Assignment 6: The Easy Animator: Part 2: Let there be Motion!

Due: Tues 03/20 at 8:59pm; self-evaluation due Wed 03/21 at 9:59pm

`Starter files: code.zip`

This assignment is to be completed with a partner. Please make sure both partners request each other as teammates for this assignment on the handin server. We will then manually approve of your team. You will not be able to submit your assignment if we do not know your group!

In this assignment will implement several *views* for your model from the last assignment. A view is responsible for rendering (some or all of) the data in a model in a form that is understandable to whoever is actually trying to use the data.

1 Working in pairs

This assignment is designed for groups with more than one person. Although the expected work is not simply double of what it was in earlier assignments, the assignment behooves mutual cooperation and often, pair programming.

Please make a plan, meet frequently and tackle the assignment in a systematic manner. Dividing the work and then meeting directly the day before it is due is recipe for disaster. This assignment requires you to self-learn some things, so enforce each other's efforts and learning.

2 (Ani)Mating the models

Before you and your partner can begin working on views, you must first come to consensus on the design of your model.

1. Unify the designs of your models. Incorporate the best ideas from both, or any methods that one of you may not have thought of.

2. **Refactor your model so that it resides in the `cs3500.animator.model` package.** Your tests should continue to be in the default package. If this change requires correcting visibility in your code so far, please make the necessary changes.

3. Document any changes made to your models from the previous assignment in a README file: explain what was added, removed, or changed (besides the package declaration), and why.

As with the model, in the abstract sense, a view is an interface; particular concrete views implement that interface. Accordingly, your work in this assignment should carefully distinguish between any one particular implementation of a view and the common interface to which they should all adhere. Moreover, if your model from Assignment 5 failed to have a similar interface/implementation split, you must fix that.

3 Views

In this assignment you will work on **three** views for your Easy Animator application.

3.1 View interface(s)

Start by planning your views, and observing which operations you need. Although different views look and behave differently, there are some common aspects to all views. The design of the actual interface(s) is left up to you. A common design technique is to have a view interface that has all functionalities and then individual views suppress or provide defaults for functionalities they do not implement (e.g. Fundies 2 uses this technique to motivate double dispatch: you can review it here.). Another relevant design rule is from the SOLID principles from Lecture 1: Why object-oriented design?: Interface Segregation (No client should be forced to depend on methods that it does not use). Think about these aspects as you come up with a design for your views.

3.2 Speed of animation

In order to produce an animation, your views may need to convert the animation from unitless "ticks" to actual time durations. This can be controlled by specifying a tempo for the animation, in ticks per second. Thus the same animation can be produced at different speeds, by altering the tempo.

3.3 A textual view: seeing is believing

This view will show a textual description of the animation. This description should be the description you produced in Assignment 5, with two changes: a rectangle has a "Min-corner" instead of a "Lower-left", and the times in ticks should be converted into actual times in seconds. To facilitate testing, this view should work with a variety of output sources.

For example the output of the text view for the textual output of the small demo animation from Assignment 5, at a tempo of 2 ticks per second, should be as follows:

```
Shapes:
Name: R
Type: rectangle
Min-corner: (200.0,200.0), Width: 50.0, Height: 100.0, Color: (1.0,0.0,0.0)
Appears at t=0.5s
Disappears at t=50.0s

Name: C
Type: oval
Center: (500.0,100.0), X radius: 60.0, Y radius: 30.0, Color: (0.0,0.0,1.0)
Appears at t=3.0s
Disappears at t=50.0s

Shape R moves from (200.0,200.0) to (300.0,300.0) from t=5.0s to t=25.0s
Shape C moves from (500.0,100.0) to (500.0,400.0) from t=10.0s to t=35.0s
Shape C changes color from (0.0,0.0,1.0) to (0.0,1.0,0.0) from t=25.0s to t=40.0s
Shape R scales from Width: 50.0, Height: 100.0 to Width: 25.0, Height: 100.0 from t=25.5s to t=35.0s
Shape R moves from (300.0,300.0) to (200.0,200.0) from t=35.0s to t=50.0s
```

You may achieve this functionality by moving your code to generate this rendering from the model to the appropriate view.

3.4 Visual Animation view

In this view, you will draw and play the animation inside a window, effectively reproducing the sample animations shown in Assignment 5.

3.4.1 Producing shapes within an animation: Tweening

Each kind of animation can be thought of as changing some attribute of the shape with time. For example, moving a shape changes its position while keeping color and size unchanged, while changing the color does not change its position and size.

Each such kind of animation is specified by how the shape is before it starts, and how it is after it ends. For example, consider moving a rectangle R from its position (min-corner) A at time $t = t_a$ to another position B at time $t = t_b$. The "initial state" of R at time $t = t_a$ is when it is at point A and the "final state" of R at time $t = t_b$ is when it is at point B. The animation should show R moving from A to B smoothly. This requires us to compute the intermediate state of R at any time $t_a \leq t \leq t_b$. This is called "in-betweening" or "tweening" (yes, an invented word!).

We can accomplish tweening in a simple manner using linear interpolation. Consider a variable f whose value is 'a' at time t=0 and 'b' at time t=1. Then its value at any time $0 \leq t \leq 1$ is given by $f(t) = a(1 - t) + bt$ (verify its values at $t = 0$, $t = 1$ and $t = 0.5$). Generalizing to any time interval (t_a, t_b):

$$f(t) = a\left(\frac{t_b - t}{t_b - t_a}\right) + b\left(\frac{t - t_a}{t_b - t_a}\right)$$

for any $t_a \leq t \leq t_b$

We can apply this formula to any attribute of a shape. For example, we can apply it to the x and y coordinates to interpolate position, or the red, green and blue components (r,g,b) to interpolate colors.

(Mathematically, linear interpolation can be visualized as drawing a line segment from a to b, and then thinking about how the values change uniformly.)

3.4.2 Implementation Details

To implement this view, you will need to use Java Swing. (You are *not* permitted to use the `javalib` library we used in Fundies 2, as it conflates the notions of model, view and controller into a single `World` class. Additionally, Java's other GUI library, JavaFX, is overly complicated for our purposes.) The code provided with the MVC code and the Turtles activity from class give you a basic beginning using Swing. You will likely need to look up documentation on `Graphics` class, `Panel` and `Frame` functionalities.

Normally a GUI draws itself whenever it is asked to, or when certain events occur (e.g. maximizing or resizing the window). In order to produce an animation, you need some mechanism (similar to an alarm clock) to tell the window to keep refreshing itself automatically.

3.4.3 Behavior of the visual view

The animation should start as the view is loaded, with no additional inputs from the user. The window may or may not be big enough to show the entire animation: you should use scroll bars to ensure that the user can see different parts.

If objects overlap during the animation, they should be drawn in the order in which they were created (specified as input).

3.5 SVG Animation view

One of our initial motivations for this application was to make it easy for a user to create and play an animation. Conceptually it should be possible to create a visual representation of our animation using existing animation frameworks, such as Flash, etc.

In this view, you will produce a textual description of the animation that is compliant with the popular SVG file format. The SVG file format is an XML-based format that can be used to describe images and animations. It is an example of "vector-based graphics" where it stores explicitly the shapes to be drawn and manipulations to be done on them, instead of pixel values. Most browsers support SVG rendering[1]. Therefore, the output of your program, if saved in a file, can be directly viewed in a web browser. Once again, to facilitate testing, this view should be able to work with a variety of output destinations to which it transmits the SVG description.

Some simple examples of SVG files have been provided to you, with embedded comments (please open the files in a text editor to see its source code). You should read the official SVG documentation to learn more about this format. You will find the descriptions on shapes and animations particularly relevant. The Mozilla Developer Network also provides some useful documentation on SVG, particularly a handy element reference.

Again, remember that since SVG is an XML-based format, it is a purely text representation. Ultimately, your SVG view produces only formatted text.

3.6 Assignment

4. Implement the views above. **Place all the code of your views in the `cs3500.animator.view` package.**

5. Document any further changes made to your models from the previous assignment: explain what was added, removed or changed (besides the package declaration), and why.

4 Running and Testing

In order to run your code as an application, you need two functionalities:

- Reading animation specifications from files
- Switching among the views implemented above

4.1 Reading from files

The skeleton code above provides you with a utility class, `AnimationFileReader`, that can read the animation files supplied for you to use. Because we do not know in advance what you've named your model, or precisely how your constructors work, `AnimationFileReader` in turn requires a `TweenModelBuilder<T>` object that describes constructing any animation, shape-by-shape and step-by-step. Think of this builder as adapting from the model interface that the `AnimationFileReader` expects to the one your model actually has.

6. Implement this builder interface, where the `T` parameter should be specialized to whatever your main interface is named that describes an animation (likely your model):

```
import cs3500.animator.util.*;

public final class WhateverYourModelImplementationIsNamed implements YourModelInterface
  ...
  public static final class Builder implements TweenModelBuilder<YourModelInterface> {
    // FILL IN HERE
  }
}
```

7. Implement a factory of views, with a single method that takes in a `String` name for a view—“text”, “visual”, or “svg”— and constructs an instance of the appropriate concrete view.

4.2 The `main()` method

Add the following class to your project:

```
package cs3500.animator;

public final class EasyAnimator {
  public static void main(String[] args) {
    // FILL IN HERE
  }
}
```

This `main()` method will be the entry point for your program. Your program needs to take inputs as command-line arguments (available in your program through the argument `args` above). Review the documentation for command-line arguments in a Java program.

The command-line arguments will be of the form

<-if name-of-animation-file> <-iv type-of-view> <-o where-output-show-go> <-speed integer-ticks-per-second>

Characteristics of a valid input are:

- In the above notation, each input set is denoted between "<" and ">" signs. These signs themselves are not part of the input.
- The sets may appear in any order (e.g. the -iv set can appear first, followed by -if and so on)
- Within a set, the input is in order. That is, if the user types -if then the next input must be the name of an input file, and so on.
- Providing an input file (the -if set) and a view (the -iv set) are mandatory. If the output set is not specified and the view needs it, the default should be `System.out`. If the speed is not specified and the view needs it, the default is 1.

This `main()` method will be the entry point for your program. You will need to create an Application run configuration in IntelliJ that chooses `cs3500.animator.EasyAnimator` as its main class. In this run configuration, you can also specify command-line arguments, such as the file you want to read in, and the view name you want to use. The options for the view name are "text", "visual" and "svg".

Here are some examples of valid command-line arguments and what they mean:

- `-if smalldemo.txt -iv text -o out -speed 2`: use smalldemo.txt for the animation file, and open a text view with its output going to `System.out`, and a speed of 2 ticks per second.
- `-iv svg -o out.svg -if buildings.txt`: use buildings.txt for the animation file, and open an SVG view with its output going to the file out.svg, with a speed of 1 tick per second.
- `-if smalldemo.txt -iv text`: use smalldemo.txt for the animation file, and open a text view with its output going to `System.out`.
- `-if smalldemo.txt -speed 50 -iv visual`: use smalldemo.txt for the animation file, and open a

visual view to show the animation at a speed of 50 ticks per second.

If the command-line arguments are invalid, the program should show an error message (see `JOptionPane` for how to open a popup error message) and exit.

4.3 Specifying command-line arguments through IntelliJ

You need to create an Application run configuration that chooses `cs3500.animator.EasyAnimator` as its main class. In this run configuration, you can also specify command-line arguments. When you run the program normally, it will use these command-line arguments.

4.4 Testing

You should be able to test your text-based view and SVG view sufficiently by parameterizing it over alternate input and output sources. We did the same thing in Assignment 3 — and it was in the Controller, in that assignment, because we didn't have an explicit View as part of our architecture.

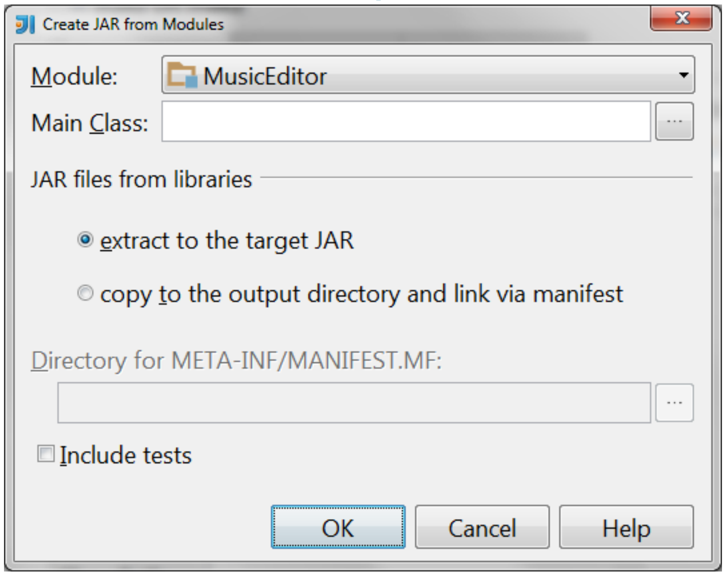
Unit-testing the visual view is optional.

5 Submission

- Submit any files created or modified in this assignment. We should be able to run your program successfully using files in your submission.
- Submit a text README file explaining your design. Make sure you explain your design changes from the previous assignment.
- Submit a file named `text-transcript.txt`, showing the output of the console view when rendering the `toh-3.txt` file at 20 ticks per second.
- Submit a file named `toh-at-20.svg`, showing the output of the of the `toh-8.txt` file at 20 ticks per second as an SVG file.
- Submit a JAR file (with extension `.jar`) file that can run your program.
- Your directory structure should be the same as in prior assignments — a `src/` directory and a `test/` directory — plus a third directory named `resources/` containing your README, your JAR file, the text transcript and SVG files as required.

To create a JAR file, do the following:

- Go to File -> Project Structure -> Project Settings -> Artifacts
- Click on the plus sign
- Choose `JAR` -> From Modules with dependencies. You should now see


Create JAR from Modules

Module: MusicEditor

Main Class:

JAR files from libraries

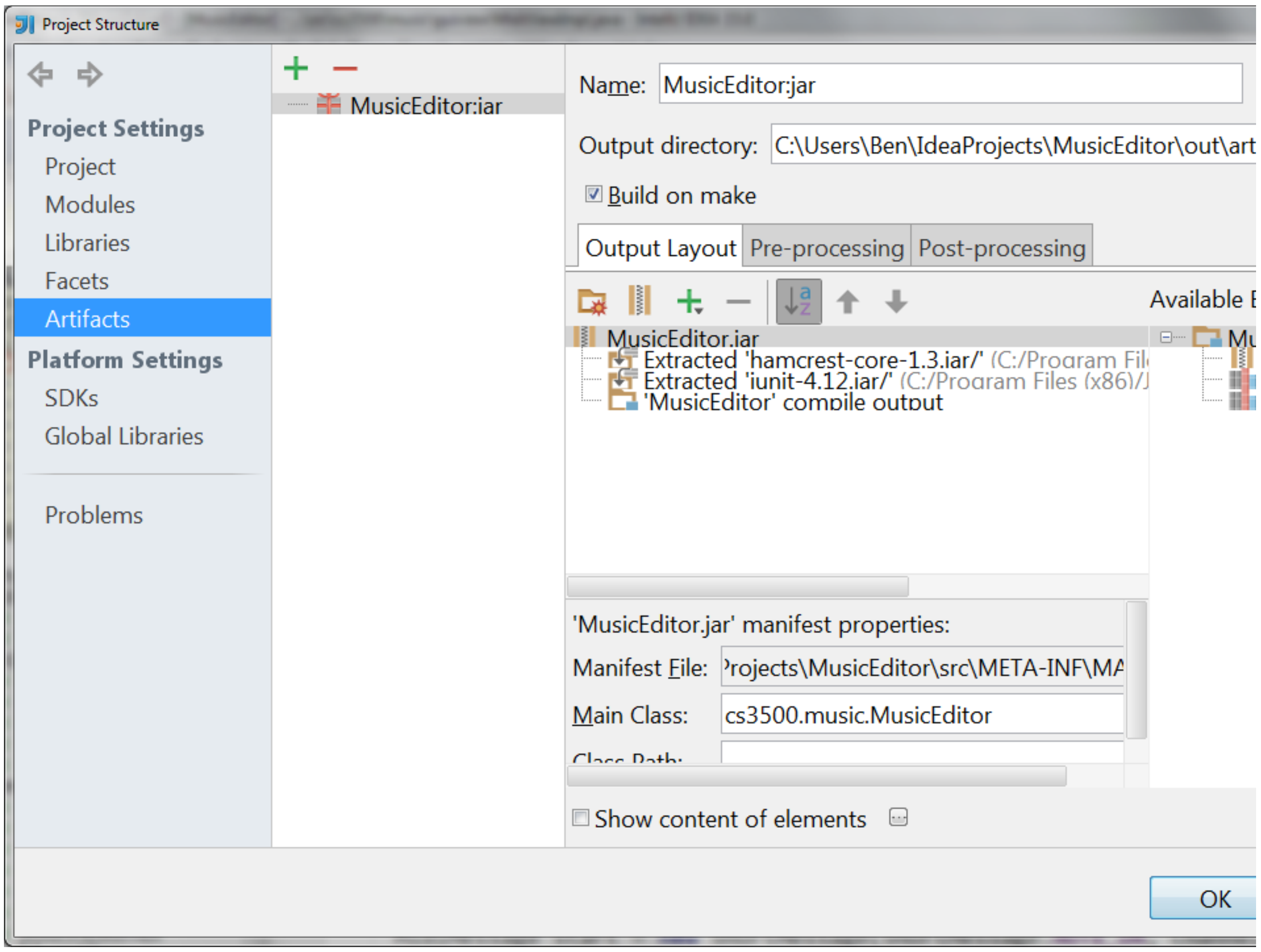
[✓ extract to the target JAR]

copy to the output directory and link via manifest

Directory for META-INF/MANIFEST.MF:

☐ Include tests

- Select the main class of your program (where you defined the `main(String[] args)` method)
- If you see a checkbox labelled "Build on make", check it.
- Hit ok
- You should now see something like



MusicEditor:jar

Name: MusicEditor:jar

Output directory: C:\Users\Ben\IdeaProjects\MusicEditor\out\art

☑ Build on make

```
MusicEditor.jar
  Extracted 'hamcrest-core-1.3.jar/' (C:/Program Fil
  Extracted 'junit-4.12.jar/' (C:/Program Files (x86)/J
  'MusicEditor' compile output
```

'MusicEditor.jar' manifest properties:

Manifest File: ›rojects\MusicEditor\src\META-INF\MA

Main Class: cs3500.music.MusicEditor

Class Path:

☐ Show content of elements

If now you see a checkbox labeled "Build on make," check it now.

- Make your project (click the button to the left of the run configurations dropdown, with the ones and zeros and a down-arrow on it). Your `.jar` file should now be in `<projectRoot>/out/artifacts/`.
- **Verify that your jar file works**. To do this, copy the jar file and your animation input files to a common folder. Now open a command-prompt/terminal and navigate to that folder. Now type `java -jar NameOfJARFile.jar -if smalldemo.txt -iv text -o out -speed 2` and press ENTER. The program should behave accordingly. If instead you get errors, review the above procedure to create the JAR file correctly. **Note that double-clicking on your JAR file will not test it correctly, because your program is expecting command-line arguments.**

6 Grading standards

For this assignment, you will be graded on

- the design of your view interface, in terms of clarity, flexibility, and how well it supports needed functionality;
- how well you justify any changes made to your model,
- the correctness and stylishness of your implementation
- whether your program accepts command-line arguments correctly
- whether your JAR file works correctly
- the comprehensiveness and correctness of your test coverage.

Please submit your homework to https://handins.ccs.neu.edu/ by the above deadline. Then be sure to complete your self evaluation by the second deadline.

1 there may be some performance issues from one browser to another. If you experience any lag, try a different browser first