Laboratory Work #1 - Creational Design Patterns

Author: Daniela Vornic, FAF-222

Objectives

- Get familiar with the Creational DPs;
- Choose a specific domain;
- Implement at least 3 CDPs for the specific domain.

Theory

Creational Design Patterns are advanced object creation mechanisms that increase flexibility and reuse of existing code. They provide solutions to instantiate objects in the most suitable way for specific situations. The main creational patterns include:

- **Singleton Pattern**: Ensures a class has only one instance and provides a global point of access to it.
- **Factory Method Pattern**: Defines an interface for creating objects but lets subclasses decide which class to instantiate.
- **Abstract Factory Pattern**: Provides an interface for creating families of related objects without specifying their concrete classes.
- **Builder Pattern**: Separates the construction of a complex object from its representation.
- **Prototype Pattern**: Creates new objects by cloning an existing object, known as the prototype.

Domain of the application

The application is a Library Management System that allows librarians to:

- Add and manage books (fiction and non-fiction)
- Register borrowers
- Handle book checkouts and returns
- Track due dates and late fees
- Create copies of existing books
- Search for books

The system operates through a console interface and implements proper validation and error handling.

Used Design Patterns

Singleton Pattern

Purpose: Ensure a single instance of the library database throughout the application.

Implementation: The `LibraryDatabase` class implements the Singleton pattern to maintain a centralized data store for books and borrowers. The class has a private constructor and a static method `getInstance()` that returns the single instance of the class. The instance is created lazily and is thread-safe, along with the data structures used to store books and borrowers.

```java
public class LibraryDatabase implements ILibraryDatabase {
  private static volatile ILibraryDatabase instance;
  private final List<Book> books;
  private final List<Borrower> borrowers;

  private LibraryDatabase() {
    books = Collections.synchronizedList(new ArrayList<>());
    borrowers = Collections.synchronizedList(new ArrayList<>());
  }

  public static ILibraryDatabase getInstance() {
    ILibraryDatabase result = instance;
    if (result != null) {
      return result;
    }
    synchronized (LibraryDatabase.class) {
      if (instance == null) {
        instance = new LibraryDatabase();
      }
      return instance;
    }
  }

  // ...
}
```

Factory Method Pattern

Purpose: Create different types of books (Fiction/Non-Fiction) while encapsulating the creation logic.

Implementation: The pattern is implemented through an abstract `BookCreator` class and concrete creators for each book type (`FictionBookSection` and `NonFictionBookSection`). The `BookCreator` class declares an abstract method `createBook()` that is implemented by the concrete creators to create specific book instances.

```java
// Abstract Creator
public abstract class BookCreator {
    public abstract Book createBook(String title, String author, String isbn, int year);
}

// Concrete Creator for Fiction Books
public class FictionSection extends BookCreator {
    @Override
    public Book createBook(String title, String author, String isbn, int year) {
        return new FictionBook(title, author, isbn, year);
    }
}
```

```
// Concrete Creator for Non-Fiction Books
public class NonFictionSection extends BookCreator {
    @Override
    public Book createBook(String title, String author, String isbn, int
year) {
        return new NonFictionBook(title, author, isbn, year);
    }
}
```

```
// Usage in BookService
public class BookService {
    private final Map<BookType, BookCreator> factories;

    private void initializeFactories() {
        factories.put(BookType.FICTION, new FictionSection());
        factories.put(BookType.NON_FICTION, new NonFictionSection());
    }
}
```

Prototype Pattern

Purpose: Create copies of existing books while maintaining their type and basic properties.

Implementation: Implemented through the `clone()` method in the `Book` hierarchy, allowing creation of book copies without knowing their specific types.

```
public abstract class Book implements Cloneable {
    @Override
    public Book clone() {
        try {
            Book clonedBook = (Book) super.clone();
            clonedBook.checkedOut = false;
            clonedBook.dueDate = null;
            clonedBook.borrower = null;
            return clonedBook;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Error cloning book", e);
        }
    }
}
```

```
public Book createBookCopy(String existingIsbn, String newIsbn, int year) {
    Book existingBook = database.findBookByIsbn(existingIsbn);
    if (existingBook == null) {
        throw new LibraryException("Original book not found");
    }
    Book newBook = existingBook.clone();
```

```
        newBook.setISBN(newIsbn);
        newBook.setYear(year);
        database.addBook(newBook);
        return newBook;
    }
```

Implementation

The project structure is organized into several packages:

```
project/
├── client/
│   ├── LibraryInitializer.java
│   └── LibraryManagementSystem.java
├── domain/
│   ├── database/
│   │   ├── ILibraryDatabase.java
│   │   └── LibraryDatabase.java
│   ├── factory/
│   │   ├── BookCreator.java
│   │   ├── BookType.java
│   │   ├── FictionSection.java
│   │   └── NonFictionSection.java
│   └── models/
│       ├── Book.java
│       ├── FictionBook.java
│       ├── NonFictionBook.java
│       └── Borrower.java
├── service/
│   ├── BookService.java
│   └── BorrowerService.java
│   └── IBookService.java
│   └── IBorrowerService.java
├── ui/
│   ├── ConsoleUI.java
│   └── InputHandler.java
└── util/
    ├── LibraryException.java
    └── ValidationUtils.java
```

Key Components:

1. **Book Hierarchy** The abstract `Book` class serves as the base for all book types, such that `FictionBook` and `NonFictionBook` extend it with specific behaviors. It successfully implements the Prototype pattern by overriding the `clone()` method to create copies of existing books.

2. **Service Layer** `BookService` handles book-related operations while `BorrowerService` manages borrower operations. Services use dependency injection for better flexibility.

```
public class BookService implements IBookService {
    private final ILibraryDatabase database;
    private final Map<BookType, BookCreator> factories;

    public BookService(ILibraryDatabase database, Map<BookType,
BookCreator> factories) {
        this.database = database;
        this.factories = factories;
    }
    // ... service methods
}
```

As an example, this is the interface for the BookService:

```
public interface IBookService {
  Book addBook(String title, String author, String isbn, int year, BookType
type);
  Book checkoutBook(String isbn, String borrowerId, int loanPeriodDays);
  Book returnBook(String isbn);
  List<Book> searchBooks(String searchTerm);
  Book findBookByIsbn(String isbn);
  List<Book> getAllBooks();
  Book createBookCopy(String existingIsbn, String newIsbn, int year);
}
```

3. **User Interface**

It is a console-based UI with clear menu options. The inputs are validated and the errors are handled gracefully.

```
public class ConsoleUI {
    private final IBookService bookService;
    private final IBorrowerService borrowerService;

    public void displayMenu() {
        System.out.println("\n=== Library Management System ===");
        System.out.println("1. Add Book");
        System.out.println("2. Search Book");
        // ... other options
    }
    // ... UI methods
}
```

```
// Example of input validation
public static int readInt(String prompt, int min, int max) {
  System.out.print(prompt);
```

```
    try {
      int value = Integer.parseInt(scanner.nextLine().trim());
      if (value < min || value > max) {
        throw new LibraryException(
            String.format("Value must be between %d and %d", min, max));
      }
      return value;
    } catch (NumberFormatException e) {
      throw new LibraryException("Please enter a valid number");
    }
  }
```

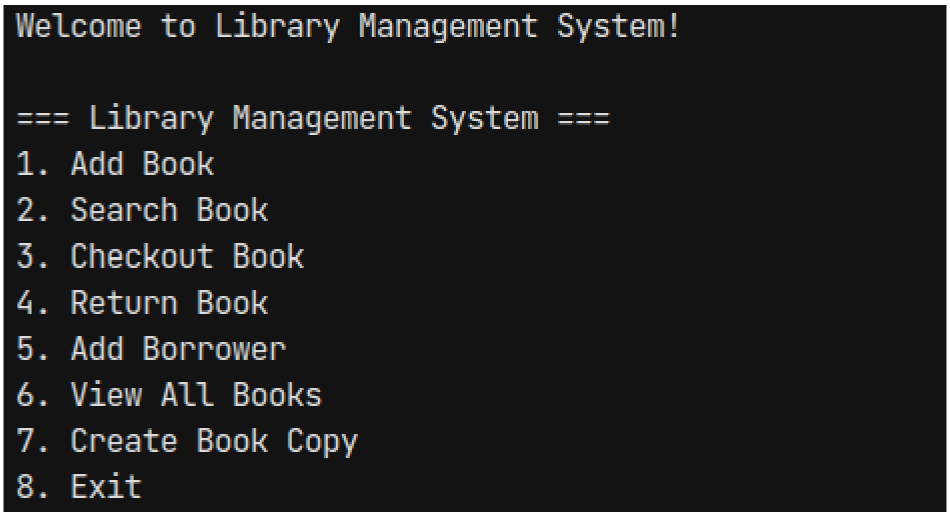
```
public class LibraryException extends RuntimeException {
  public LibraryException(String message) {
    super(message);
  }
}
```

4. **Database Layer**

The `LibraryDatabase` class implements the Singleton pattern to ensure a single instance of the database throughout the application. It uses synchronized collections to store books and borrowers, as shown in the snippet from the Singleton Pattern section.

Screenshots / Results

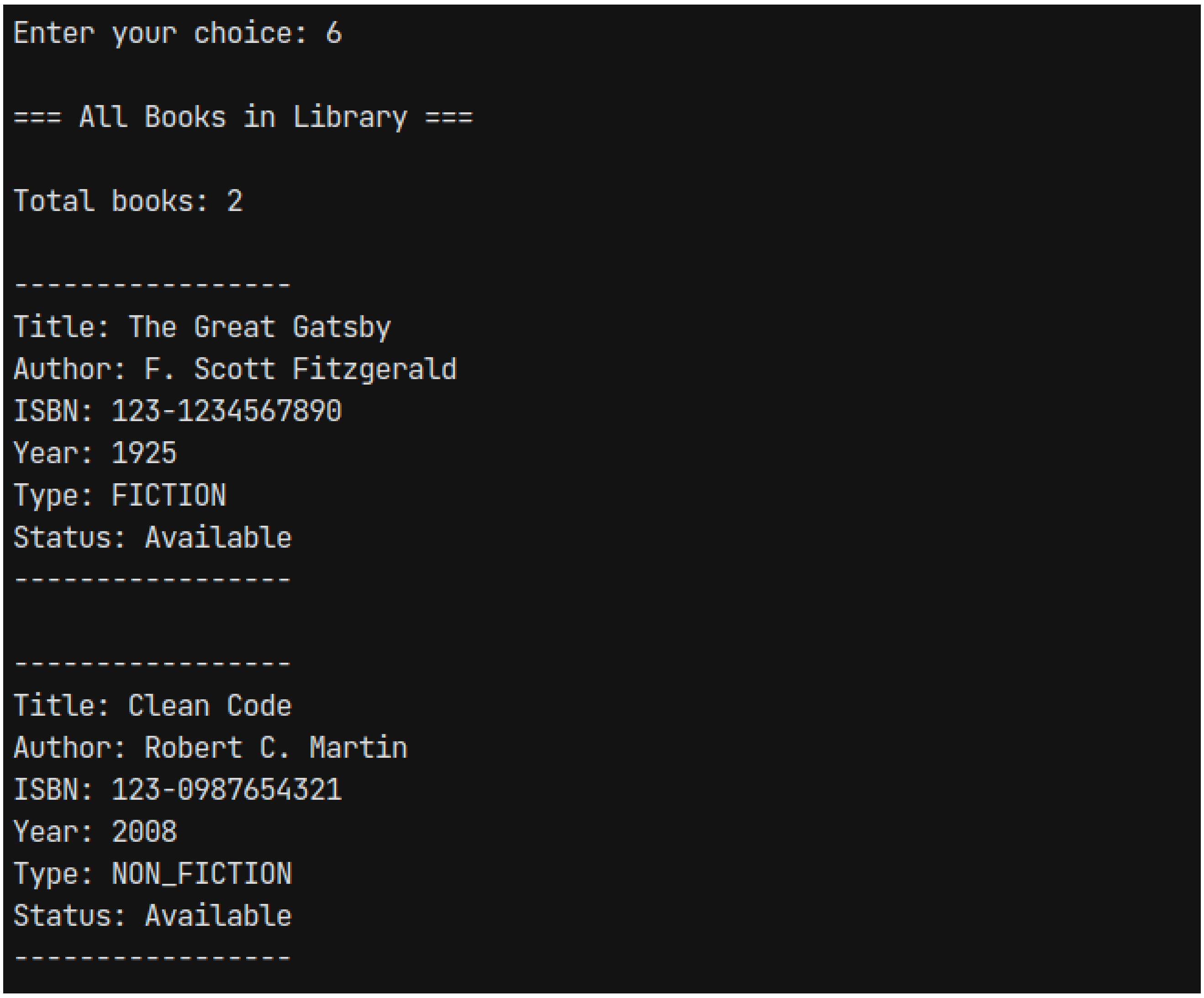
In Figure 1, the main menu of the Library Management System is displayed, showing the available options for librarians. The system allows adding books, searching for books, registering borrowers, and handling book checkouts and returns.



```
Welcome to Library Management System!

=== Library Management System ===
1. Add Book
2. Search Book
3. Checkout Book
4. Return Book
5. Add Borrower
6. View All Books
7. Create Book Copy
8. Exit
```

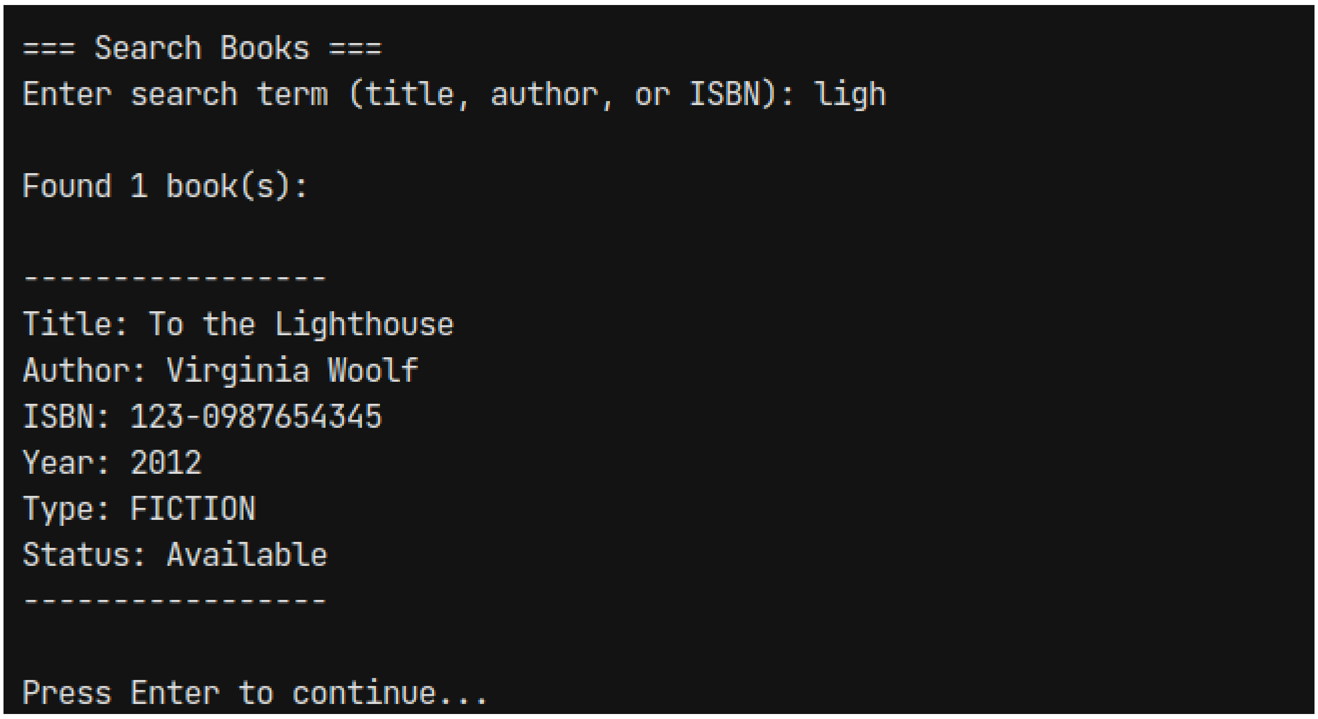
Figure 1: Main menu of the Library Management System

From the database perspective, the system maintains a list of books and borrowers, as shown in Figure 2. The librarians can view all books in the library, add new books, and create copies of existing books. The system also allows searching for books by title, as shown in Figure 4.



```
Enter your choice: 6

=== All Books in Library ===

Total books: 2

-----------------
Title: The Great Gatsby
Author: F. Scott Fitzgerald
ISBN: 123-1234567890
Year: 1925
Type: FICTION
Status: Available
-----------------


-----------------
Title: Clean Code
Author: Robert C. Martin
ISBN: 123-0987654321
Year: 2008
Type: NON_FICTION
Status: Available
-----------------
```

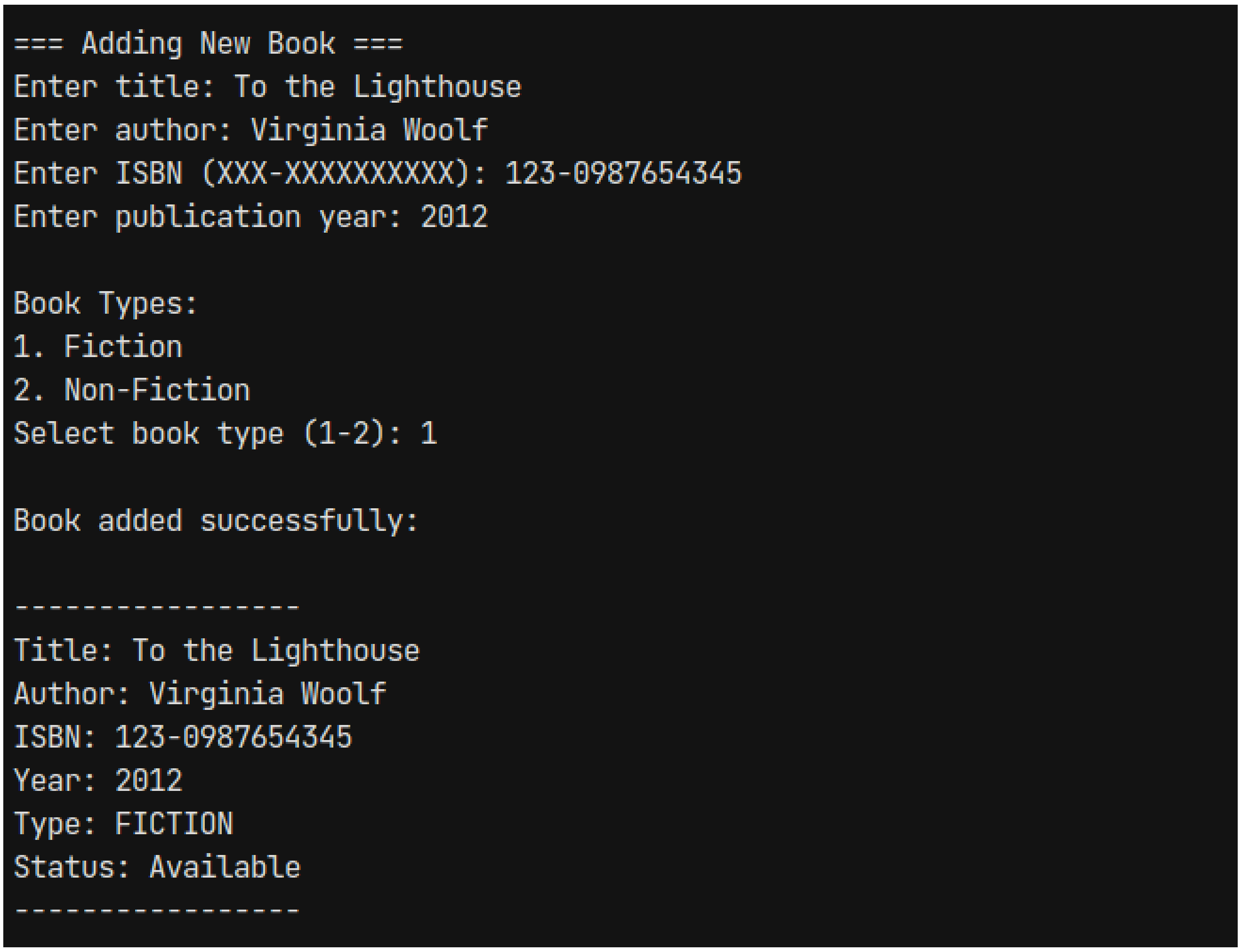
Figure 2: Viewing all books in the library



```
=== Search Books ===
Enter search term (title, author, or ISBN): ligh

Found 1 book(s):

-----------------
Title: To the Lighthouse
Author: Virginia Woolf
ISBN: 123-0987654345
Year: 2012
Type: FICTION
Status: Available
-----------------

Press Enter to continue...
```

Figure 3: Searching for a book by title

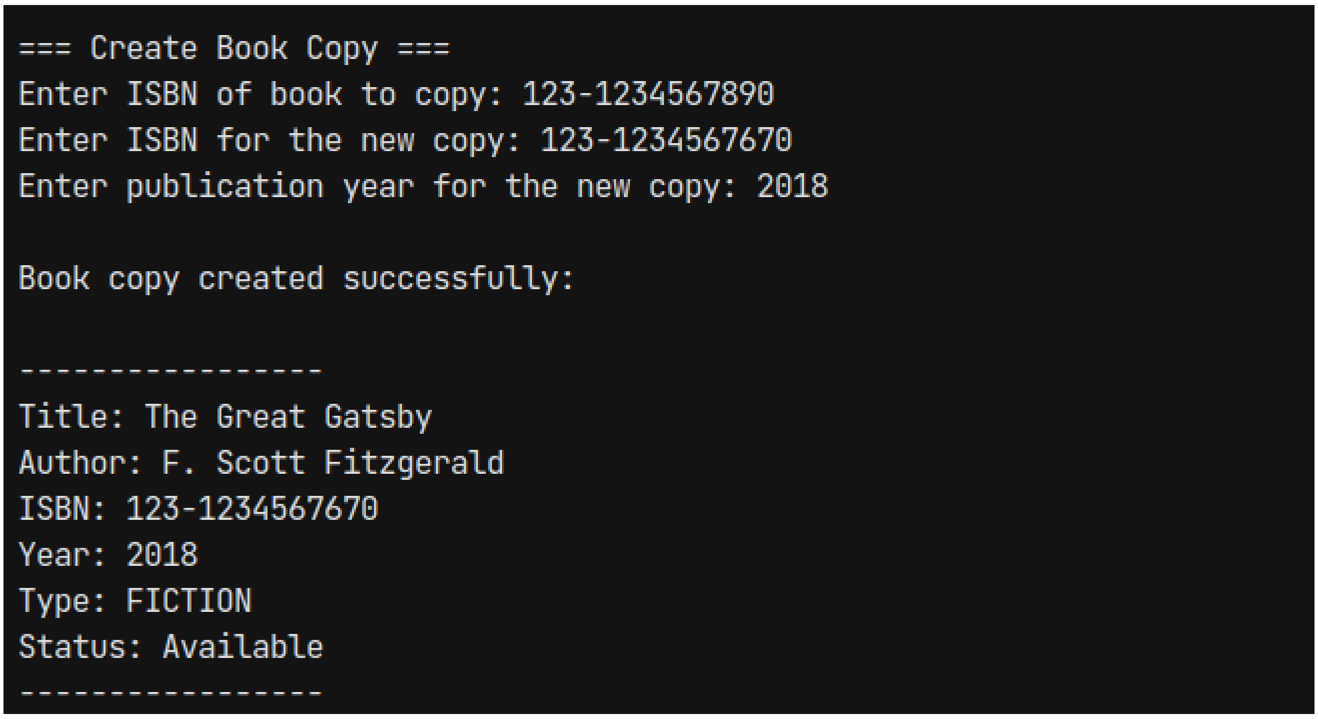
Using the Factory Method pattern, the system can create different types of books (Fiction and Non-Fiction) based on the user's input. Figure 4 shows the process of adding a new book to the library, where the librarian can choose the book type.



```
=== Adding New Book ===
Enter title: To the Lighthouse
Enter author: Virginia Woolf
Enter ISBN (XXX-XXXXXXXXXX): 123-0987654345
Enter publication year: 2012

Book Types:
1. Fiction
2. Non-Fiction
Select book type (1-2): 1

Book added successfully:

-----------------
Title: To the Lighthouse
Author: Virginia Woolf
ISBN: 123-0987654345
Year: 2012
Type: FICTION
Status: Available
-----------------
```

Figure 4: Adding a new book to the library

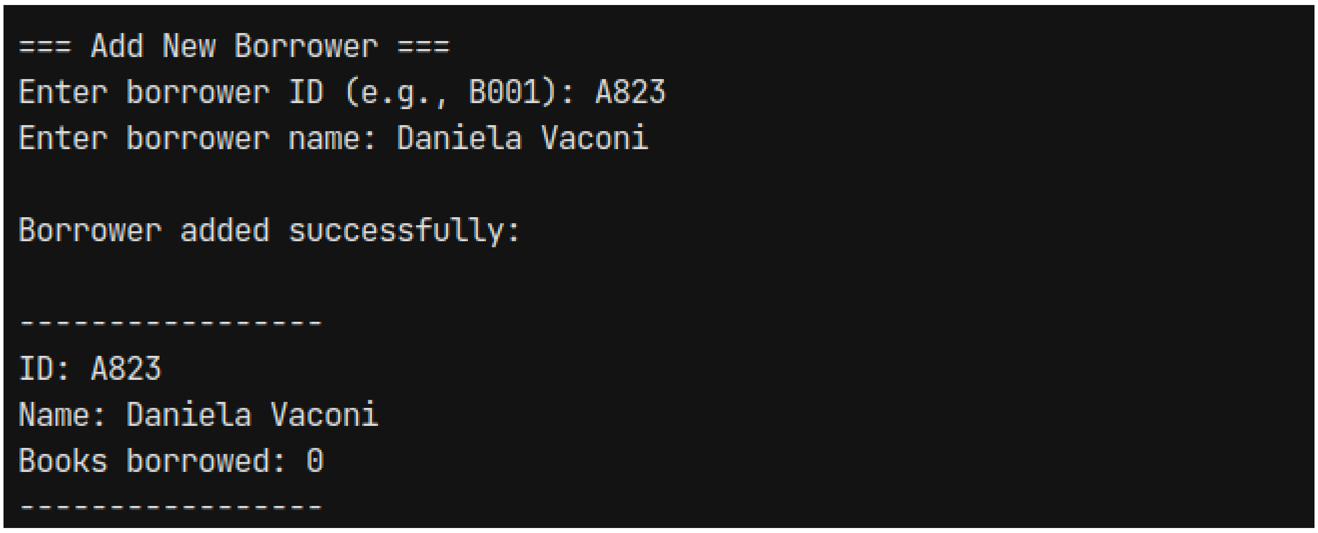
Using the Prototype pattern, the system can create copies of existing books while maintaining their type and basic properties. Figure 5 shows the process of creating a copy of an existing book by providing a new ISBN and publication year.



```
=== Create Book Copy ===
Enter ISBN of book to copy: 123-1234567890
Enter ISBN for the new copy: 123-1234567670
Enter publication year for the new copy: 2018

Book copy created successfully:


-----------------
Title: The Great Gatsby
Author: F. Scott Fitzgerald
ISBN: 123-1234567670
Year: 2018
Type: FICTION
Status: Available
-----------------
```

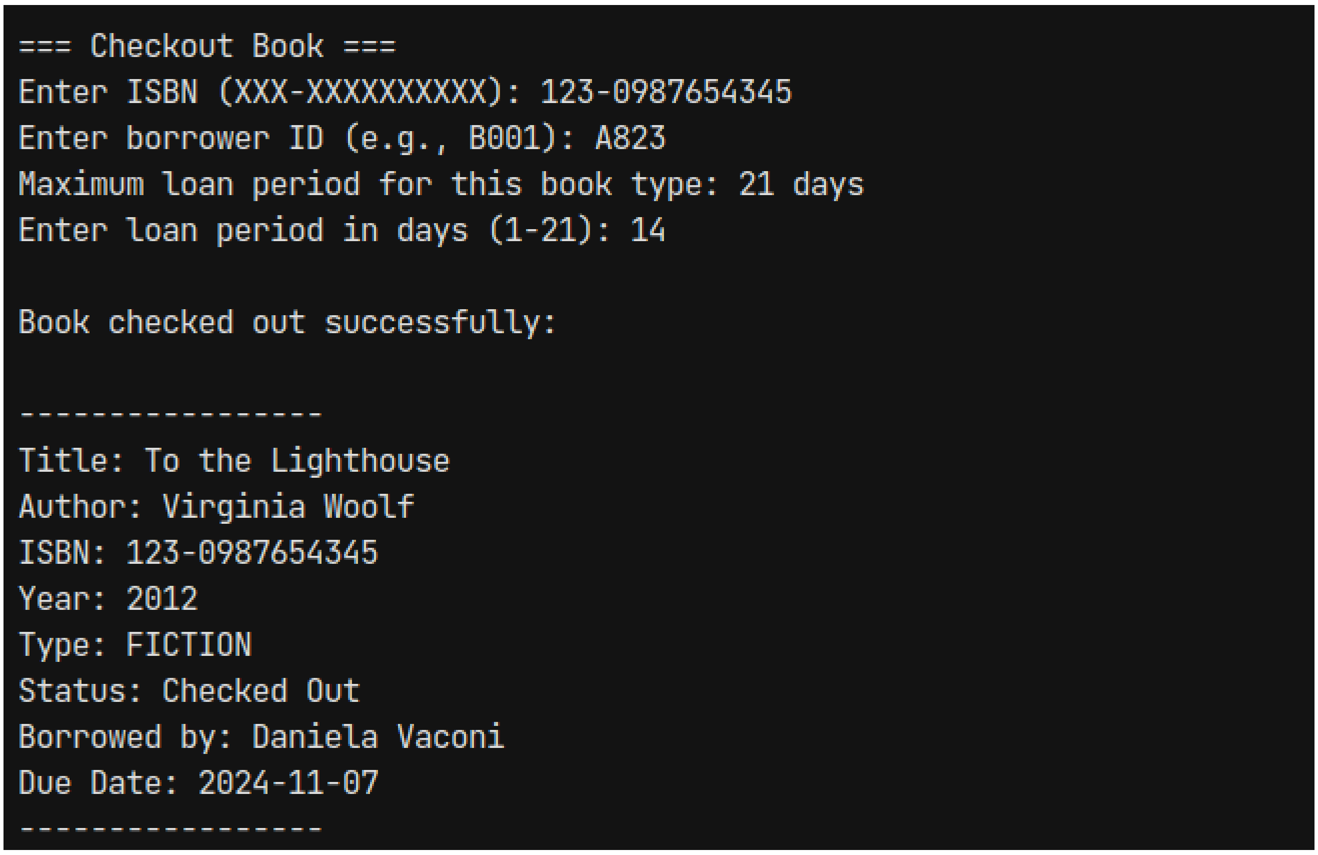
Figure 5: Creating a copy of an existing book

The system also allows librarians to register new borrowers and handle book checkouts and returns. Figure 6 shows the process of adding a new borrower to the system, while Figure 7 and Figure 8 demonstrate checking out and returning a book to the library, respectively.



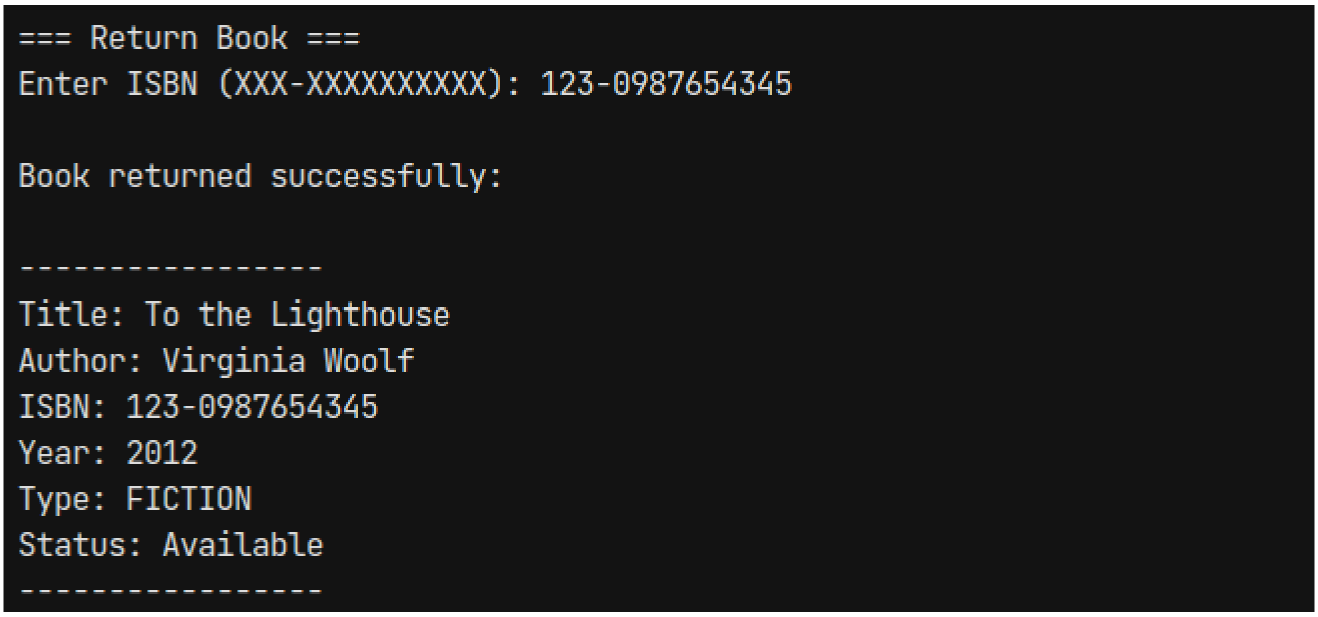
```
=== Add New Borrower ===
Enter borrower ID (e.g., B001): A823
Enter borrower name: Daniela Vaconi

Borrower added successfully:


-----------------
ID: A823
Name: Daniela Vaconi
Books borrowed: 0
-----------------
```

Figure 6: Registering a new borrower



```
=== Checkout Book ===
Enter ISBN (XXX-XXXXXXXXXX): 123-0987654345
Enter borrower ID (e.g., B001): A823
Maximum loan period for this book type: 21 days
Enter loan period in days (1-21): 14

Book checked out successfully:

-----------------
Title: To the Lighthouse
Author: Virginia Woolf
ISBN: 123-0987654345
Year: 2012
Type: FICTION
Status: Checked Out
Borrowed by: Daniela Vaconi
Due Date: 2024-11-07
-----------------
```

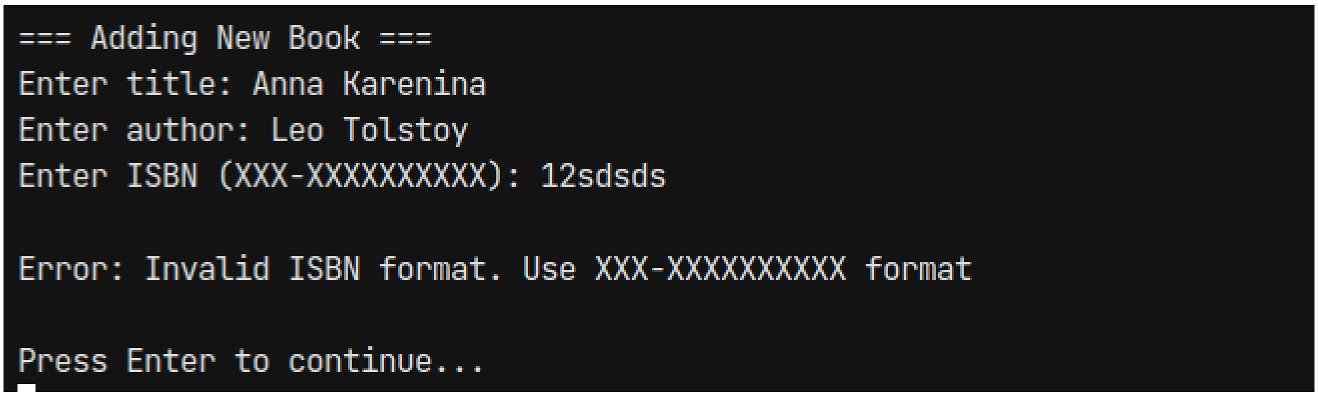
Figure 7: Checking out a book to a borrower



```
=== Return Book ===
Enter ISBN (XXX-XXXXXXXXXX): 123-0987654345

Book returned successfully:

-----------------
Title: To the Lighthouse
Author: Virginia Woolf
ISBN: 123-0987654345
Year: 2012
Type: FICTION
Status: Available
-----------------
```

Figure 8: Returning a book to the library

Last but not least, the system provides proper validation and error handling for user inputs. Figure 9 shows an example of handling a validation error when the user enters an invalid number for the ISBN of a book.



```
=== Adding New Book ===
Enter title: Anna Karenina
Enter author: Leo Tolstoy
Enter ISBN (XXX-XXXXXXXXXX): 12sdsds

Error: Invalid ISBN format. Use XXX-XXXXXXXXXX format

Press Enter to continue...
```

Figure 9: Handling validation errors in the UI

Conclusions

This laboratory work demonstrated the practical application of creational design patterns in a real-world scenario. The combination of these patterns not only improved code organization but also enhanced maintainability. By properly separating concerns and following SOLID principles, the codebase remained clean and accessible, making it easier to implement new features or modify existing ones.