Cormac Duggan
Student ID: 17100348

In this paper I will explain the design choices and analyze the differences between the Object-Oriented design paradigm and its implementation of a to do list and the Imperative design paradigm and its implementation of the same to do list.

Beginning with the Object-Oriented implementation I decided to use a queue class along with separate classes to define an event versus a task. The queue class is initiated to create a list of events and classes that allows the user to add items to their list and remove any item they have completed only if that item is at the top of the to do list. Upon starting the program it will check the current directory for a to do list text file containing the previously used list and, if it exists, the items will be added to the current list. The user is then presented with a list of options upon running the program that allows them to select how they would like to interact with the list. Based on which option of task or event the user chooses to add to the list, different information will be requested from the user to appropriately fill out the object and place it in the to do list (queue class). Upon exiting the program the current state of the list will be written to a text file in order to save the info that has been added. If the file does not already exist it will be created. This way whenever the list program is run again the previous usage of the list is not forgotten.

In the other program, the imperative implementation, I attempted to retain as many features as possible from the Object-Oriented version without creating another Object-Oriented to do list. Similar to the previous the imperative version gives the user multiple different options upon loading the program. They are able to add tasks and events to the list and print out the list as well as exit the program. When deciding to add to the list they will also be required to input additional information to fill out the time, place, etc for said task or event. When printing the list similar to the Object-Oriented version all items on the to do list will be printed out in the same format and will be separated by a line of “~”s. This line, however, unlike the other implementation will not resize appropriately to the terminal’s size and instead will remain 10 characters long each time. Also unlike the other, this implementation does not have a save feature as that would require writing to a file which is considered an object in python and would thus break the Imperative paradigm.

The essential difference between the two implementations is the use of objects which behave based on their class versus the lack of objects and in their place the use of explicit commands designed to be executed in sequence. The imperative design is set to run commands from top to bottom and read down the file until it reaches a finishing state. In contrast to the object-oriented system implements this methodology somewhat when executing the to do list but otherwise uses the provided classes to access object properties that can interact with each other, the user, and the execution of the objects.

In the imperative version of the list each line of code is executed in sequence causing a change in the global state of the saved data. This constant changing of global variables, although not used here, can be used to create “side effects” and change the program state. Though no functions are used, the closest thing to a side effect can be found when the user chooses to exit and the global variable cont is changed to false to properly exit the program, or when any user input is added to the to do list and the toDo variable is changed appropriately. Contrastly, the object-oriented implementation makes changes to single instances of objects as the code is executed. The run function still runs similarly to the

imperative version and also contains a side effect to exit the program, however rather than changing the entirety of the data at one time, this implementation changes only a single part of the data upon user input. The side effect that would be changing the to do list is changed to adding or removing an object from another object rather than editing a single global variable and changing the program state.

Aside from the differences between the two paradigms there are also differences between available functionality of the two different implementations. The lack of use of objects in the imperative version invalidates a large amount of optimization options as well as many options to improve functionality. It could be python but, because files are treated as objects in python there is no way to save the to do list after the program is exited. On the other hand the object-oriented design does allow data for the list to be recorded to a file, however because of the way the data is formatted in objects it takes more effort to save the data to a file than it would to simply write a string to a file in the imperative implementation. The lack of functionality in the imperative version also requires the use of a delimiter to be able to remove items from the to do list which means we have to just hope the user listens to directions and doesn't type in the delimiter otherwise they could break the system. In the counterpart design a list is used to store the data which allows us to forget about the delimiter and just delete the item whenever we want thus making the program more resilient and less likely to break.

Be it a lack of functionality with easy printing and code that is relatively concise or all the functionality available with longer code and more difficult methods for printing, both paradigms and implementations have their merits and shortcomings. If I had to choose I would typically write an object-oriented implementation simply for the additional functionality I could provide it with and the fact that I have more practice writing this way.