For each of the rubric categories described below, say if this is a strong point of your program or not, and why. You are welcome to point point values/ranges, but do not need to. For example, you might argue:

Functionality: 20/20

Our code is very robust. In each screen, you can click whatever you want and it things change only when they are supposed to for program functionality. If you "sit on the keyboard" while the program is running, it does not interrupt, error, or change the functionality of our code (outside of computer commands and such, which we are not responsible for). Try catch statements have been used to catch IOExceptions and other potential errors with GUI, file-writing, and event handling. Anytime the toTxt() method is called within the Input class, it has a try-catch for if an error occurs with the file handling code. If-else suites and loops have been used in many places to handle different cases of events.

Design: 18/20

Our design is very good. Our design as described in the Program Specification document included in this submission has several advantages. First, we made it easy for two different people to develop code by splitting up input and animation into subclasses of Display. Also, exporting information to text files made it easy to communicate a lot of information between the two primary classes.

Creativity: 20/20

We created something completely new. We don't know of any other software that will do this for you. We found a problem and solved it. This will save choreographers a lot of time in setting formations. Also, being able to see transition on a screen before actually seeing it with people will alert people in advance to possible collisions. So this tool could be used by choreographers to save time and also it looks really cool.

Sophistication: 20/20

This project is very sophisticated.

We have over 700 lines of code for this project. A lot of work has gone into this.

With the input from the user by clicking buttons. It uses many swing components to create a very interactive and visual simulation. It handles many different events, offers and integrates user choices to simulate dancer choreography at the beginning and throughout, etc. It differentiates according to the situations; once the initial formation is established, the other larger

case is for next formations to be set up. Pop-ups and messages keep the user updated in what is occuring or will occur, and letting them know if/when something is not valid/appropriate. For example, if the user attempts to place two dancers in the same spot, the board will message and let them know while treating that event like it did not happen. It recognizes many different actions and cases. Another example is the random/scatter option at the beginning of each non-initial formation. They can choose to randomly assort the dancers on the stage for a formation. This is realistic and helpful in choreography that needs dispersed or another formation that is not yet set, or one that needs inspiration. The smooth sequence between panels provides updates and ease of use that is clear and easy to follow.

We also developed our own regression algorithm to find the shortest cumulative distance that everyone has to travel. The algorithm finds all of the possible permutations of the next stage positions and then calculates the cumulative distance that each person would have to travel in that permutation. Then, it finds the smallest cumulative distance and sets the next formation to that permutation.

Also, our classes that we developed more or less independently (Input and Animation) needed to play well with each other; we kept constants in the parent class Display and loaded the majority of the information through text files.

Broadness: 20/20

We ticked boxes 1, 2, 5, 6, 7, 8.

1. Java libraries: JButton and JPanel used to a large extent in the displaying of our program, you can see this by looking at the program running. Display, and all of its children, extend JPanel, and buttons are in Welcome, Input, and Animation. Also, changeListeners and actionListeners were used in the Input class to get information from button presses.

2. Subclassing: the Display parent class has three children: Input, Animation, and welcome. It holds variables that are constant across all three, and thus are not necessary to re-instantiate and store in each. For example, the available panel dimensions and the pixel to position/index scales. These are both important for consistency and clarity. The panel dimensions should not vary between each too much, or it would cut the flow or visually draw attention away from the actual simulation; instead confusing the eyes with things such dimensions changing unnecessarily. The scale is necessary so that the user can actually see the simulation and interact with it even if they don't have the best eyes.

5. Built In Data Structures: Used an Arraylist in the regression algorithm to hold permutations of the dancer arrays. Arrays cannot hold arrays, so this is integral to the regression analysis. Also, an arraylist holds the stages because we do not set the number of formations

inputted, so it can be added to the arraylist without worry of running out of space. Also there's too many arrays to count in this program, in the Stage class, the dancer[], all of the permutations of dancer[], cumulativeDist[]. In the input class, there's a button[][].

6. File Input/Output: the way the Input class communicates information to Animation. Input class takes in data from clicks of buttons and typing in information about dancers, and stores it in 3 text files. Animation takes in the text files (coordinates, colors, and names), parses them, and uses this information to store the algorithm. File readers and writers are used in Animation and Input respectively.

7. Randomization: if the user doesn't know what formations to choose, or simply wants a scatter formation, we have a randomize button that will place dancers in random positions. This method is especially helpful for a lazy choreographer in setting formations. This method also made it much faster to test different cases.

8. Generics: generics were used with the positions buttons. These buttons, a child class of the JButton class, holds Dancer type objects. This was necessary and made it easier to directly reference and adjust/store dancer information according to the buttons (representing positions on stage) that the user specifies and interacts with. Otherwise, each time something occurs with a button or something were to be edited or changed, two additional methods would be needed to connect each dancer's information and placement with a button while still allowing for quick runtime and non-convoluted code. Also used generics in ArrayList<E>, we stored Stages and Dancer[] in ArrayLists in Stage and Animation

Code Quality: 18/20

We used a similar commenting style to projects 1 and 2. We had consistent naming conventions. All mutator and accessor methods are named getVariable() and setVariable(newVariable). Code is clean and organized across files. Also we have very professional running software, especially in the Input and Animation displays. It feels professional.

Total: 116/100 → 100%!