Redesigning Poor Code

Your task is to reformat a poorly designed code for a Library Management System with the following functionalities

1. Manage books (add, update, delete, list, and search by various attributes like title, author, or ISBN)
2. Manage users (add, update, delete, list, and search by attributes like name, user ID)
3. Check out and check-in books
4. Track book availability
5. Simple logging of operations

Task for the Candidate

1. The application should utilize classes and objects more effectively, with clear relationships and responsibilities among them with Object Oriented Design.
2. Implement or refactor the storage.py for reliable storage and retrieval using file-based storage (JSON, CSV, etc.)
3. Obtain input from the user using CLI to access information from the user in a friendly and intuitive manner.
4. Implement error handling and input validation throughout the application.
5. The application's design doesn't facilitate easy extension or modification.
6. Ensure that the application is modular and scalable, allowing for future expansions such as new types of items to manage or additional features like due dates for books, late fees, etc.
7. Include documentation and comments to explain the design decisions, architecture, and usage of classes and methods.

Code

You can find the code to be worked on : here

```
.
├── book.py
├── check.py
├── main.py
├── models.py
├── storage.py
└── user.py
```

Judging Parameters:

1. **Object-Oriented Design**
    a. Class Usage
    Check if the candidate has effectively used classes to encapsulate related functionalities, such as Book, User, and management systems (BookManager, UserManager, etc.)
    b. Inheritance and Polymorphism
    Evaluate whether inheritance is used appropriately for shared behaviors and if polymorphism is applied to handle different types or behaviors dynamically.

2. **Encapsulation**
    a. Look for proper encapsulation of data and behaviors within classes, ensuring that access to data is controlled through methods rather than directly.
    b. Readability
    The code should be easy to read, with meaningful variable and method names, consistent indentation, and clear structure.
    c. Modularity
    The application should be well-structured into modules or packages, with a clear separation of concerns.
    d. Documentation
    Expect well-documented code, including docstrings for classes and methods, and comments explaining complex logic or decisions.
    e. Error Handling
    The candidate should have added robust error handling to manage exceptions gracefully and validate user inputs.

3. **Use of Pythonic Idioms and Features**
    a. Pythonic Code
    Look for the use of Pythonic idioms and constructs, such as list comprehensions, generator expressions, context managers, decorators, and the use of standard library modules where appropriate.

b. Efficiency
   The code should efficiently handle operations, avoiding unnecessary computations or data processing.

4. Design Patterns and Best Practices

a. Design Patterns
   Identify the use of design patterns, such as Factory for creating objects, Singleton for managing shared resources, or Command for decoupling operation execution from the invoker.
b. Best Practices
   The submission should follow best practices in software development, including the DRY (Don't Repeat Yourself) principle, SOLID principles for OOP design, and effective state management.

5. Testing and Validation

a. Unit Tests
   Check for the presence of unit tests that cover critical functionalities and edge cases, demonstrating the candidate's commitment to reliability and maintainability.
b. Input Validation
   Ensure that all user inputs are validated before processing to prevent errors or unexpected behavior.

6. Scalability and Maintainability

a. Scalability
   Evaluate how well the codebase is prepared to handle future requirements, such as adding new features or managing a large number of users or books.
b. Maintainability
   The code should be structured and written in a way that makes it easy to modify, extend, or refactor parts of the system without affecting others.

7. User Interface and Experience

a. CLI Design

If the application includes a command-line interface, assess how user-friendly and intuitive it is. Check for helpful prompts, clear error messages, and a logical flow of operations.