SVG and text view.

If I was you, I would consider to put SimpleAnimatorView on the top, and having TextualAnimatorView extending it. Rather than the other way around. Following that, I will consider to let AbstractTextAnimatorView implement the TextualAnimatorView interface.

Your SVGAnimatorView and TextAnimatorView are fine. The AbstractTextAnimatorView class is doing a decent job in defining the common methods from SVGAnimatorView and TextAnimatorView.

In terms of your AnimatorViewCreator:
The create method inside takes in three arguments, with one being the 'tps.' However, only text and SVG view will need the tps. Thus, users of your program will be confused of what to put inside the method while they want a visual or interactive view. Such confusion is not wanted and should be dealt with. To be more specific, currently your create method is always asking for a 'tps.' Thus, some users will definitely be worried about missing something while they want to create a visual or interactive view. They would assume that 'tps' is needed for visual or interactive view, which isn't being the case.

To deal with this issue, I suggest you to create two 'create' methods. With one taking in three arguments, with one being 'tps.' The other only takes in 'type' and 'model.'

Thus, users will not have to worry about entering a ‘tps’ if they only want a visual or interactive view.

In terms of AnimatorController; AnimatorControllerFeatures; and InteractiveController:

Please consider to let AnimatorControllerFeature extends AnimatorController. In this way, your InteractiveController will only need to implement AnimatorControllerFeature interface, which makes more sense. Since it is rare to have a class implementing two interfaces that you created.
At the same time. AnimatorControllerFeature contains methods which are specific to the interactive part for this assignment. Thus it makes more sense to have it extends AnimatorController, and have InteractiveController implementing it.