Design Critique Report:

For the group (Dylan Barry, Robert Smyth, Calvin Wong and Donnacha O'Donoghue)

Features (A general critique + where SOLID principles were applied)

A general critique

To sum up the general structure of the project, we elected to create four general classes (Library, Book, eBook, Customer) that represented real-life objects (Which is an OOP principle, the less abstraction the mind needs to make to real-life objects the easier it is to code). In a real-life project these classes might have pulled data from a database using an API, but we chose to hard code the data in, as the project is simply a C# prototype with no functionality.

The project of course follows the principle of Encapsulation in AEIP; variables are set to private while methods are made public and used to change them, reducing the chance of human error. Polymorphism occurs in the eBook class, an inherited class of Book, where the displayBookInfo() method is overwritten by the child class eBook to print out a unique double variable called sizeInMB.

The "Library" class holds a hard-coded ArrayList of "Book" and "EBook" objects, (the EBook class being a child class of Book, with a 'double sizeInMBs' variable simply to demonstrate the I – inheritance principle of AEIP)

Example:

public class EBook : Book // EBook inherits from Book

The Library class gets instantiated as soon as you boot up the project and is handed back and forth between the 'Main Menu' form and the other forms, carrying over any changes.

We tried as much as possible to make sure coding was not done mostly in the forms, but that these classes (Library, Book etc.) were called using methods like for example, addBook() in Library is called by the addBook form instead of all the coding being contained in the form, which would be messy and not OOP anyway. A portion of the coding was done in the forms anyway, because in some cases it was much more convenient.

The UI design was built quickly due to sprint deadlines and naming conventions were not given much attention, but we improved them after some feedback between us. A consistent layout was sought with exit buttons on every form. In a real-world setting, these forms would need refactoring to improve usability and maintainability. We did run into an issue where if we tried to refactor the projects name within visual studio, the whole project would crash. This would be an example of technical debt in our project. Not adhering to proper conventions from the start made things harder to achieve later down the line.

If this were a real-world project, we would have thought much harder and longer, “How do we make some of these long methods (e.g., addBook() in Library) as short as possible, taking in as little variables as possible? How do we split them into multiple smaller methods, making maintenance and troubleshooting and adding onto that code much easier?” However in the noticeably short, small scope of a class project, we never really encounter those problems that you would if you made spaghetti code like this in real life.

An example of a monster method (taking in a whopping 6 variables!):

```
public static void addBook(String title, String author, String description, Double sizeInMBs,
Library library, bool checkedOut)
    {
        //adds book to DB/ArrayList if all fields have been filled and it does not clash with
another stock

        // Basic validation
        if (string.IsNullOrWhiteSpace(title) ||
            string.IsNullOrWhiteSpace(author))
        {
            MessageBox.Show("Title and Author are required.");
            return;
        }

        // Generate a new ID automatically
        int newId = library.Books.Count + 1;
}

        Book bookToAdd;

        // Is it an EBook?
        if (sizeInMBs == -1)
        {
            Book b = new Book(newId, title, author, description);
            if (checkedOut)
                b.Checkout();
            bookToAdd = b;
        }
        else
        {
            // Create an ebook
            EBook eb = new EBook(newId, title, author, sizeInMBs, description);
            // Set its checked-out status
            if (checkedOut)
                eb.Checkout();
            bookToAdd = eb;
        }

        // Add to the library
        library.Books.Add(bookToAdd);
        MessageBox.Show("Book Added Successfully!");
    }
}
```

For example, for a start, we could have passed in the variables into a “validation” method first, which then calls the addBook() method - splitting it into two smaller methods from the get-go.

SOLID principles

S - Single design principle

This is followed to the letter and quite naturally implemented in our design, the Book, EBook, Library and Customer classes do not escape their scope. A Library for example manages itself. It adds books, but it does not micro-manage books, it does not change the specific details of a given book, it simply adds books to its ArrayList and does not function outside of being a Library and managing Books. Meanwhile a Book for example can change itself, its own details, but it cannot call a method that adds it to a library.

O – Open/Closed principle

This principle is about not duplicating code, not rewriting old code. Open for extension, but closed for modification. An example is EBook being inherited from Book. We didn’t modify the old Book class code and risk breaking it, we simply added a new child class EBook that meets those slightly different but otherwise the same requirements.

L – Liskov Principle

A child class should work anywhere the parent class does. Essentially, I would take this to mean that the parent is either smaller then the child class or rather, any method that the parent class uses, should have implementation in the child class.

In our project, we have EBook derived from Book with zero issues, as EBook is very simply just the same as Book, it calls the Book constructor for the shared variables and then calls its own constructor for fileSizeMB, its only unique variable. It also practices polymorphism in overwriting the displayBookInfo method for Book, because it must also display its own unique variable for file size.

Example:

```
public EBook(int id, string title, string author, double fileSizeMB, string description)

        : base(id, title, author, description) // parent constructor call, it calls that then calls this constructor for the fileSizeMB variable

    {
        this.fileSizeMB = fileSizeMB;

        this.description = description;

    }


    // AEIP - Polymorphism: overrides DisplayInfo from Book to include file size from EBook

    public override void displayBookInfo()

    {

        base.displayBookInfo();

        Console.WriteLine("File Size: " + fileSizeMB + " MB");

        Console.WriteLine("Description: " + description);

    }
```

I – Interface Segregation Principle (ISP)

This principle is, “Don’t make huge interfaces with many methods that are not implemented by the implementation classes” - Methods should be divided into many smaller interfaces.

This doesn’t apply to our project as we did not create any interfaces.

D – Dependency Inversion Principle.

“Depend on abstractions, not concrete classes” - This is fulfilled in the Book class being used as a generic base class, where Book works, EBook will work also.