Report

- *<Singelton pattern>*

  For our project, we have chosen the Singelton design pattern, as it provides shared access to the database from different parts of the program and guarantees that there's just one instance of a class (Nothing, except for the Singleton class itself, can replace the cached instance.)

- *<Singelton implementation>*

```
A class representing Database which stores books
final class DataBase {
    Singleton object
    private static DataBase instance;

    A Hashmap which stores an array of books by genre
    final HashMap<Genre, ArrayList<Book>> genre_of_books;

    Creates a database using the genres hashmap
    Params: genres – hashmap storing an array of book by genre
    private DataBase(HashMap<Genre, ArrayList<Book>> genres) { this.genre_of_books = genres; }

    Params: genres – hashmap An alternative to the constructor and is the access point to an instance of Database
            class.
    public static DataBase getInstance(HashMap<Genre, ArrayList<Book>> genres) {
        if (instance == null)
            instance = new DataBase(genres);
        return instance;
    }
```

- *<OOP principles>*

  The main principles of OOP used by us :
  - ***abstraction*** for a contextual understanding of the subject, formalized as a class. it "shows" only the necessary attributes and "hides" unnecessary information;
  - ***encapsulation*** for the fast and secure organization of hierarchical manageability proper: so that a simple "what to do" command is enough, without simultaneously specifying exactly how to do it since this is already another level of management;
  - inheritance and polymorphism were not required in our problem;

- *<User interaction with the system>*

  The user interacts with the system only through the console (there is no need to change anything manually in the code). The actions that the user can perform are written on the screen.

```
Welcome to the "Booka" Bookstore website.
What do you want to do?
{(0)return book; (1)see books; (-)exit}
```

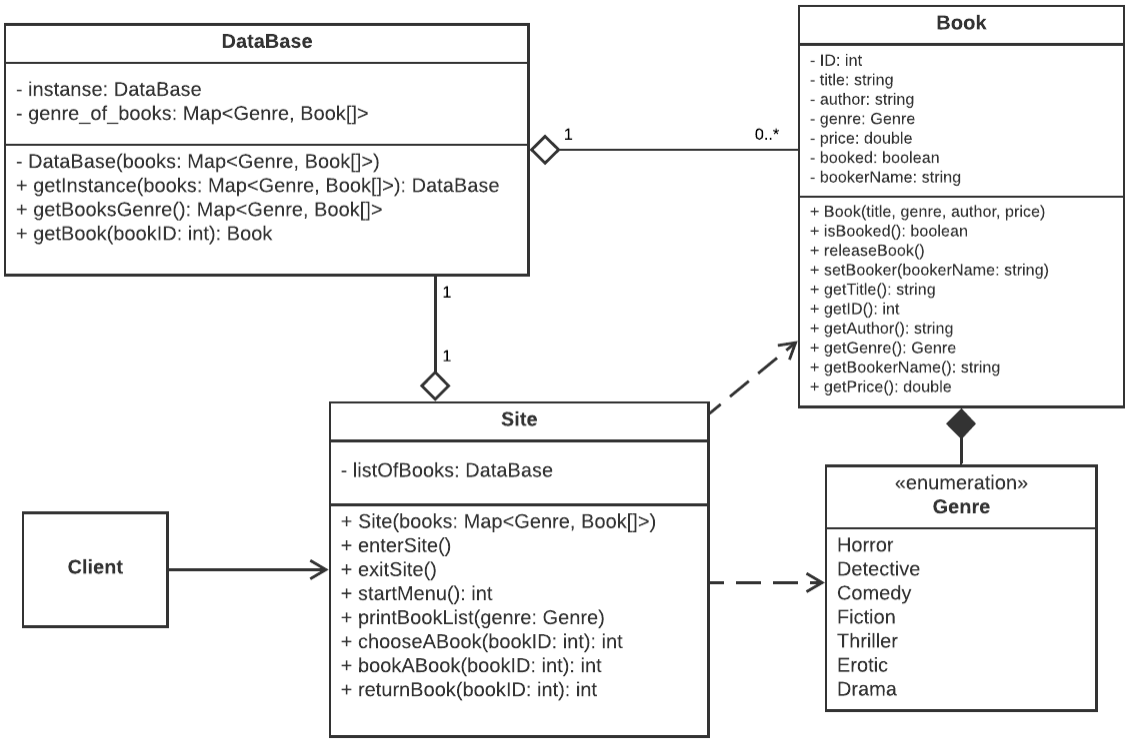
- *<Example of Javadoc documentation standard in code comments>*

```java
/**
 * Prints out a list of variants to the user and returns the answer
 *
 * @param var list of variants
 * @return user response
 */
private int AskUser(String[] var) {
    printVariantsList(var);
    return calculateInput(getInput(), var);
}

/**
 * @return the text printed by the user
 */
```

- *<uml class diagram>*

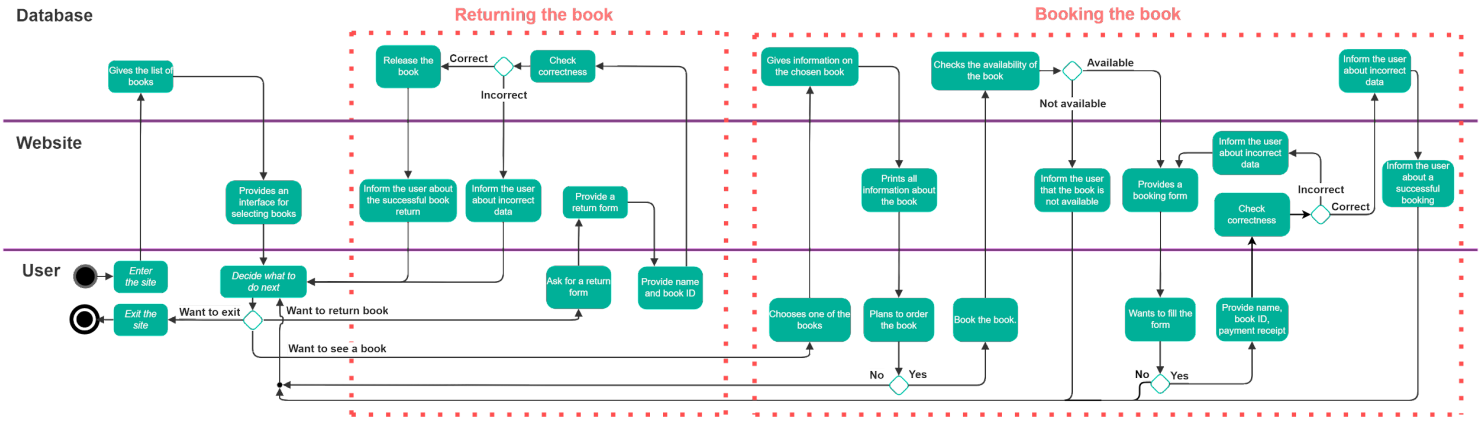
The class diagram illustrates the basic structure of the project. The Site class provides an interface for browsing, booking, and returning books. Site contains a Database inside, which is responsible for storing books by genre and guaranteeing a single copy of the data. The Book class stores all information about itself and booking data.



https://lucid.app/documents/view/0748af22-9d68-4c76-bc15-e54920e698c6

- *<uml activity diagram>*

To clearly show the behavior of the system and the order and flow of actions in the system, we have built a UML activity diagram. The starting point system will show the list of books and provide an interface for its selection. After that, we have to choose which action to do: returning or booking a book. These two actions contain many action states that represent the processing of the request by the system, as shown in the diagram. Action states are connected with transitions that are needed to show the sequence of steps that the system performs.



https://drive.google.com/file/d/13HDOYpsdjqaUF4wc67ez6Lr4-G0zzhTV/view?usp=sharing