BookResource - God Class

Link to chat

Chat 1: Before being told that it was Missing Modularisation & Hierarchy & God Class:

There are a few design smells in the provided BookResource class:

Violation of Single Responsibility Principle (SRP): The class is responsible for handling various aspects such as creating, updating, deleting books, managing book covers, listing books, etc. It's advisable to split these responsibilities into separate classes.

Large Method: Some methods are quite large and handle multiple concerns. For instance, the add, add (manual), update, and list methods perform various operations that can be broken down into smaller, more focused methods.

Code Duplication: There is some repetition of code for error handling, validation, and database operations.

Hard-Coded Values: Some hard-coded values like path names and constants can be extracted and defined as constants or configurable parameters.

To refactor the code, we can follow these steps:

Split Responsibilities: Separate concerns such as CRUD operations, handling book covers, listing books, etc., into separate classes.

Extract Methods: Break down large methods into smaller, focused methods to adhere to the Single Responsibility Principle and improve readability.

Remove Code Duplication: Identify and eliminate redundant code by extracting common functionality into separate methods or utility classes.

Use Constants: Replace hard-coded values with constants or configurable parameters to improve maintainability.

Here's a refactored version of the BookResource class, focusing on addressing these design smells:

```
package com.sismics.books.rest.resource;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/book")
public class BookResource extends BaseResource {
    private final BookService bookService;

    public BookResource(BookService bookService) {
        this.bookService = bookService;
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response addBook(@FormParam("isbn") String isbn) {
        return bookService.addBook(isbn);
    }

    @DELETE
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteBook(@PathParam("id") String userBookId) {
        return bookService.deleteBook(userBookId);
    }

    @PUT
    @Path("manual")
    @Produces(MediaType.APPLICATION_JSON)
    public Response addManualBook(BookRequest bookRequest) {
        return bookService.addManualBook(bookRequest);
    }

    // Other methods...

    // Define other methods for updating, listing, and managing books
}
```

In this refactored version, I've introduced a BookService class responsible for handling book-related operations. Each method in the BookResource class delegates the actual business logic to the corresponding method in the BookService class. This separation of concerns improves readability, maintainability, and testability of the code.

> Chatgpt recognized the design smells almost perfectly and it was even also able to recognize that the CRUD functionalities should be handled separatly. It was able to remove the business logic from the resouce class and put it in a separate class similar to what we have done, however in the refactored code, it didn't create multiple classes to handle the different types of logics and created a single class for handling all the requests.

Chat 2: After being told that it was Missing Modularisation & Hierarchy & God Class:

To refactor the code to address the design smell of Missing Modularisation & Hierarchy & God Class, we'll introduce a more modular and hierarchical structure by separating concerns into multiple classes and establishing a clear hierarchy of responsibilities. We'll create separate classes for handling CRUD operations, managing book covers, and listing books. Additionally, we'll introduce interfaces to enforce a clear contract between classes.

Here's a refactored version of the code:

```
@Path("/book")
public class BookResource {
    private final BookService bookService;

    public BookResource(BookService bookService) {
        this.bookService = bookService;
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response addBook(@FormParam("isbn") String isbn) {
        return bookService.addBook(isbn);
    }

    @DELETE
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteBook(@PathParam("id") String userBookId) {
        return bookService.deleteBook(userBookId);
    }

    @PUT
    @Path("manual")
    @Produces(MediaType.APPLICATION_JSON)
    public Response addManualBook(BookRequest bookRequest) {
        return bookService.addManualBook(bookRequest);
    }

    // Other methods...

    // Define other methods for updating, listing, and managing books
}
```

```
public interface BookService {
    Response addBook(String isbn);
    Response deleteBook(String userBookId);
    Response addManualBook(BookRequest bookRequest);
    // Other methods...
}
```

```
public class BookServiceImpl implements BookService {
    private final BookManager bookManager;

    public BookServiceImpl(BookManager bookManager) {
        this.bookManager = bookManager;
    }

    @Override
    public Response addBook(String isbn) {
        // Implement logic to add a book
    }

    @Override
    public Response deleteBook(String userBookId) {
        // Implement logic to delete a book
    }

    @Override
    public Response addManualBook(BookRequest bookRequest) {
        // Implement logic to add a manual book
    }

    // Other methods...
}
```

```
public interface BookManager {
    void createBook(Book book);
    void deleteBook(String bookId);
    Book getByIsbn(String isbn);
    // Other methods...
}
```

```
public class BookManagerImpl implements BookManager {
    @Override
    public void createBook(Book book) {
        // Implement logic to create a book
    }

    @Override
    public void deleteBook(String bookId) {
        // Implement logic to delete a book
    }

    @Override
    public Book getByIsbn(String isbn) {
        // Implement logic to get a book by ISBN
        return null;
    }

    // Other methods...
}
```

```
public class Book {
    // Define properties and methods related to the book entity
}
```

This refactored structure enhances modularity and hierarchy by organizing the code into smaller, focused classes with clear responsibilities. It also helps in better code maintenance, scalability, and testability.

> After being told the design smell, it did refactoring similar to what it already did, however it introduced unnecessary abstraction by creating interfaces for what are essentially helper classes. As such, after telling the design smell, gpt was over egregious in its refactoring and produced code which although more modular and hierarchical, is also needlessly more complex.