SOFTWARE DESIGN AND ARCHITECTURE

Project Assignment # 01

Group Members

Muhammad Ahsan.	406267
Muhammad Affan Amir	405260
Muhammad Nabeel	413522
Saif Muhammad Sheikh	429592
Ibrahim Qaiser	405459

Applying Dependency Inversion Principle

Before Code:

```
public class Database {

    private ArrayList<User> users;
    private ArrayList<String> usernames;
    private ArrayList<Book> books;
    private ArrayList<String> booknames;
    private ArrayList<Order> orders;
    private ArrayList<Borrowing> borrowings;

    private File usersfile = new File("C:\\Library Management
System\\Data\\Users");
    private File booksfile = new File("C:\\Library Management
System\\Data\\Books");
    private File ordersfile = new File("C:\\Library Management
System\\Data\\Orders");
    private File borrowingsfile = new File("C:\\Library Management
System\\Data\\Borrowings");
    private File folder = new File("C:\\Library Management System\\Data");

    public Database() {
        // check and create files
        getUsers();
        getBooks();
        getOrders();
        getBorrowings();
    }

    public void AddUser(User s);
    public int login(String phonenumber, String email);
    public User getUser(int n);
    public void AddBook(Book book);
    private void getUsers();
    private void saveUsers();
    private void saveBooks();
    private void getBooks();
    public Book parseBook(String s);
    public ArrayList<Book> getAllBooks();
    public Book getBook(int i);
    public void deleteBook(int i);
    public void deleteAllData();
    public void addOrder(Order order, Book book, int bookindex);
    private void saveOrders();
    private void getOrders();
    public boolean userExists(String name);
```

```
    private User getUserByName(String name);
    private Order parseOrder(String s);
    private void saveBorrowings();
    private void getBorrowings();
    private Borrowing parseBorrowing(String s);
    public void borrowBook(Borrowing brw, Book book, int bookindex);
    public ArrayList<Borrowing> getBrws();
    public void returnBook(Borrowing b, Book book, int bookindex);
}

public interface IOOperation {

    public void oper(Database database, User user);

}

public class BorrowBook implements IOOperation {

    @Override
    public void oper(Database database, User user) {
        // create frame
        // for borrowing book
    }

}
```

Why is this not adhering to DIP?

- **Direct Dependence on Database Class:** The BorrowBook class directly references the Database class in its oper method. This tightly couples BorrowBook to the specific implementation of Database.
- **Limited Reusability:** The oper method takes a Database object as an argument. This makes it difficult to reuse the BorrowBook logic with different data storage mechanisms (e.g., a cloud database).
- **Lack of Abstraction:** In order to separate high-level policy from low-level specifics, there is insufficient abstraction. For instance, it is challenging to switch between multiple data access implementations without changing the client code since there is no abstraction or interface for data access activities.
- **Violating Encapsulation:** The Database class has a few methods that are specified as private, such as getUsers, saveUsers, saveBooks, getBooks, saveOrders, getOrders, getUserByName, parseOrder, saveBorrowings, getBorrowings, and parseBorrowing. These techniques, however, belong in the internal implementation details and shouldn't be available through the interface.

Refactored Code:

```
public interface IDataAccess
{
    public void AddUser(User s);
    public int login(String phonenumber, String email);
    public User getUser(int n);
    public void AddBook(Book book);
    public void getUsers();
    public void saveUsers();
    public void saveBooks();
    public void getBooks();
    public Book parseBook(String s);
    public ArrayList<Book> getAllBooks();
    public Book getBook(int i);
    public void deleteBook(int i);
    public void deleteAllData();
    public void addOrder(Order order, Book book, int bookindex);
    public void saveOrders();
    public void getOrders();
    public boolean userExists(String name);
    public User getUserByName(String name);
    public Order parseOrder(String s);
    public void saveBorrowings();
    public void getBorrowings();
    public Borrowing parseBorrowing(String s);
    public void borrowBook(Borrowing brw, Book book, int bookindex);
    public ArrayList<Borrowing> getBrws();
    public void returnBook(Borrowing b, Book book, int bookindex);
}

public abstract class Database implements IDataAccess
{
    private ArrayList<User> users;
    private ArrayList<String> usernames;
    private ArrayList<Book> books;
    private ArrayList<String> booknames;
    private ArrayList<Order> orders;
    private ArrayList<Borrowing> borrowings;

    public Database();

}

public class FileDatabase extends Database{

    private ArrayList<User> users;
    private ArrayList<String> usernames;
    private ArrayList<Book> books;
```

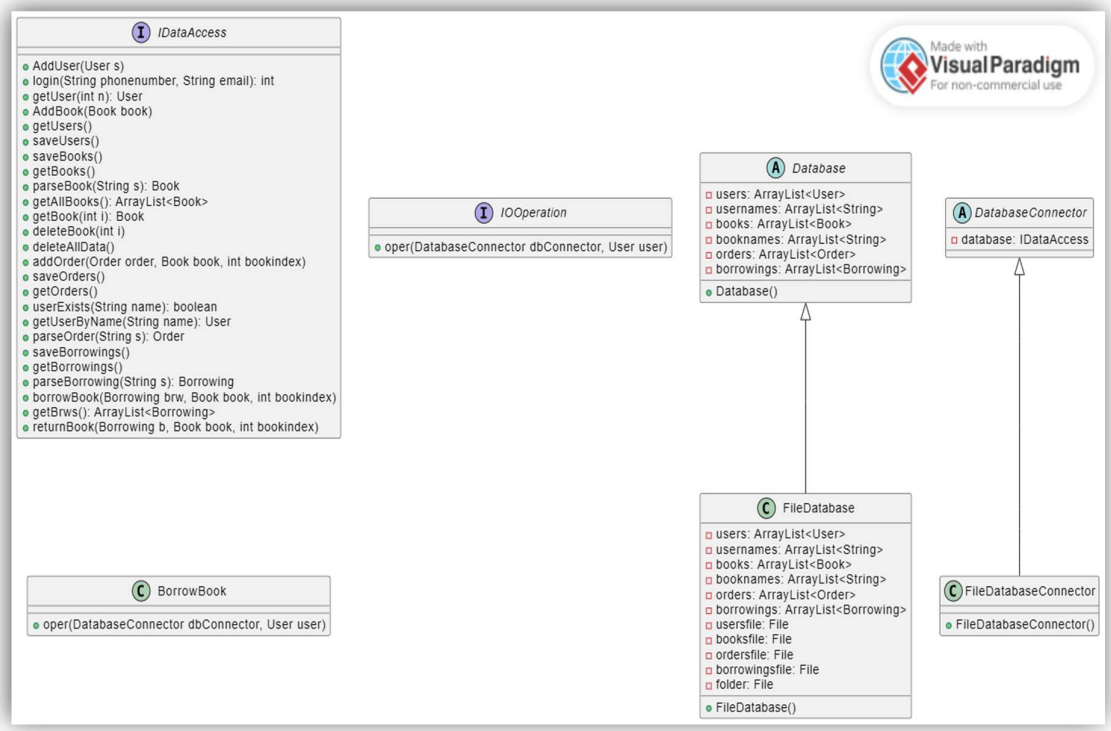
```
    private ArrayList<String> booknames;
    private ArrayList<Order> orders;
    private ArrayList<Borrowing> borrowings;

    private File usersfile = new File("C:\\Library Management
System\\Data\\Users");
    private File booksfile = new File("C:\\Library Management
System\\Data\\Books");
    private File ordersfile = new File("C:\\Library Management
System\\Data\\Orders");
    private File borrowingsfile = new File("C:\\Library Management
System\\Data\\Borrowings");
    private File folder = new File("C:\\Library Management System\\Data");

    public FileDatabase() {
        // check and create files
        getUsers();
        getBooks();
        getOrders();
        getBorrowings();
    }
}

public abstract class DatabaseConnector
{
    private IDataAccess database;
}

public class FileDatabaseConnector extends DatabaseConnector
{
    public FileDatabaseConnector() {
        database = new FileDatabase();
    }
}

public interface IOOperation {

    public void oper(DatabaseConnector dbConnector, User user);

}

public class BorrowBook implements IOOperation {

    @Override
    public void oper(DatabaseConnector dbConnector, User user) {
        // create frame
        // for borrowing book
    }
```

```
}
```

How refactored code is adhering to DIP?

- The code doesn't directly depend on the Database class and accesses the database through an API (Database Connector). The database connector contains the actual database i.e. FileDatabase, CloudDatabase, or an SQL Database. Each database class separately implements the functions of IDataAccess. Other class like BorrowBook gets data from the IDataAccess object of the Database Connector API.
- Because high-level modules rely on abstractions (IDataAccess), which may be implemented by a variety of concrete classes (FileDatabase, etc.), the connection between modules is minimised. This encourages the codebase to be more testable and maintainable.
- File operations are handled by the FileDatabase class; file paths and File objects are not directly handled by the Database class or other high-level modules. Rather, they abstract away the specifics of file processing and engage with the IDataAccess interface.

Class Diagram:



Applying Single Responsibility Principle

Before Code:

```
package Library;

public class Order {
    private Book book;
    private User user;
    private double price;
    private int qty;

    public Order(Book book, User user, double price, int qty);
    public Order();

    public Book getBook();
    public void setBook(Book book);

    public User getUser();
    public void setUser(User user);

    public double getPrice();
    public void setPrice(double price);

    public int getQty();
    public void setQty(int qty);

    public String toString();
    public String toString2();
}
```

Why does this code violate the Single Responsibility Principle?

The class 'Order' is responsible for managing details of an order (book, user, price, quantity), as well as providing string representation methods: 'toString()' and 'toString2()' . This violates the SRP because the class has multiple reasons to change:

- If the structure of order itself changes, then it must be modified.
- If the format or content of string representation of order changes, then the class must be modified.
- The class contains order-related data (book, user, price, quantity). It offers access to and modification of these data characteristics using getter and setter methods. Keeping an order in good condition is one of the responsibilities represented by this class component.

- Additionally, the class has methods like toString2() and toString(). These methods are in charge of translating the object state into representations in strings. Another duty represented by this part of the class is to provide string representations for the object.

Refactored Code:

```
package Library;

public class Order {
    private Book book;
    private User user;
    private double price;
    private int qty;

    public Order(Book book, User user, double price, int qty);

    public Order();

    public Book getBook();
    public void setBook(Book book);

    public User getUser();
    public void setUser(User user);

    public double getPrice();
    public void setPrice(double price);

    public int getQty();

    public void setQty(int qty);
}

public class OrderFormatter {
    public String formatOrder(Order order) {
        StringBuilder sb = new StringBuilder();
        sb.append("Order details:\n");
        sb.append("Book: ").append(order.getBook().getTitle()).append("\n");
        sb.append("User: ").append(order.getUser().getName()).append("\n");
        sb.append("Price: ").append(order.getPrice()).append("\n");
        sb.append("Quantity: ").append(order.getQty()).append("\n");
        return sb.toString();
    }
}
```

How refactored code is adhering to SRP?

- The ‘Order’ class is now responsible for managing details of an order while ‘OrderFormatter’ class is responsible for generating string representations of orders. This separation makes it adhere to the Single Responsibility Principle.
- The Order class is in charge of encapsulating an order's associated data (book, user, price, quantity). To see and edit this data, it has getter and setter methods. This class is dedicated to order state management only, in accordance with the SRP.
- The formatting of an Order object into a string representation is done by the OrderFormatter class. It has a single method, formatOrder(Order order), which builds a formatted string with order data from an Order object. Formatting orders is isolated from the Order class by transferring the formatting logic into a different class, which complies with the SRP.

Class Diagram: