COMP11124 Object Oriented Programming

SOLID and Python Exceptions

Learning Outcomes

After you have completed this lab, you should be able to:

- Discuss the fundamental principles of SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) and their significance in software design.
- Critically appraise code examples to identify violations of SOLID principles and propose code refactoring to adhere to them.
- Demonstrate proficiency in handling exceptions in Python, including raising and catching exceptions.
- Understand the separation of concerns principle and how to apply it.

Topics Covered

Python: SOLID Principles, Python Exceptions, Separation of Concerns in the ToDO App.

Getting Started Task: Create a new copy of your ToDo app for this week (e.g. suffix it with _week7 or similar).

1. SOLID Principles

As discussed in the lecture, there are several OOP principles that, if followed, can help you to write better OO code. A commonly used set of principles follows the SOLID acronym. These principles are:

S: Single Responsibility Principle. Ensures classes have a single responsibility, enhancing modularity.
O: Open/Closed Principle. Allows for extension of classes without modifying existing code.
L: Liskov Substitution Principle. Subclasses can substitute superclasses without altering functionality.
I: Interface Segregation Principle. Clients only depend on the interfaces they use, promoting modularity.
D: Dependency Inversion Principle. High-level modules depend on abstractions, not low-level details.

Let us now look at each in detail, see some code examples and then apply them to our ToDoApp.
We will be making use of code that is hosted on GitHub. If you have not heard of GitHub, it is a platform that allows you to store and share your code with others. For software developers, it is a very good resource as it allows you to: see code examples, collaborate with others, and store your code in a safe place.

Exercise 1. Single Responsibility Principle

The single responsibility principle states that a class should have only one reason to change. Essentially, this means that a class should only have one job. If a class has more than one job, it is more difficult to maintain and extend and violates the SRP.

A term that we will often hear in this regard is "separation of concerns". This is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program.
Additionally, we also want to avoid "coupling". Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; and the strength of the relationships between modules. Good OO design aims to reduce coupling as much as possible.

For this principle, look at the following code example from GitHub:
https://github.com/heykarimoff/solid.python/blob/master/1.srp.py

Read the code and the explanations and you should be able to understand SRP within Python.
This code also mentions the Facade Pattern. This is a design pattern that provides a simplified interface to a larger body of code, such as a class library. (Highly recommended reading here: https://refactoring.guru/design-patterns/facade).
Another example of SRP (with good and bad examples) can be found here:
https://github.com/gicornachini/SOLID/tree/master/SRP

Now, think of our ToDoApp. We have the following classes (without the portfolio tasks included):

- **`Task`**
- **`RecurringTask`**
- **`TaskList`**
- **`TaskTestDAO`**
- **`TaskCsvDAO`**
- **`TaskPickleDAO (optional)`**

If we look at each class, we can see that without directly thinking about the SRP, each class has only a single responsibility. The **`Task/RecurringTask`** class is responsible for representing a task, the **`TaskList`** class is responsible for managing a list of tasks, and the DAO classes are responsible for reading and writing tasks to a file.
One part where we are very likely violating the SRP is the main module. We use this (although it is not a class) to handle the user input, print the menu and call methods of the various classes. This is a clear violation of the SRP. We should aim to separate the user interface from the business logic. We will fix this later in the lab.

Exercise 2. Open/Closed Principle

The Open/Closed Principle states that functionality should be open for extension but closed for modification. This means that you should be able to extend a class's behaviour, rather than modify it if we want to add a new feature. This is often achieved by using inheritance and polymorphism. You can find a set of Python code examples that demonstrate the OCP here:
https://github.com/heykarimoff/solid.python/blob/master/2.ocp.py
Another example is here: https://github.com/gicornachini/SOLID/tree/master/OCP (but please ignore the **`ABCMeta`**, **`abstractmethod`**, as we will cover that next week).

In very simple terms, to implement the OCP think about future changes that may be made to your code. If you think that you may need to add a new feature to a class, it would be best to create a higher-level class that the original class inherits from. This way, if you create a new subclass that is similar, but has some additional functionality, you can simply add the new functionality to the subclass (after inheriting) without modifying the original class.

In our ToDoApp, we have already implemented the OCP. We have created a Task class and a **`RecurringTask`** class that inherits from the Task class. The task has the main functionality such as the common attributes and methods, and the **`RecurringTask`** class has the additional functionality that is specific to recurring tasks; A good example of the OCP in action. If we were to add a new type of task such as a "**`HighPriorityTask`**" we could simply create a new class that inherits from the Task class and add the new functionality to this class. We would not need to modify the Task class at all.
However, there are other areas such as the DAO that we could improve. We could create a higher-level class that the DAO classes inherit from. This way, if we were to add a new type of DAO, we could simply add the new functionality to the new class without modifying the original class and retaining the same interface.

Exercise 3. Liskov Substitution Principle

The LSP is a principle that states that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. E.g. replacing our **`Task`** object with a **`RecurringTask`** object should not break the application (it does not!)
You can find a set of Python code examples that demonstrate the LSP here: https://github.com/gicornachini/SOLID/tree/master/LSP (where rather than inheritance, we simply use two separate classes) and https://github.com/heykarimoff/solid.python/blob/master/3.lsp.py . Please look through the code and the explanations and you should be able to understand functional implementations of LSP.

In our ToDoApp, we have implemented the LSP. We can replace a **`Task`** object with a **`RecurringTask`** object without breaking the application. This is because the **`RecurringTask`** class inherits from the **`Task`** class and has the same interface in terms of attributes and methods. Although there are some functionality differences (e.g. the completed method updates the due date in the **`RecurringTask`** class), the main functionality is the same and would not directly break the application.

Exercise 4. Interface Segregation Principle

The Interface Segregation Principle (ISP) emphasizes that clients should not be forced to depend on interfaces they do not use. In other words, it promotes the idea of smaller, more specific interfaces tailored to the exact needs of clients, rather than large, monolithic interfaces. In statically typed languages, this is often achieved by using "proper" interfaces which serve as a kind of blueprint for classes that implement them. By default, Python does not have interfaces, but we can still follow the ISP by creating classes that have a minimal and focused set of methods. (We will look at modules such as **`ABCMeta`** and **`abstractmethod`** (mentioned earlier) next week, which are used to create interfaces in Python). Think of it as a set of rules of which methods a class should have to be able to be used in a certain way but without providing the implementation of the methods.

In our ToDoApp, we adhere to the ISP by ensuring that the interfaces provided by our classes are minimal and focused on specific functionalities. For example, the **`Task`** and **`RecurringTask`** classes provide only the necessary methods for managing tasks, such as **`mark_completed()`** and **`change_date_due()`** and the Task class has no methods that the recurring task does not have. An example violation of the ISP would be if we had a method in the Task class, that would not be used in the **`RecurringTask`** class. An example is a fictional method such as "**`assign_standard_task_to_user`**" which would not be used in the **`RecurringTask`** class (as it's not a standard task).

Please read through, and see the examples of good/bad practices regarding ISP here:
https://github.com/gicornachini/SOLID/tree/master/ISP and here
https://github.com/heykarimoff/solid.python/blob/master/4.isp.py

Exercise 5. Dependency Inversion Principle

The Dependency Inversion Principle (DIP) advocates for high-level modules to depend on abstractions, rather than concrete implementations, thus decoupling them from specific details of lower-level modules. This principle facilitates flexibility, scalability, and ease of maintenance in software systems. Again, this is often achieved by using interfaces in statically typed languages. How to mainly do this in Python will be covered next week.

In our **`ToDoApp`**, the **`TaskList`** class interacts with tasks through the **`Task`** abstraction, allowing it to manage tasks without needing to know the specific details of each task type (whether recurring or not). Instead of accessing the stored data directly within the **`TaskList`** class, we use the DAO classes to read and write the tasks to a file. This allows us to change the way we store the tasks without needing to change the **`TaskList`** class (not violating the DIP).

Please look at the examples of good/bad practice regarding DIP here:
https://github.com/heykarimoff/solid.python/blob/master/5.dip.py and
https://github.com/gicornachini/SOLID/tree/master/DIP

2) Exception Handling

Previously we have looked at common errors that can occur in Python and how you would be able to fix them. However, some errors are not easily fixed such as opening a file that does not exist or trying to access an index in a list that does not exist based on user input. Those so-called exceptions can be handled in Python and you could write code that allows you to "fall back" to a default value or print a message to the user.
Please look at the following tutorial to understand more about exception handling in Python:
https://docs.python.org/3/tutorial/errors.html
Then also follow the tutorial here as well: https://www.w3schools.com/python/python_try_except.asp

After this, you should understand the usage of the try, except, else and finally blocks in Python.

Let us now apply this to our ToDoApp:

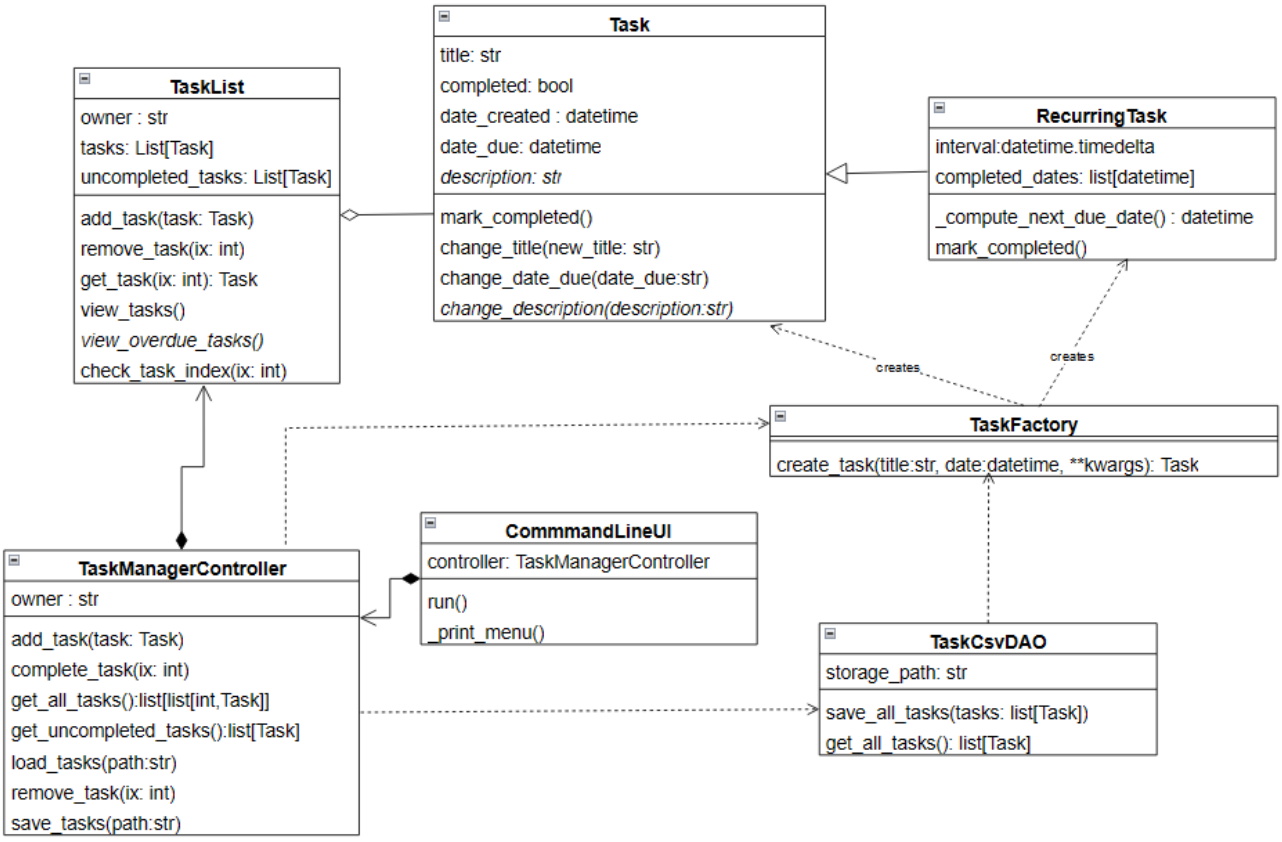
Task: Modify the main module to handle errors that occur when users try to do something with a task that does not exist (everywhere where the **`get_task`** method is called). For this, you should:
- use a **`try-except`** block to catch the **`IndexError`** that is raised when the user enters an index that does not exist.
- print a message to the user that the index does not exist and ask them to try again.
- Note: you may violate the DRY principle here. We obtain and handle the same error in multiple places. We will fix this later in the lab, just keep it in mind for now.

If you now run the **`main.py`** file and try to do something with a task that does not exist, you should see that the program does not crash and that you are prompted to try again.

3) Putting it together for the ToDoApp

Although a large chunk of our ToDoApp is already well designed and follows the SOLID principles, there are some areas that we can improve.
The one that we are going to focus on in this part is the main module. As mentioned earlier, the main module is responsible for handling the user input, printing the menu, and calling methods of the various classes. But how would you turn this into an actual User Interface through an App or Desktop application?
We want to achieve a separation of concerns and reduce the coupling between the various parts of the program. We have done so already in the form of removing the persistence logic from the main module and moving it to the DAO classes but want to also start to separate the user interface from the business logic.

Look at the following class diagram:



Here, we have moved the presentation logic to a new class called **`CommandLineUI`**. This class is responsible for handling the user input and output. We also have created a new class called **`TaskManagerController`**. This class is responsible for accessing and calling the methods of the various classes and handling the business logic.
We also have included a new task, **`TaskFactory`**. This class is responsible for creating new tasks and includes options for determining whether a task is a recurring task or not. The "Factory" pattern is a design pattern that allows you to create objects without specifying the exact class of object that will be

created. We will focus on those design patterns in one of the later classes, but you can use the code for this class here:

```
class TaskFactory:
    @staticmethod
    def create_task(title:str, date:datetime.datetime, **kwargs:Any) -> Task:
        if "interval" in kwargs:
            return RecurringTask(title, date, kwargs["interval"])
        else:
            return Task(title, date)
```

To use this Factory, you can call it like this:

```
normal_task = TaskFactory.create_task("Do homework", datetime.datetime.now() +
datetime.timedelta(days=3))
recurring_task = TaskFactory.create_task("Go to the gym", datetime.datetime.now(),
interval=datetime.timedelta(days=7))
```

Note that it determines whether a task is a recurring task based on whether the "interval" parameter is passed to the method. If it is, it creates a **`RecurringTask`** object, otherwise it creates a Task object. You also see that we have added a new method to the **`TaskList`** class called **`check_task_index(ix: int)`**. We want to follow the DRY principle, and at the current stage, we obtain and handle the same error in multiple places. This method is responsible for returning a Boolean value that indicates whether a task with a given index exists in the task list, and you should add it to your code:

```
    def check_task_index(self, ix: int) -> bool:
        return 0 <= ix < len(self.tasks)
```

We can then use this method in the **`TaskManagerController`** class to check whether a task exists before we try to do something with it.

Task: Modify the main module to separate the user interface from the business logic. For this, you should:
- create a new module called **`ui`** and create a class within this module called **`CommandLineUI`**. This class should be responsible for handling the user input and output. This includes all the printing of the menu (within the "**`_print_menu`**" method) and the handling of the user input including the "**`if, elif, else`**" statements. This class should not access the **`TaskList`** nor **`Tasks`** directly, rather it should call the methods of the **`TaskManagerController`** class.
- create a new module called **`controllers`** and create a class within this module called **`TaskManagerController`**. This class is responsible for the actual business logic, including the creation of the task list, adding tasks, completing tasks, etc. This class should not handle the user input or output, rather it should call the methods of the **`TaskList`** and **`TaskFactory`** classes and return the results to the **`CommandLineUI`** class.
- use the **`TaskFactory`** class everywhere where a task is created.

If you have added any portfolio tasks to your ToDoApp you can (optional) also add functionality that handles users' login, showing the overdue tasks and allowing for a task to change its description.

Note that this is a larger task, and you may not be able to complete it in the time that we have left. However, it is a good exercise to think about how you would do this and to start implementing it, even if

you do not finish it. The solutions will be provided on Aula after the lab so that you can either integrate them into your code or use them as a reference for future projects. It would be beneficial if you copy/paste your current ToDo app into a new folder and work on the new version there (suffix the folder name with "_solid" for example).