### **PM3**

Tanay Khopey

Philip Clement

Camden Johnson

Ahman Hale

Jonathan Hsin

### **High-level Design (4%)**.

For our project we decided that event-based architectural pattern would be the best to structure our system. This is because a TODO list app at its core is many small events happening over and over to slowly build up someone’s TODO list.

Firstly starting with the pros of event-based architectural pattern, it has an anonymous event handling. This is perfect for a TODO list application as the user doesn’t need to know who or what is doing each of the tasks. Additionally, it’s very easy to add new features which is crucial in a TODO list app because there are so many already out there. The ability to be adaptable and easily be able to add a new feature to keep the app new or to stay ahead of other TODO list apps is imperative. Furthermore, one of the biggest uses for event-based architectural pattern is graphical user interfaces which is all a TODO list app is so it would fit this architectural pattern perfectly.

The biggest con for the event-based architectural pattern is that the there isn’t an order of execution. Each event doesn’t know when it has to take place or what has to happen before it. However, in a TODO list app this isn’t a very difficult problem to overcome as it is very easy to create checks for this. For instance, if a user tried to delete a task, but they haven’t made any tasks a simple if statement checking to see if there are any tasks would resolve this issue. Even though the delete component has no idea about its order of execution it would prevent it from deleting before a task is made.

### **Low-level Design (4%)**

We think that using a behavioral design pattern would be most effective when implementing our project. For our application, it will be important to keep the user interface refreshed in real time whenever changes are made to the list. For example, an addition, deletion, or swap of certain items. Specifically, the observer pattern would allow us to achieve this by implementing UI components where the observers are notified automatically whenever the state of the to-do list (observable) changes.

In addition, our application could use commands, an element of the behavioral pattern, to encapsulate actions such as adding, deleting, or swapping as command objects. This allows you to parameterize methods with requests and decouple the invoker from the actual operations. We could also benefit from the state element of the behavioral pattern by introducing different states for our application. For instance, there could be a “view only” state and an “editing” state.

Overall, we think that the behavioral design pattern would work best when implementing our project because it is what aligns closest with the dynamic of our system. The Observer pattern will help users remain up-to-date in real time while the command pattern allows us to encapsulate various actions. Finally, the state pattern will allows us to handle different states of the application effectively.

Code from our prototype:

#!/usr/bin/python3

"""

prints out a list starting at 1

"""

def printlist(strlist: list):

if len(strlist)>0:

print("Tasklist:")

for i in range(len(strlist)):

print(str(i+1)+". "+strlist[i])

"""

prints out the commands that the program takes

"""

def printcommands():

print("[q-quit, a-add to list, r-remove from list, s-swap list items]")

"""

swaps two items in a given list at indices e1 and e2

"""

def swapelem(e1:int, e2:int,strlist: list)->bool:

if e1<0 or e1>=len(strlist) or e2<0 or e2>=len(strlist):

print("Requested ID(s) out of bounds")

return False

elif e1==e2:

print("Can't swap two same IDs")

return False

else:

hold=strlist[e1]

strlist[e1]=strlist[e2]

strlist[e2]=hold

return True

"""

check that all elements in a list are integers

"""

def checkListIsInt(alist: list)->bool:

for x in alist:

if not str.isdigit(x):

return False

return True

run = True

tasklist=[]

print("Welcome to the A-list! This program will help you keep track of your tasks!")

while run:

#first print out the list and the available commands

print()

printlist(tasklist)

print()

printcommands()

#Then take in a command

command=input("Enter your command: ")

#when user wants to quit

if command=="q":

print("Thank you for using the A-list")

run=False

#when user wants to add a new item

elif command=="a":

command=input("Enter the task name you want to add: ")

tasklist+=[command]

#when user wants to remove an item from list

elif command=="r":

#if the list is empty remove can't be ran

if len(tasklist)==0:

print("List is empty, there is nothing to remove")

#if list isn't empty, ask for the item number

else:

command=input("Enter the task id you want to remove or c to cancel: ")

remove=True

#runs until remove is canceled or a succeeded

while remove:

#if user was prompted to enter another number but decided to cancel instead

if command=="c":

print("remove canceled")

remove=False

elif not str.isdigit(command):

command=input("please input a valid number or c to cancel: ")

#if the number is out of bounds then request another number

elif int(command)<=0 or int(command)>len(tasklist):

command=input("please input a number within the bounds of the list or c to cancel: ")

#if the number is valid

else:

print("item "+command+" was removed successfully")

tasklist.pop(int(command)-1)

remove=False

#when the user wants to swap items

elif command=="s":

#if the list has less than 2 items, nothing can be swapped

if len(tasklist)<2:

print("not enough elements to swap")

#if the lenght of the list is at least 2

else:

ids=input("Please input two list id's separated by space or c to cancel:\n")

ids=ids.split()

swap=True

#runs until swap is canceled or succeeded

while swap:

#if the swap was canceled

if ids[0]=="c":

print("Swap canceled")

swap=False

#if the input wasn't all ints or user inputted more than two entries

elif not checkListIsInt(ids) or len(ids)!=2:

ids=input("Please input two valid list id's or c to cancel:\n")

ids=ids.split()

#if user inputted two numbers, attempt to swap them

else:

success=swapelem(int(ids[0])-1, int(ids[1])-1, tasklist)

#if the swap was successful

if success:

print("Swap successful")

swap=False

#if the swap wasn't successful, inform the user and request another input

else:

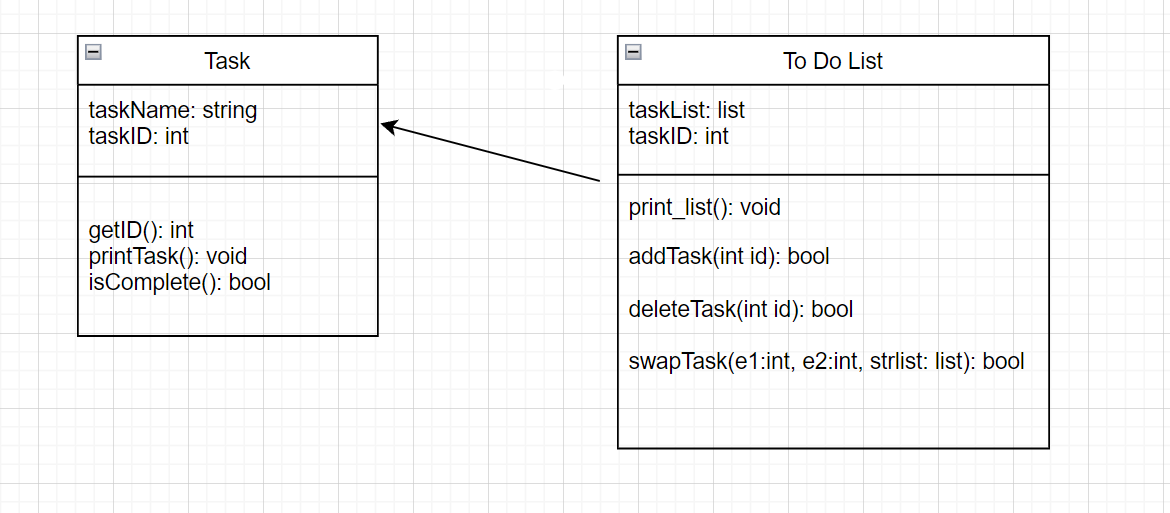
ids=input("Please input two valid list id's or c to cancel:\n")

ids=ids.split()

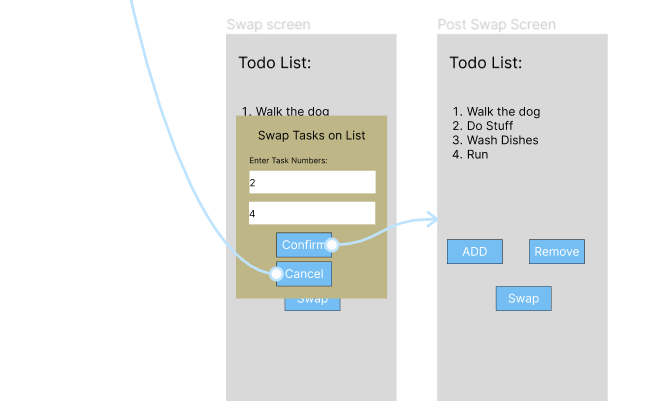
else:

print("please input a valid command")

**Class diagram:**

****

### **Design Sketch (4%)**



We wanted to create a visually simple application so that any user can quickly access and change the list. The four functions that we want to include are adding tasks to the list, removing tasks, swapping tasks, and ways to cancel each process. Adding and removing are crucial especially because that is the primary way people will be able to track their to-do list with each new task that they need to complete and a way to get rid of tasks that are already done. Swapping the order of tasks is important as well because oftentimes priority for certain tasks would change so it is also important to include it as a feature, this will also be an improvement to traditional to-do lists which are just written down on paper. The last feature is the ability to cancel each process for adding, removing, and swapping because it is a good way to help users prevent mistakes and errors as it gives them the option to stop the process of making changes that they may not necessarily want to make. These four functions are relatively basic but make up the most necessary functions to include in this to-do list application.

### **Project Check-In**

Complete [this survey](https://forms.gle/) to provide an update on your team progress on the project for this semester. Only one team member needs to complete this for the group.

### **Process Deliverable (3%)**

https://github.com/KoolBushido/todolist