Functionality 20 pts

Our program successfully compiles and runs, which is integral to the success of any program. Thankfully, due to the nature of our program, we only really had to worry about a handful of user inputs. These inputs included mouse inputs and key inputs. More specifically, the user has to click the screen in order to move from area to area, and the user has to click specific hitboxes on the screen (sometimes in specific sequences) in order to solve puzzles. Because of this, we didn't have to worry about many other common errors that could arise in other projects, for example, users loading incorrect file types into their code. All we had to worry about was user input from both the mouse and the keyboard--two inputs we believe to have handled quite robustly in our program. As of now, there are no holes apparent to us in our program. Unless the user intentionally chooses not to solve the escape room, nothing should prevent/hinder them from completing the escape room.

20 points.

Design: 20 pts

From a high level, we believe to have attacked this problem very logically. From the offset, we knew that our program would likely need to involve a room class, containing several instances of the area class, linked together in a way that would make moving from area to area relatively easy. From there, we knew that each area would need a few defining characteristics--an associated image to print to the screen, and a group of hitboxes responsible for moving to adjacent areas.

As with any good escape room, our room had to contain a series of complex, brain-teasing, yet fun puzzles for the user to solve. So, we thought to ourselves, what would be the best way to implement such a feature? Well, similar to the area class, we thought that any puzzle would have with it an associated image, and a group of hitboxes responsible for moving in and out of the puzzle. One key difference, however, was how a puzzle changes when it is solved. Thus, we decided to make the puzzle class a child class of the area class, overriding the method puzzleSolved(), as well as guaranteeing that all puzzles had a second image.

Outside of the classes described above, we had three more very important, substantial, and what we believe to be necessary classes in our program.

1. Hint Class: Our hint class allows us to guide the user along through our escape room. If, at any time, they decide to press h, they recieve a hint on screen. Each hint corresponds to an area/puzzle, and by keeping track of Hint objects, we are able to print out the correct hint for a corresponding puzzle, ultimately helping to guide stuck users through the room.
2. Inventory Class: Our inventory class allows us to provide the user with a graphical depiction of the items they currently possess in their inventory. In our escape room we

only have two items the users can possess, but with the use of this inventory object, we can track whether or not each of the two objects should be printed or not yet printed in the inventory. (One could imagine a more extensive version of this, tracking whether or not many more items should/shouldn't be included in the inventory).

3. myDS: The inspiration for this class came from one major problem we encountered while writing the program--tracking whether or not puzzles were solved. More specifically, we wanted puzzles to be solved if and only if the user put in the correct mouse/key input in the correct sequence. Due to the inherent structure of mouselistener/keylistener, we realized that solving this problem effectively would need to be more complex than a simple series of if statements within the mouseClicked method--this is because the mouseClicked method is called every time the mouse is clicked so we would require a large nested loop to ensure that the puzzle was solved correctly(if we wanted to just use if and else if statements). We, therefore, we decided that creating a data structure would be the most effective way to solve this problem, constantly checking to see if the correct code/sequence of keys/mouse clicks was entered after every user event. This data structure made checking to see if puzzles were completed very easy.

Finally, we had a series of less substantial, yet very important classes. These classes included Pair, Runner and Rectangle. These classes were responsible for storing an x and y coordinate (the pair class), a desired area on the screen (the rectangle class), and constantly listening to mouse input (the runner class).

We believe to have, without over-complicating problems, solved them in an eloquent and concise way.

20 points

Creativity: 20 pts

Our program excels in terms of creativity The idea of a virtual, interactive escape room is brain-teasing, fun, and unique. We included a number of puzzles that try to make the user think in an outside-the-box manner, and we hope that Amherst College users will appreciate a familiar setting on campus--the Mead Art Museum.

20 points.

Sophistication: 20 pts

A large part of our program's sophistication comes from the myDS data structure that we implemented (previously used in lab 7). This data structure allows us to accurately and simultaneously track both mouse and key input from the user, which ends up being important in tracking whether or not a puzzle has been solved or not. (Side note: a particular part of the program that we are very proud of is that in the chess puzzle, specifically that the user can use

both a combination of key inputs and mouse inputs to solve the puzzle. This is achievable thanks to the help of our data structure.)

In addition, a couple of other sophisticated aspects in our program include the following: the ability to move to adjacent areas freely and easily, the fact that we update area images after corresponding puzzles have been solved (even for areas with puzzles in the distance), and the printed user inventory.

Additionally, we believe that the way in which our program is connected holds a large part of the sophistication aspect. We integrate a number of classes together fairly well to ultimately create a game that combines these classes seemingly well.

Judging by your rule of thumb, this project is definitely sufficiently sophisticated. The coding certainly took far more effort than 6 labs. A large part of this, we believe, has to do with not working from a prior template, like we have done in the past. We explored a number of different avenues of Java including mouseListener and loading and printing images. Because of all of the above, we believe that we should earn 20 points on sophistication.

20 points.

Broadness: 20 pts

To obtain full credit, you should use at least 6 of the following in some justified way.

We used 6 of the 8 criteria in our project, justified below. Therefore, we believe that we should earn full credit on Broadness.

1. Java libraries that we haven't seen in class

The major java interface that we didn't see in class but had to learn a ton about in order to complete our project was the mouselistener interface. By importing this and properly implementing this interface we were able to access users' current mouse positions--accessing this information (these user mouse coordinates) ended up being essential to tracking whether or not many puzzles were solved by user input. Additionally, we used the Image library fairly extensively in our project and are now quite familiar with how this class works.

2. Subclassing (`extends` a class you created)

Whilst creating our program we found that the puzzle class would be a logical child class of the area class. That is, just like the area class we thought that the puzzle class should have a number of similar properties (an associated image, hitboxes, etc.) but that the puzzle class should have different properties when solved and that all puzzle classes should have a boolean to track whether or not the puzzle had been solved.

3. User-defined data structures

As discussed above, we utilised a slightly modified version of the myDS data structure we used in lab 7. This data structure allowed us to check whether or not puzzles were solved. Additionally, we implemented a linked list of areas which connected them together. Although it seems as though we used an array in our room class, we rely on many of the properties of our linked list of areas in our code, such as changing our current area to the left or right neighbor and checking a number associated with each area (not the index value of the array). Granted, the Area class is not defined by an API.

4. Built in data structures

We used arraylists in our area class in order to store rectangles corresponding to a given area. These arraylists of rectangles kept track of hitboxes corresponding to each area. The decision to use arraylists came mostly from the fact that we did not want to be constrained by a length of hitboxes because we wanted each area to be able to have its own length and add hitboxes at later times if we so desired.

5. File input/output

We used several dozen images as our file inputs for this program. These images were loaded and stored as variables in area/puzzle objects. In order to determine how to load these images, we spent quite a bit of time on stack overflow and oracle. Some of the resources we used are below. While loading these images, we ensured that we did not have an error if a file was named incorrectly by including a try catch loop with an IO Exception.

https://docs.oracle.com/javase/7/docs/api/java/awt/Toolkit.html
https://docs.oracle.com/javase/tutorial/2d/images/loadimage.html
https://docs.oracle.com/javase/7/docs/api/javax/imageio/ImageIO.html
https://stackoverflow.com/questions/13038411/how-to-fit-image-size-to-jframe-size

6. Randomization

We implemented randomization in the hint class. More specifically, for each puzzle we decided to encode three possible hints. We did this so that if people couldn't complete the puzzle off the first given hint, they could go about the room and look at different puzzles and when they came back to the hint they could have a shot at a different hint, which they may understand better.

20 points

Code quality: 15 pts

Our code is extensively and clearly commented--ie, each important variable, method, class is explained within the code to ensure that future developers (possibly ourselves) can comprehend what we were aiming to do.

Similarly, we found our organization across files to be pretty top-notch. We created a new class and .java file when we found it appropriate, and tried to ensure that all files would work together smoothly.

In terms of code cleanliness, we do think that given a bit more time we could improve in this category. While reading in many files is necessary for simply setting up this program (as we do in Room.java), we do believe we could clean up our code overall.

Finally, we do think our code and program is of semi-professional quality. We use complex solutions to solve tough problems, and we placed considerable care into making sure everything (all images, user inputs, etc.) in the program run smoothly and look aesthetically appealing.

15 points.