Lab 6 - SOLID Design Principles

Rahul Kukreja & Jeffrey Yim

jyim3@bcit.ca

Welcome!

By the end of today's lab we will:

1. Be able to read and understand UML Class Diagrams and translate them into working code
2. Be able to identify coupling and dependencies in a given codebase
3. Write object oriented code that is modular (decoupled) and adheres to SOLID principles

This lab consists of two tasks. Task 1 has you examining existing code that has been provided for you. Task 2 is when you must refactor the code from Task 1 to write modular object oriented code that follows SOLID principles. Don't forget to ask questions if anything is unclear, confusing or if you are stuck.

Grading

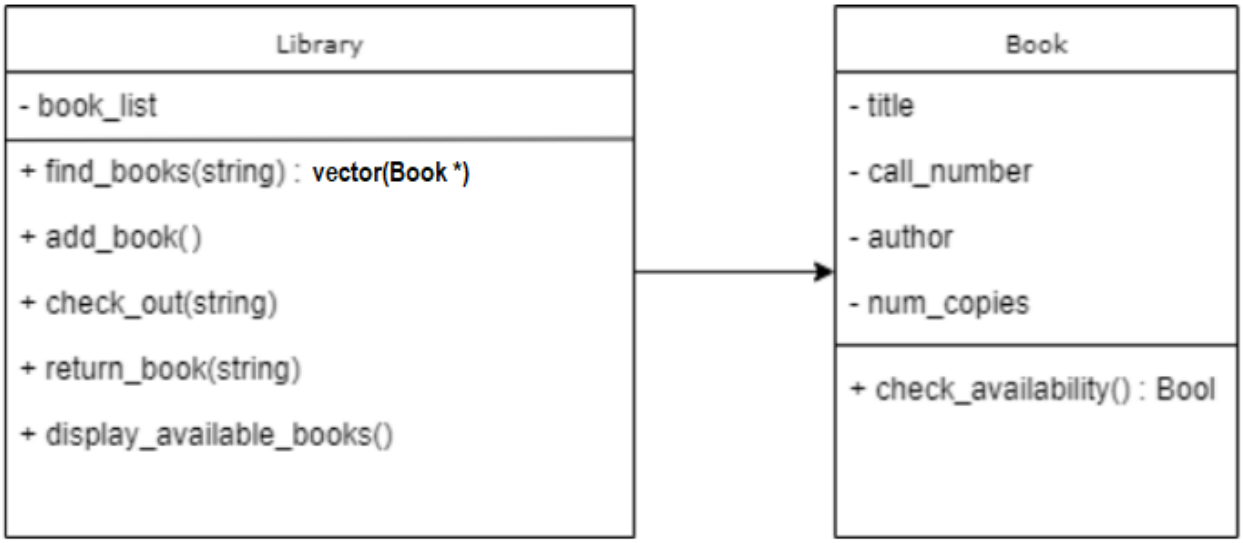
This lab and all future labs will be marked out of 10

For full marks this week, you must:

1. **(2 point)** Commit and push to GitHub after each non-trivial change to your code
2. **(3 points)** Generate a correct solution to the problem(s) in this lab
3. **(3 points)** Successfully test the library behaves exactly as described in this document
4. **(2 points)** Write code that is consistently commented and formatted correctly using good variable names, efficient design choices, atomic functions, constants instead of magic numbers, etc.

Task 1: UML Class Diagrams

Take a look at the UML Class Diagram below This is an extremely simple, bare-bones conceptual model of how a Library program may function. Libraries can search, check out or return books. The first task is to read over and understand a library program based on this class diagram. Remember, **commit often!**



1. Clone the project from: **https://classroom.github.com/a/hLQYyt3j**

2. The Library and Book files have already been written for you. Steps 2-13 explain what has already been written.
3. Examine both classes and see how they match the UML diagram along with some extra methods.
4. The `operator<<` function should return a string with the book details in a nicely formatted manner.
5. The `check_availability()` method should check the number of copies available and return **true** if there is at least 1 copy available, **false** otherwise
6. I've created a file called `library` and implemented the Library class. Note: I've included the `book` file for library to use books.
7. The `find_book()` function should accept a string `title` as a parameter and search the list of books for the title and return if it exists.
    - If the user provides incorrect input, print out a helpful message stating that the book requested could not be found
8. The `add_book()` function creates a new book based on input from the user and appends it to the `book_list` if it doesn't exist.

9. The `remove_book()` method should accept a `call_number` string as a parameter and attempt to delete the item from the `book_list`.

10. The `check_out()` method should accept a `call_number` string as a parameter and attempt to find the book. If found and available, decrement the number of available copies. It notifies the user if the book is unavailable for check out.
11. The `return_book()` method should accept a `call_number` string as a parameter, and increment the number of copies available for that book if it is found.
12. The `display_available_books` method should print out the list of books and their details in a nice formatted manner.

13. There is a main function that creates a new library in `main.cpp`. Run the program to test out the functionality and examine the code to fully understand how the entire program works.

Task 2: Using SOLID Principles and dealing with Dependencies & Coupling

The code provided so far can clearly be improved. Right now, any change in `Book` will eventually cause changes in the `Library`. `Library` is not only dependent, but **coupled** with `Book`. We will refactor our code, and modify the class diagram. Let's begin!

1. Read instructions 2-8 before writing any code. Sketch a UML Class diagram to help visualize your solution. Come chat with me if you need help with design.

2. Let's refactor our code. We need to split our `Library` class into `Library` and `Catalogue` (Why are we doing this? How does this improve our code?). The `Catalogue` class will now be responsible for maintaining a list of books. Move all the functions and code related to searching, adding and removing books to this new class.

   From this point onwards I won't dictate which class should go into which file. Think about it for a second and feel free to create new files and restructure your program as you proceed through this lab.

3. The `Library` class still needs to be able to access and retrieve book information, but it doesn't need to concern itself with how the search and retrieval occurs. (Which SOLID principle and OOP principles are we working with here?).
   - Is the book list implemented as a vector? What kind of search algorithm is being used? None of these matter to the Library class. All it needs to do is call the `find_book()` method in the Catalogue class.
   - By refactoring our code this way, we can say that the **Library is dependent on the Catalogue class, but decoupled from the implementation of the book list**.
4. Say the Library underwent a massive upgrade and got access to extra funds through grants! It now maintains `DVD's` and `Scientific Journals` in addition to `books`! (How exciting).
   - This presents an issue. The Catalogue is highly dependent and coupled with the Book class. We need to use abstraction and interfaces to decouple this unhealthy relationship. How would you do this? Hint: Refer back to our lecture on Inheritance, Interfaces and Abstract Classes.
5. Implement a `Journal` class (Journals have `author`, `issue number`, and a `publisher`), and a `DVD` class (DVD's have a `release date`, and a `region code`). What attributes would be shared amongst Books, DVDs, and Journals? `call_num`, `title`, and `num_copies`? What changes would you need to make to the Catalogue

class? Would this have any effect on the Library Class? Do you need new files? Do you need to restructure the existing files?

6. Let's create a (possibly static) `LibraryItemGenerator` class that is responsible for providing the user with a list of library item types, accepting input and generating that type of item.
7. Add an `add_item()` function to the `Catalogue` class that is responsible for redirecting the user to the `LibraryItemGenerator` class, creating an item, and then adding it to the catalogue's vector of library items. (Your catalogue should maintain a single vector of library items. It should **not** maintain separate vectors for books, DVD's and Journals)
8. Clarification: Generally, all the functionality in the main menu should now support Books, DVDs, and Journals. Pro-tip: Supporting all item types will require minimum work if properly using pointers, polymorphism, and SOLID design:
    - Display all items
    - Check Out an item
    - Return an item
    - Find an item
    - Add an item
    - Remove an item

That's It!

That's all for this lab. I hope you had fun writing, architecting and refactoring code. Don't hesitate to ask questions if you still don't understand Dependencies, Coupling, Inheritance or the SOLID principles.

Ensure you push your work to github classroom. I'd like to see sensible git commits comments, and commits must take place at logical points in development.