The Visitor Pattern - 'Revisited' using Data Oriented Programming techniques

Hello 👋

Design patterns are great 🤘🏻

- You don’t reinvent the wheel
- You can copy paste a proven template that has been industry tested
- But… are they all necessary?

some design patterns only exist in a programming language because that language lacks a way to elegantly and efficiently express it in its syntax

The strategy pattern - pre java 8

```java
// Product Category Enum
enum ProductCategory {
    STANDARD, PREMIUM, LUXURY
}

// 1. Strategy Interface
interface TaxCalculationStrategy {
    double calculateTax(double amount);
}

// 2. Concrete Strategy Implementations
class StandardTaxCalculation implements TaxCalculationStrategy {
    @Override
    public double calculateTax(double amount) {
        return amount * 0.10; // 10% tax
    }
}

class PremiumTaxCalculation implements TaxCalculationStrategy {
    @Override
    public double calculateTax(double amount) {
        return amount * 0.20; // 20% tax
    }
}

class LuxuryTaxCalculation implements TaxCalculationStrategy {
    @Override
    public double calculateTax(double amount) {
        return amount * 0.30; // 30% tax
    }
}
```

```java
// 3. Usage with switch statement
class TraditionalStrategyDemo {
    public static void main(String[] args) {
        //...
        double base = 100.0;

        // Using switch statement to select strategy
        TaxCalculationStrategy strategy;
        switch (category) {
            case STANDARD:
                strategy = new StandardTaxCalculation();
                break;
            case PREMIUM:
                strategy = new PremiumTaxCalculation();
                break;
            case LUXURY:
                strategy = new LuxuryTaxCalculation();
                break;
            default:
                System.out.println("Unsupported product category");
                return;
        }

        double taxAmount = strategy.calculateTax(base);
        System.out.println("Tax for $" + base + " (" + category + "): $" + taxAmount);
    }
}
```

Java 8 - yay for lambdas! ƛ

```java
// Product Category Enum
enum ProductCategory {
    STANDARD, PREMIUM, LUXURY
}

// 1. Functional Interface (Strategy)
@FunctionalInterface
interface TaxCalculator {
    double calculate(double amount);
}

// 2. Usage with Lambda Expressions and switch statement
class Java8StrategyDemo {
    public static void main(String[] args) {
        // ...
        double base = 100.0;

        // Using switch statement with lambda expressions
        TaxCalculator calculator;
        switch (category) {
            case STANDARD:
                calculator = amount1 -> amount1 * 0.10; // 10% tax
                break;
            case PREMIUM:
                calculator = amount1 -> amount1 * 0.20; // 20% tax
                break;
            case LUXURY:
                calculator = amount1 -> amount1 * 0.30; // 30% tax
                break;
            default:
                System.out.println("Unsupported product category");
                return;
        }

        double taxAmount = calculator.calculate(base);
        System.out.println("Tax for $" + base + " (" + category + "): $" + taxAmount);
```

Java 14 - switch expressions 😎

```java
// Using switch statement with lambda expressions
TaxCalculator calculator;
switch (category) {
    case STANDARD:
        calculator = amount1 -> amount1 * 0.10; // 10% tax
        break;
    case PREMIUM:
        calculator = amount1 -> amount1 * 0.20; // 20% tax
        break;
    case LUXURY:
        calculator = amount1 -> amount1 * 0.30; // 30% tax
        break;
    default:
        System.out.println("Unsupported product category");
        return;
}
```



```java
// 2. Usage with Switch Expressions
class ModernJavaStrategyDemo {
    public static void main(String[] args) {
        double amount = 100.0;

        // Direct lambda assignment with switch expression
        TaxCalculator calculator = switch (category) {
            case STANDARD -> amount1 -> amount1 * 0.10; // 10% tax
            case PREMIUM -> amount1 -> amount1 * 0.20;  // 20% tax
            case LUXURY -> amount1 -> amount1 * 0.30;   // 30% tax
        };

        double taxAmount = calculator.calculate(amount);
        System.out.println("Tax for $" + amount + " (" + category + "): $" + taxAmount);
    }
}
```

Design patterns become language functionality

- The strategy pattern pre java 8 was verbose and less readable
- Twisting in turns because java lacks proper language support
- As Java evolves, design patterns become **first class language citizens**

Enter the visitor pattern… 

The book curation system 👨🏽‍🏫

- A curator tasks you to process a library of books
- Collect some interesting facts
- Cant touch the domain layer 🚫
- An ideal candidate for the visitor pattern!

The Book Domain

Books always have the following properties:

- ISBN
- title
- author
- summary
- pages

However, the library also consists of fiction books and non-fiction books.

Fiction books

These are further subdivided in:

- Children's tale books
- Fantasy books
- Scifi books
  - with a ScifiTheme property

Non-fiction books

Non-fiction books are rated:

- They can optionally have 0, 1 or 2 ratings.
- A rating can be anonymous or be by a named reviewer.

The curators’ rules

Non - fiction

- A NonFictionBook always needs to have an InterestingNessFactor of at least interesting in order to be deemed interesting.
- • A NonFictionBook also always needs to have two ratings:
  - OR: both the first and second rating are good and by named reviewers
  - OR: the first rating is good and by a named reviewer and the second rating is BAD.
  - OR: there are two bad ratings

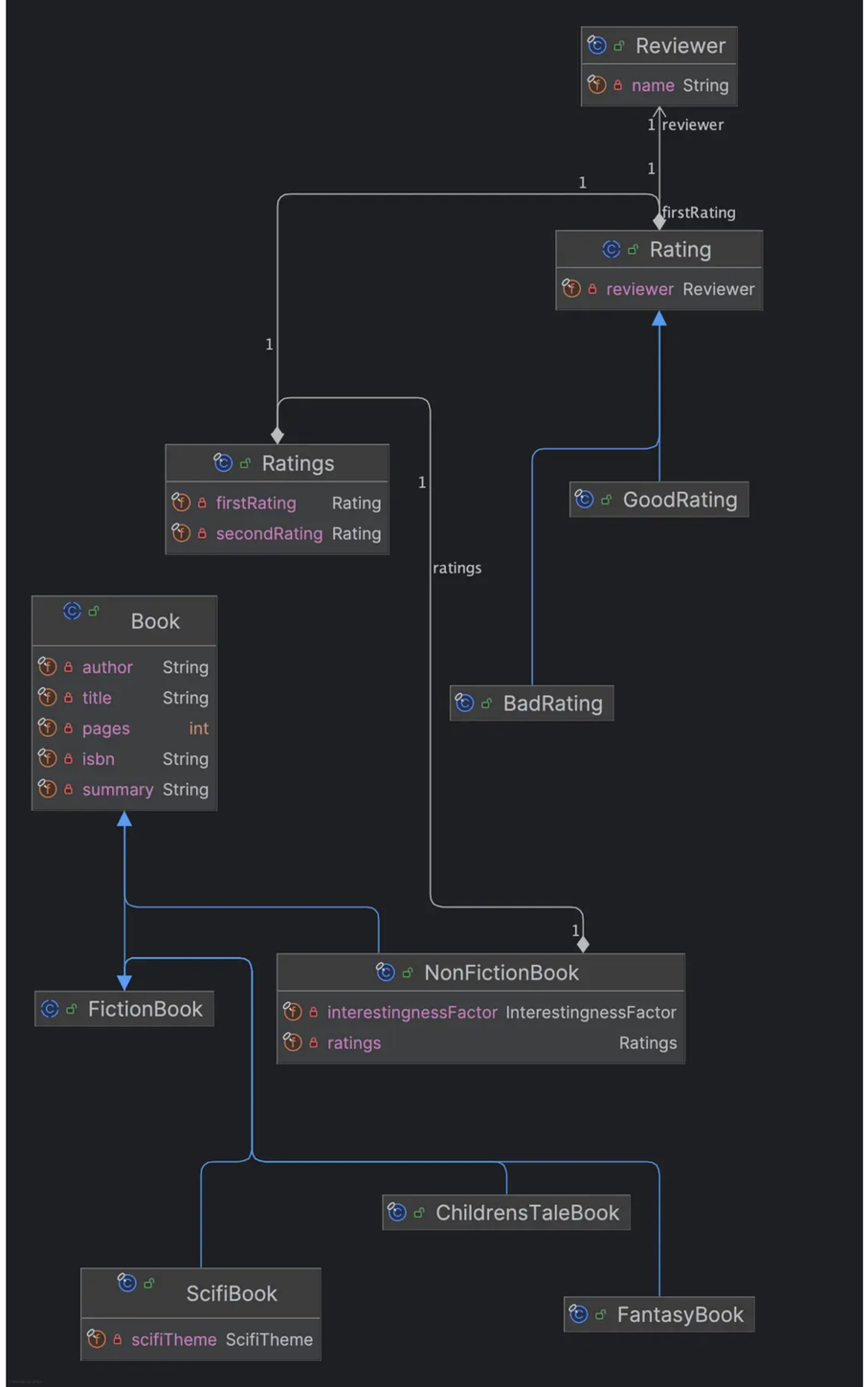
The curators’ rules

Fiction

- Fantasy book
  - A fantasy book is always found interesting.
  - Sci-fi book (if theme is space exploration or time travel)
  - Childrens' tale book
    - Books with 0 pages are considered interesting as they leave everything to imagination
    - Books with exactly 100 pages have a certain appeal
    - Books with 1000 or more pages are noteworthy due to their unusual length for children's literature

Everything that doesn’t match this pattern is uninteresting 😴

The OOP implementation 😷


Reviewer
name String
1
reviewer
1
1
firstRating
Rating
reviewer Reviewer
1
Ratings
firstRating Rating
secondRating Rating
1
ratings
GoodRating
Book
author String
title String
pages int
isbn String
summary String
BadRating
1
NonFictionBook
interestingnessFactor InterestingnessFactor
ratings Ratings
FictionBook
ChildrensTaleBook
ScifiBook
scifiTheme ScifiTheme
FantasyBook

Intermediate nodes 

Book.java

```java
public abstract class Book {

     private final String isbn;
     private final String title;
     private final String author;
     private final String summary;
     private final int pages;

     public Book(String isbn, String title, String author, String summary, int pages) {
          this.isbn = isbn;
          this.title = title;
          this.author = author;
          this.summary = summary;
         this.pages = pages;
     }

     // getters omitted for brevity
}
```

Leaf nodes 

ChildrensTaleBook.java

```java
public final class ChildrensTaleBook extends FictionBook  {

    public ChildrensTaleBook(String isbn, String title, String author, String summary, int pages) {
        super(isbn, title, author, summary, pages);
    }

}
```

Time to add the visitor pattern…

Component 1 - Visitor interface

BookVisitor.java

```
public interface BookVisitor {

    void visit(NonFictionBook nonFictionBook);

    void visit(ChildrensTaleBook childrensTaleBook);

    void visit(FantasyBook fantasyBook);

    void visit(ScifiBook scifiBook);
}
```

Time to add the visitor pattern…

Component 2 - Visitable interface (double dispatch)

Visitable.java

```java
public interface Visitable {

    void accept(BookVisitor visitor);
}
```

Book.java

```java
public abstract class Book implements Visitable { //... }
```

ChildrensTaleBook.java

```java
public final class ChildrensTaleBook extends FictionBook  {

    public ChildrensTaleBook(String isbn, String title, String author, String summary, int pages) {
        super(isbn, title, author, summary, pages);
    }

    // Each leaf node implements the interface like this:
    @Override
    public void accept(BookVisitor visitor) {
        visitor.visit(this);
    }
}
```

Time to add the visitor pattern…

Component 3 - Glue code

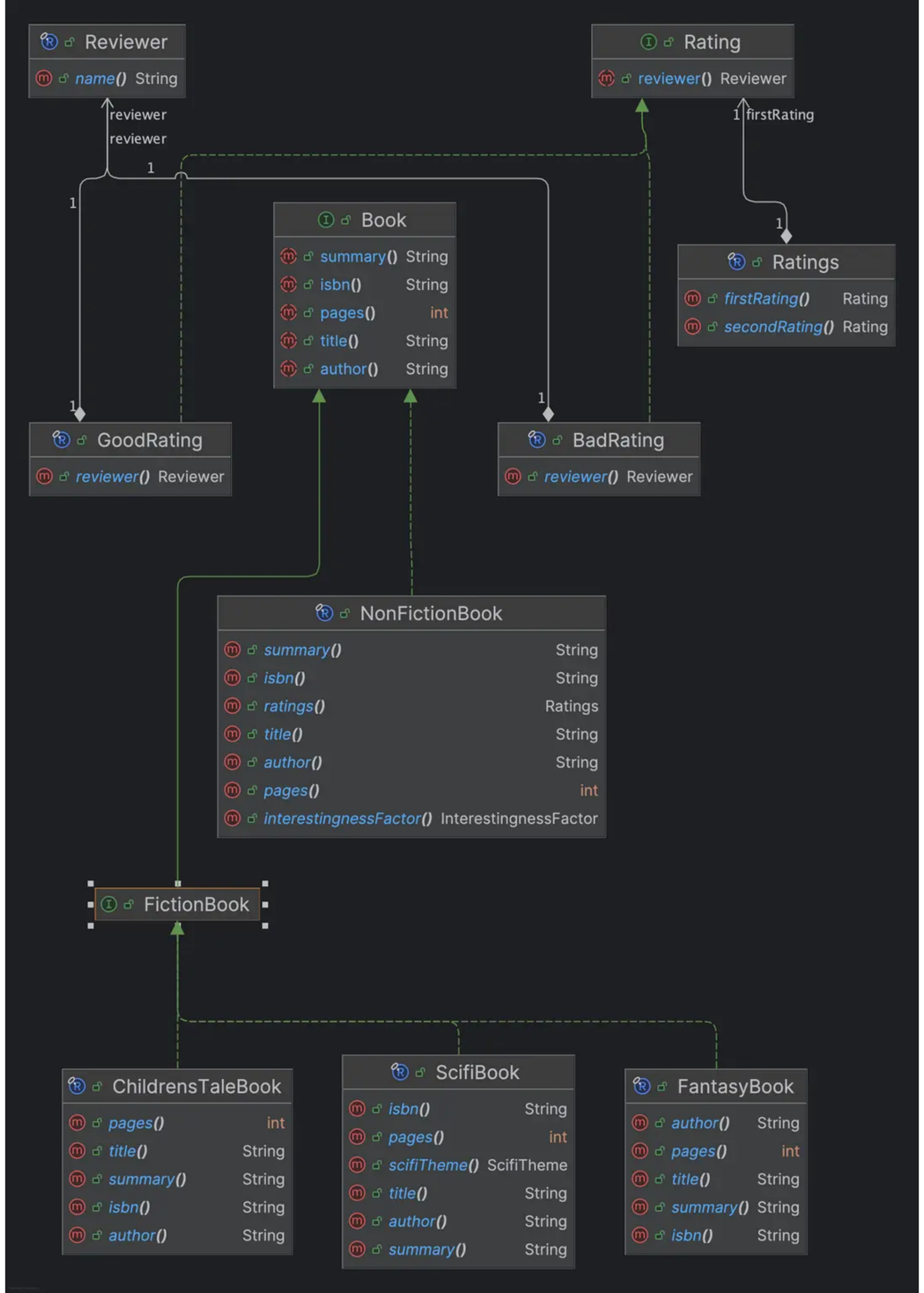
OOPSolution.java

```java
public class OOPSolution {
    public static void main(String[] args) {
        var mockLibrary = BookProvider.createMockLibrary();
        BookInterestingInfoVisitor booksInterestingInfoVisitor = new BookInterestingInfoVisitor();
        System.out.println("-- Begin collecting interesting information ---");
        for (Book book : mockLibrary) {
            book.accept(booksInterestingInfoVisitor);
        }
        System.out.println("-- End collecting interesting information ---");
        System.out.println(booksInterestingInfoVisitor.retrieveInformationCollection());

    }
}
```

And now… the algorithm 👹

DOP to the rescue 


Reviewer
name() String
Rating
reviewer() Reviewer
reviewer
reviewer
1
1
1
firstRating
1
Book
summary() String
isbn() String
pages() int
title() String
author() String
Ratings
firstRating() Rating
secondRating() Rating
1
1
GoodRating
reviewer() Reviewer
BadRating
reviewer() Reviewer
NonFictionBook
summary() String
isbn() String
ratings() Ratings
title() String
author() String
pages() int
interestingnessFactor() InterestingnessFactor
FictionBook
ChildrensTaleBook
pages() int
title() String
summary() String
isbn() String
author() String
ScifiBook
isbn() String
pages() int
scifiTheme() ScifiTheme
title() String
author() String
summary() String
FantasyBook
author() String
pages() int
title() String
summary() String
isbn() String

Seal the hierarchy : sealed interfaces!

Book.java

```java
public sealed interface Book permits FictionBook, NonFictionBook {
     String isbn();
     String title();
     String author();
     String summary();
     int pages();
}
```

FictionBook.java

```java
public sealed interface FictionBook extends Book permits ChildrensTaleBook, FantasyBook, ScifiBook { }
```

Seal the hierarchy : sealed interfaces!

- Let data just be data
- Define a closed type hierarchy for the compiler
- Switch statement will iterate *exhaustively* over these finite states later on
- Interfaces instead of classes: records cant extend sealed classes

The leaf nodes are records now!

FantasyBook.java

```
public record FantasyBook(
        String isbn,
        String title,
        String author,
        String summary,int pages) implements FictionBook{}
```

And now… the algorithm 👼

Are you excited about DOP yet?

- https://www.youtube.com/watch?v=8FRU_aGY4mY -> talk by Nicolai Parlog
- https://www.infoq.com/articles/data-oriented-programming-java/
- https://www.manning.com/books/data-oriented-programming-in-java
- https://www.youtube.com/watch?v=GurtoM8i2TE -> future of pattern matching
- https://wimdetroyer.com/blog/visitor-pattern-in-dop this presentation in blog form