Practical Task 4.2

(Credit Task)

Submission deadline: 10:00am Monday, April 27
Discussion deadline: 10:00am Saturday, May 9

General Instructions

This practical task focuses on ***program design***, mainly ***functional composition*** and ***code reusability***, and ***composition*** as a type of ***class relationships***.

In the first part, you will need to examine an existing source code to identify ***duplicate code***, which means that there are sequences of commands that occur more than once. Duplicate code is generally considered undesirable for a number of reasons, for example:

- Duplicate code makes your program lengthy and bulky.
- Duplication makes ***software maintenance*** more difficult.
- Duplication decreases your ***code quality***. Code quality is a necessity to make your software survive for long. Having duplicate code will make your code smelly and increases ***technical debt*** associated with it. The cost of repair of this debt is the amount of capital and time required to pay to a developer to simplify or de-duplicate it.
- Duplication leads to so-called ***Shotgun Surgery***: Suppose you wrote a buggy code and have to do a ***code review*** to find out the issue and then fix it. You now have to fix every location of that code, losing your time, efficiency and sometimes temper.
- Duplicate code increases security risks: When someone takes a code from somewhere and adds into other parts of program, they might forget about the holes and endings that plagiarized code has.

This issue related to the given code will require you to work on function composition, which implies that complex problems should be solved by decomposing them into many smaller problems so that each can be worked out easily. Eventually, those small pieces have to be put together to form the overall solution. One way of combining these small pieces is function composition, a mechanism to combine simple functions to build more complicated ones. Also function composition is a great tool that makes the code more compact and reduces noise. Because of the concise syntax, there are fewer possibilities to make mistakes like, for example, mixing up parameters.

In the second part, your task is to extend the program by adding new features and functionality. To make the design efficient, you should use a mechanism of ***class composition***, the most commonly used class relationship. In general terms, composition allows a class to contain an object instance of another class. Composition can be denoted as being an "***as a part***" or a "***has a***" ***relationship*** between classes. Thus, the main class becomes the front-end class and another represents the back-end class. The composition approach provides strong ***encapsulation*** because a change to a back-end class does not necessarily break any code that relies on the front-end class. Thus, it maintains the same access interface to the front-end despite of back-end changes.

Now, let's move to the task. Imagine, as it commonly happens in IT world, that you have been hired by BuggySoft, a leading producer of buggy software, to replace one of their staff who is leaving. You are asked to continue working on a simple Task Planner. They assure you that the task is very simple and should not take more than a couple of hours because the program looks fine from the perspective of user interface. So they just want you to extend the program a bit as their customer wants to add some additional features. You agree, and …

1. Open the source code file attached to this practical task. Suppose that this is the code inherited from the previous programmer. As you can see, the code itself is not large, but everything is placed in a single `Main` method. This code is messy and not documented, but you are lucky because you at least can compile and run it.

   You now must spend some time to understand how the code works, what it produces, and how it interacts with the user. The best way to do this is to create a new C# Console Application project and import the given file. You then should run the code step-by-step in debug mode to explore the control flow.

2. To demonstrate that you are a more experienced programmer than the leaving colleague, you should ensure function composition and avoid any code duplication because of the aforementioned reasons. Find all sequences of commands that should be placed together to form logically complete methods parametrised with a set of arguments. Work on this to make your code as clean as possible. In addition, add comments so that your methods and key points of your program make sense in case you want to return to it later.

   Once finished, save your source code file as '`RevisedCode.cs`'. You will need to submit this as part of your solution.

3. As you can see, this is a Task Planner program that partitions tasks into a number of categories. The tasks in a same category form a priority list. All tasks are printed in a table format. The program provides a very limited functionality and is for sure not competitive. Because of this, the customer requested the BuggySoft to add the following capabilities:
   - The user must be given an option to add new and delete existing categories. If a category is removed, all associated tasks should be removed as well.
   - The user must be given an option to delete an existing task.
   - It must be possible to move a task in its respective priority list by setting the place (priority).
   - It must be possible to move a task from one category to another.
   - It must be possible to highlight important tasks so that they appear in red in the terminal.
   - Every task must be characterised with a due date printed along with the name. This might need to enlarge the size of the table's columns.

   From what you see, the customer does not specify particular design and leaves this to you. You, as a programmer, are also free to implement the internal logic of the program as you decide. However, a proper design will likely rely on ***composition***. For example, you might have already noticed that tasks are to be "***a part of***" category and they are deleted along with their category. Remember that class composition ensures ***whole/part*** or ***parent/child relationship***; that is, the life cycle of the part (or child) is controlled by the whole (or parent) that owns it. If the parent object is destroyed, then the child objects also cease to exist.

   Note that you will likely need to create new classes to manipulate objects as data structures. For example, each task is characterised by a colour tag (to stress its importance) and a due date. This is not currently presented in the existing source code as well as any classes but the main class. Because we do not know how many classes you wish to create, we leave it to you and expect you to add them together to the '`FinalCode.cs`', which is to be the only file that you must submit as your final solution.

   In addition, you will deal with a collection of objects to represent categories and task priority lists. Therefore, we recommend you to focus on the use of dynamic `List<T>` data structure rather than a quite static array. You are also allowed to alter the existing code in order to improve it.

   Finally, you will have to significantly extend the existing menu to interact with the user to serve their requests.

Remember that we do understand that there is no unique correct solution to this problem. We expect you to provide us with a well-documented, neatly written code that has a justified program design.

Further Notes

- Study the connection between object relationships and class relationships, their differences, and how to implement them in your programs by reading Section 4.2 of SIT232 Workbook available in CloudDeakin → Resources.
- The following links will give you more insights about the program design aspects related to this practical task:
  - What is duplicate code? (https://www.codegrip.tech/productivity/what-is-duplicate-code/)
  - Code smell (https://en.wikipedia.org/wiki/Code_smell)
  - Everything you need to know about Code Smells (https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/)
- The following links will give you more ideas about class/object relationships, and specifically about composition:
  - https://creately.com/blog/diagrams/class-diagram-relationships/
  - https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/
  - https://www.c-sharpcorner.com/UploadFile/ff2f08/association-aggregation-and-composition/

Marking Process and Discussion

To get your task completed, you must finish the following steps strictly on time.

- Make sure that your program implements the required functionality. It must compile and have no runtime errors. Programs causing compilation or runtime errors will not be accepted as a solution. You need to test your program thoroughly before submission. Think about potential errors where your program might fail.
- Submit the expected code files as an answer to the task via OnTrack submission system. Cloud students must record a short video explaining their work and solution to the task. Upload the video to one of accessible resources, and refer to it for the purpose of marking. You must provide a working link to the video to your marking tutor in OnTrack.
- On-campus students must meet with their marking tutor to demonstrate and discuss their solution in one of the dedicated practical sessions. Be on time with respect to the specified discussion deadline.
- Answer all additional questions that your tutor may ask you. Questions are likely to cover lecture notes, so attending (or watching) lectures should help you with this **compulsory** interview part. Please, come prepared so that the class time is used efficiently and fairly for all students in it. You should start your interview as soon as possible as if your answers are wrong, you may have to pass another interview, still before the deadline. Use available attempts properly.

Note that we will not accept your solution after the submission deadline and will not discuss it after the discussion deadline. If you fail the deadline, you also fail the task and this may impact your performance and your final grade in the unit. Unless extended for all students, the deadlines are strict to guarantee smooth and on-time work throughout the unit.

Remember that this is your responsibility to keep track of your progress in the unit that includes checking which tasks have been marked as completed in the OnTrack system by your marking tutor, and which are still to be finalised. When marking you at the end of the unit, we will solely rely on the records of the OnTrack system and feedback provided by your tutor about your overall progress and quality of your solutions.