Required elements explained:

1. Object-oriented elements
   a. Classes:
      i. This is exemplified through my use of several classes, which include the following: Achievement, Upgrade, GameManager, and both of the controller classes.
   b. Subclasses:
      i. Only one subclass is used, the Upgrade class, which is a subclass of Achievement.
   c. At least one abstract class and/or interface:
      i. Technically I use more than one interface, although the one that I wrote myself would be the Achievable interface, which ensures that each possible achievement or upgrade earned by the user will show its information in a dialogue box. The other interface I used would be Serializable, which was used to write the current state of a GameManager object to file.
2. Code elements
   a. One or more collection classes
      i. The only collection class I used can be found in the GameManager class. An ArrayList with generic type of String can be found as one of the fields in this class with the name of "upgradeList". Its purpose is merely to keep track of the upgrades that have been purchased by the user as they play

through the game. This is done through helper methods also contained in the "GameManager" class such as the addUpgradeToList() method. These methods for adding members to the ArrayList aren't actually called until an instance of GameManager in the GameController class calls the method purchaseUpgrade(). Then, purchaseUpgrade() will call addUpgradeToList() with some useful String literal to represent the number of the upgrade, e.g., "1" will be passed as a parameter when the first upgrade is purchased by the user. The other significant time this collection class is used is when loadManagerFromFile() gets called in the MenuController class. By checking which values are stored in the ArrayList, the proper button state and label state for the game scene is determined prior to showing the stage. I should also mention that there is a method within GameManager called printCurrentUpgrades() that also uses the ArrayList, but it is purely for debug purposes as it only prints messages to the console after each upgrade is purchased.

b. Exception handling

   i. Exceptions are used fairly often throughout CoffeeClicker. In general, generic exceptions are used in try-catch blocks whenever a scene switch occurs that usually prints an error to the console. Anywhere reading and writing data occurs, such as in handleSave() in the GameController, and IOException is used in the try-catch block to that also then prints a message to the console with the appropriate file path if reading or writing data were to fail for some reason. Another example of catching an IOException and even a ClassNotFoundException can be found in the method handleLoadGameButtonAction() from the MenuController class.

3. Clearly defined model
    a. I believe I have adhered well enough to a MVC design. To me this is represented through my use of implementing Serializable in only one class: GameManager. All it takes to save the data of a current game is a valid instance of the GameManager class and an accompanying method call from the method in the controller class that is linked to the save button in the actual scene. None of the game logic is actually written inside any of the controller classes. In fact, the controller only facilitates the saving and loading of data or changing the state of the UI. Pretty much all of the game logic is found only in the GameManager class.
4. UI changes scenes
    a. This is relatively obvious after running the application. Again the code for switching scenes is found within both of the controllers. For example, the handleStartNewGameButtonAction() method in the MenuController class switches the scene to a new game from the main

menu. Conversely, from the GameController class, the handleReturnToMenu() switches the scene back to the main menu from the game screen. If one were to inspect the scene graph in each scene, the difference is rather apparent between them.

5. Accessing "About" information for the application
    a. The "About" information can be found as a MenuItem in the Menu scene. Clicking the MenuItem calls the handleAboutMenuItem() method in the MenuController class, which pops up a dialogue box and displays information about the developer as well as the application itself.
6. Saving and loading data
    a. Saving data is only possible from within the game screen. A MenuItem can be found in the scene called "Save Game" which saves the current state of the game and writes it to file at a location of the user's choice. The code for this can be found in the GameController class in the method called handleSave().
    b. Loading data is only possible from the main menu via the "Load Game" button. Pressing load game opens a file explorer that allows the user to open a previously saved file. Once the file is read, the data is fed to an instance of GameController so that it can be restored and the new game scene is then loaded. The code for this can be found in the MenuController class in the method called handleLoadGameButtonAction().