Object Oriented Analysis and Design with Java

UE19CS353

Prof.J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY ONLINE

UE19CS353: Object Oriented Analysis and Design with Java

OO Design Principles and Sample Implementation of Patterns in Java

Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

PES UNIVERSITY ONLINE

UE19CS353: Object Oriented Analysis and Design with Java

OO Design Principles and Sample Implementation of Patterns in Java

SOLID: Single Responsibility, Open-Closed principle

Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

Important Aspects of Bad design

Rigidity

- the impact of a change is unpredictable
- every change causes a cascade of changes in dependent modules
- costs become unpredictable

Fragility

- the software tends to break in many places on every change
- the breakage occurs in areas with no conceptual relationship
- on every fix the software breaks in unexpected ways

Immobility

- it's almost impossible to reuse interesting parts of the software
- the useful modules have too many dependencies
- the cost of rewriting is less compared to the risk faced to separate those parts

Viscosity

- a hack is cheaper to implement than the solution within the design
- preserving design moves are difficult to think and to implement
- it's much easier to do the wrong thing rather than the right one

Good Design

What's the reason why a design becom
rigid, fragile, immobile, and viscous?

Improper dependencies between modules

Characteristics of a good design

High cohesion

Low coupling

How can we achieve a good design?

SOLID

The SOLID principles were introduced by **Robert J. Martin**.
It leads to more flexible and stable software architecture that's easier to maintain and extend, and less likely to break

SOLID principles

SOLID acronym:

S- **S**ingle Responsibility Principle (SRP)

O- **O**pen-Closed Principle (OCP)

L- The **L**iskov Substitution Principle (LSP)

I- The **I**nterface Segregation Principle (ISP)

D- The **D**ependency Inversion Principle (DIP)

- SOLID principles are software design coding standards
- SOLID principles help us to obtain good software with low coupling and high cohesion.
- SOLID principles is applied to reduce dependencies - changes in one part of software will not impacting others.

Single Responsibility Principle(SRP)

Object Oriented Analysis and Design with Java

Single Responsibility Principle(SRP)

A class should have one, and only one, reason to change -Robert C. Martin

- Each class only does one thing.
- Every class or module only has responsibility for one part of the software's functionality.
- Ensure low coupling code.
- Ensure easier coding to understand and maintain.
- It can be applied to classes, software components, and microservices.
- Utilizing this principle makes code easier to test and maintain, it makes software easier to implement, and it helps to avoid unanticipated side-effects of future changes.

Example Code Violates SRP

Assume that we have a Java application for a Book store. Create a Book class that lets users get and set the titles and authors of each book, and search the book in the inventory

```
class Book {
        String title;
        String author;
        String getTitle() {
                return title;
        }
        void setTitle(String title) {
                this.title = title;
        }
        String getAuthor() {
                return author;
        }
        void setAuthor(String author) {
                this.author = author;
        }
        void searchBook() {...}
}
```

Violation of SRP

The code violates the Single Responsibility Principle, as the Book class has two responsibilities.

First, it sets the data related to the books (title and author).

Second, it searches for the book in the inventory.

The setter methods change the Book object, which might cause problems when we want to search the same book in the inventory.

Solution

- Identify things that are changing for different reasons
- Group together things that change for the same reason
- Decouple the responsibilities.

In the refactored code, the Book class will only be responsible for getting and setting the data of the Book object.

Create another class called InventoryView that will be responsible for checking the inventory.

Move the searchBook() method to the new class and reference the Book class in the constructor.

Good Design

```
class Book {
	String title;
	String author;

	String getTitle() {
		return title;
	}
	void setTitle(String title) {
		this.title = title;
	}
	String getAuthor() {
		return author;
	}
	void setAuthor(String author) {
		this.author = author;
	}
}
```

Good Design

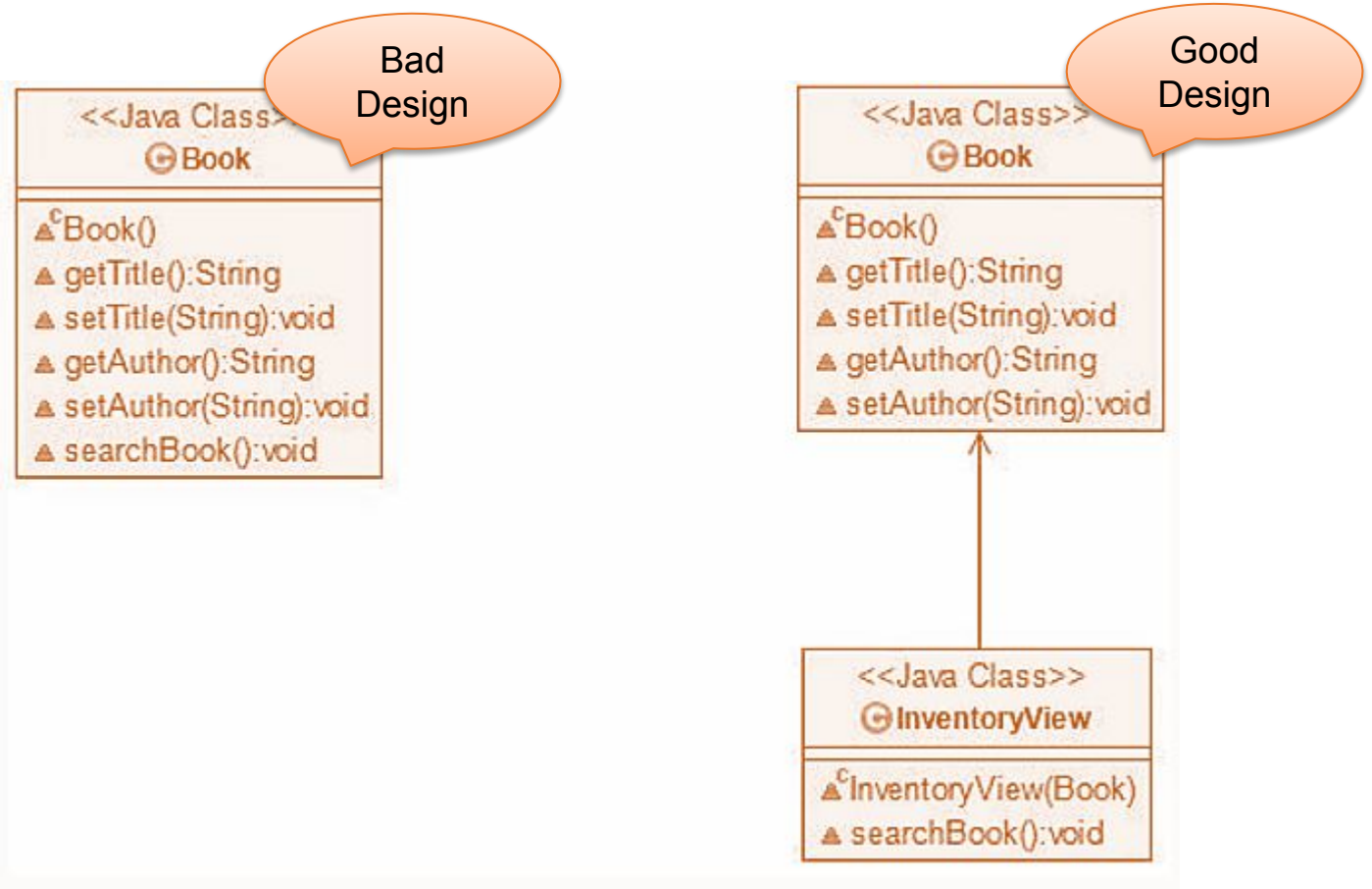
```
class InventoryView {

  Book book;

      InventoryView(Book book) {
            this.book = book;
      }

      void searchBook() {...}

}
```

Object Oriented Analysis and Design with Java

UML Class Diagram

UML diagram shows how the architecture changed after refactoring the code following Single Responsibility Principle.

Initial Book class that had two responsibilities is split into two classes, each having its own single responsibility.



Open-Closed Principle (OCP)

The Open-Closed Principle states that classes, modules, microservices, and other code units should be open for extension but closed for modification.

- A software entity must be easily extensible with new features without having to modify its existing code in use.
- We should be able to extend the existing code using OOP features like inheritance via subclasses and interfaces.
- Never modify classes, interfaces, and other code units that already exist, as it can lead to unexpected behavior.
- While adding a new feature extend the code rather than modifying it, so that you minimize the risk of failure as much as possible.

- Let’s consider the book store example.
- The store wants to hand out cookbooks at a discount price before Christmas.
- Based on the Single Responsibility Principle, Create two separate classes: CookbookDiscount to hold the details of the discount and DiscountManager to apply the discount to the price.

Example Code violates OCP


```
class CookbookDiscount {
    String getCookbookDiscount() {

        String discount = "30% between Dec 1 and 24.";
        return discount;
    }
}

class DiscountManager {
    void processCookbookDiscount(CookbookDiscount discount) {...}
}
```

Example Code violates OCP

This code works fine until the store management informs us that their cookbook discount sales were so successful that they want to extend it. Now, they want to hand out every biography with a 50% discount on the subject's birthday. To add the new feature, create a new BiographyDiscount class

```
class BiographyDiscount {

    String getBiographyDiscount() {

        String discount = "50% on the subject's birthday.";

        return discount;

    }
}
```

Violates Open-closed principle

To process the new type of discount, we need to add the new functionality to the DiscountManager class, too:

```
class DiscountManager {

    void processCookbookDiscount(CookbookDiscount discount) {...}

    void processBiographyDiscount(BiographyDiscount discount) {...}

}
```

As we changed existing functionality, we violated the Open/Closed Principle. Although the above code works properly, it might add new vulnerabilities to the application. We don't know how the new addition would interact with other parts of the code that depends on the DiscountManager class.

Object Oriented Analysis and Design with Java

Solution

Refactor the code by adding an extra layer of abstraction that represents all types of discounts.

Create a new interface called BookDiscount that the CookbookDiscount and BiographyDiscount classes will implement.


```
public interface BookDiscount {

    String getBookDiscount();
}

class CookbookDiscount implements BookDiscount {

    @Override
    public String getBookDiscount() {
        String discount = "30% between Dec 1 and 24.";

        return discount;
    }
}
```


```
class BiographyDiscount implements BookDiscount {

    @Override
    public String getBookDiscount() {
        String discount = "50% on the subject's birthday.";

        return discount;
    }

}
```

Now, DiscountManager can refer to the BookDiscount interface instead of the concrete classes. When the processBookDiscount() method is called, pass both CookbookDiscount and BiographyDiscount as an argument, as both are the implementation of the BookDiscount interface.

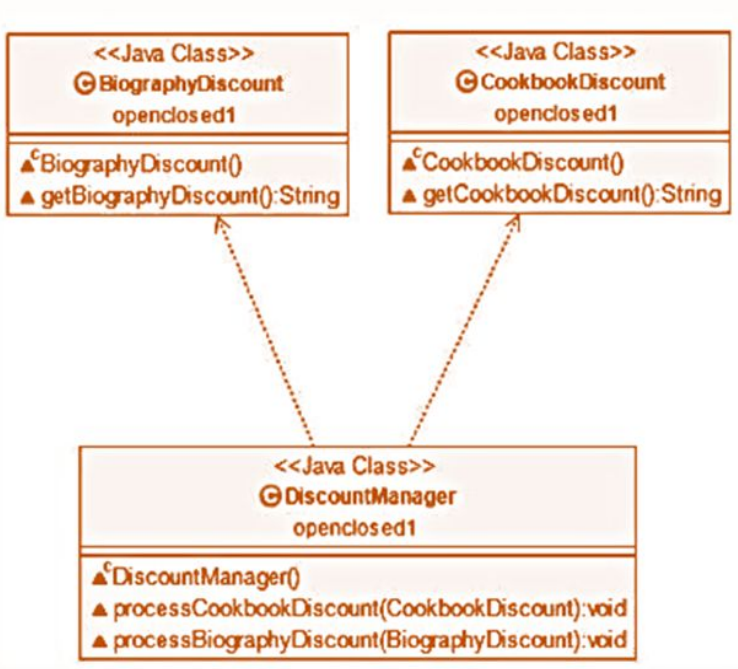
```
class DiscountManager {

    void processBookDiscount(BookDiscount discount) {...}
}
```

Object Oriented Analysis and Design with Java

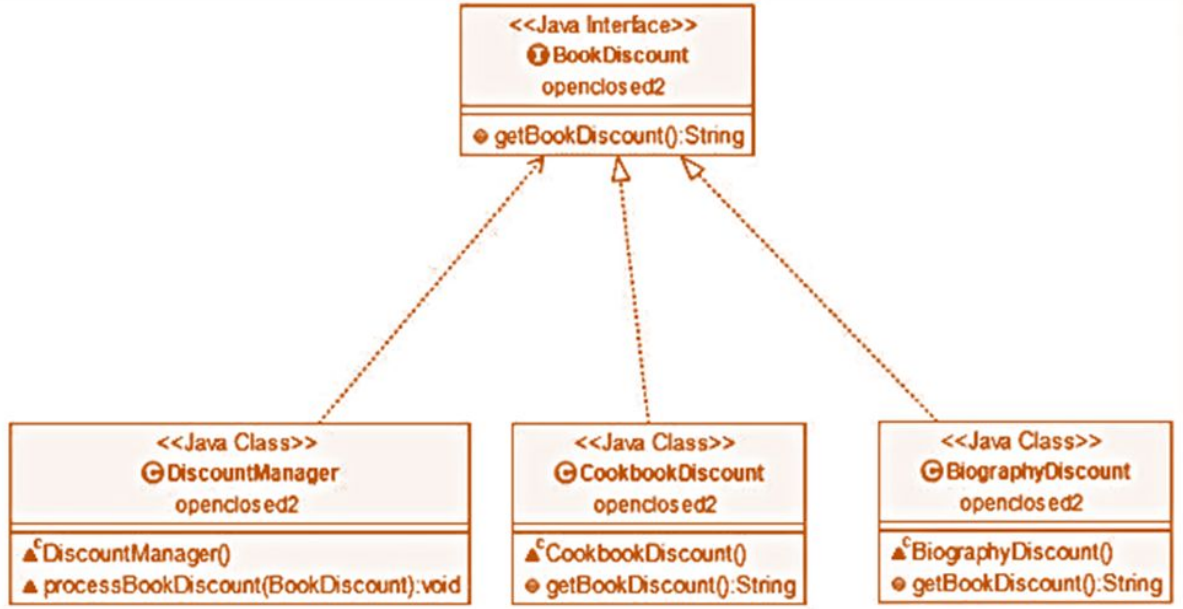
UML Class Diagram

The UML graph below shows how the code looks like before and after the refactoring.
On the left, you can see that DiscountManager depends on the CookbookDiscount and BiographyDiscount classes. On the right, all three classes depend on the BookDiscount abstract layer (DiscountManager references it, while CookbookDiscount and BiographyDiscount implement it).

Bad Design



Good Design



THANK YOU

J.Ruby Dinakar

Department of Computer Science and Engineering

rubydinakar@pes.edu