Visitor Design Pattern

Module Objectives

At the end of this module, the student will be able to:

1. Grasp the core concept of the Visitor Pattern ;
2. Recognize common scenarios where the Visitor Pattern shines;
3. Become familiar with the building blocks of the Visitor Pattern.

The Visitor Pattern is a behavioral design pattern that lets you add new operations to a hierarchy of objects without modifying their classes. It separates the logic for performing operations on these objects from the objects themselves. This allows for:

* **Open/Closed Principle compliance:** New functionalities can be added through Visitor implementations without touching existing object classes.
* **Loose Coupling:** Objects are unaware of the specific visitor classes that operate on them.

Benefits

The Visitor Design Pattern is a powerful tool for separating the logic of operations on objects from the objects themselves. This leads to several advantages, making it applicable in various scenarios:

**1. Adding New Operations to Existing Object Hierarchies:**

Imagine you have a complex hierarchy of shapes (Circle, Square, Triangle) and want to introduce new operations like calculating area, perimeter, or drawing them on the screen. The Visitor Pattern allows you to define new visitor classes (AreaCalculator, PerimeterCalculator, DrawVisitor) that implement these operations without modifying the existing shape classes.

**2. Processing Objects in Different Ways:**

Suppose you have a collection of products (Electronics, Clothing, Furniture) and need to apply different tax calculations or discounts based on their type. Visitor Pattern lets you create specific visitor classes (ElectronicsTaxCalculator, ClothingDiscountVisitor) that handle the logic for each product type, keeping the product classes clean.

**3. Double Dispatch**

In certain situations, you might need runtime polymorphism based on both the object type and the visitor type. The Visitor Pattern can simulate double dispatch by having the accept method in the Account interface take the visitor as an argument. This allows the visitor to determine the specific logic based on the account type it's dealing with.

**4. Decoupling Algorithms from Data Structures:**

If you have different data structures (ArrayList, LinkedList) and want to implement sorting algorithms (BubbleSort, MergeSort) that work on both, the Visitor Pattern can help. Define a sorting visitor interface with a sort method, and create concrete visitor classes for each sorting algorithm. This keeps the sorting logic separate from the data structures.

Here are some additional real-world examples where the Visitor Pattern might be useful:

* **Compiling expressions:** In a compiler, the Visitor Pattern can be used to define different visitor classes (e.g., CodeGeneratorVisitor, SemanticAnalyzerVisitor) that perform specific tasks on an abstract syntax tree representing the code.
* **Generating reports:** In a reporting system, visitor classes can be used to generate reports in different formats (e.g., PDFVisitor, HTMLVisitor) based on the same data structure.
* **Validating XML documents:** Visitor classes can be created to perform different validation checks on different parts of an XML document.

However, the Visitor Pattern isn't always the best choice. Here are some considerations:

* **Complexity:** It can introduce additional complexity to your code, especially for simple operations.
* **Performance:** The extra method calls and runtime type checks can sometimes lead to a performance overhead.
* **Overuse:** Don't use it just for the sake of using a design pattern. If a simpler solution exists, it might be preferable.

The Visitor Pattern offers a flexible and maintainable approach for adding new operations to existing object hierarchies. Consider using it when you need to decouple the logic of operations from the objects themselves, especially when dealing with complex object structures or the need for runtime polymorphism based on both object and visitor types.

Components

1. **Visitable Interface:** This interface declares an accept method that takes a Visitor as an argument. This method allows the visitor to perform the desired operation on the Visitable object.
2. **Concrete Visitable Classes:** These classes implement the Visitable interface and represent the objects you want to perform operations on. They typically override the accept method to delegate the operation to the specific Visitor.
3. **Visitor Interface:** This interface declares a visit method for each concrete Visitable class. Each visit method takes the corresponding Visitable object as an argument and defines the operation to be performed on that type of object.
4. **Concrete Visitor Classes:** These classes implement the Visitor interface and provide specific implementations for the visit methods defined in the Visitor interface. These implementations define the actual operations to be performed on each concrete Visitable class.

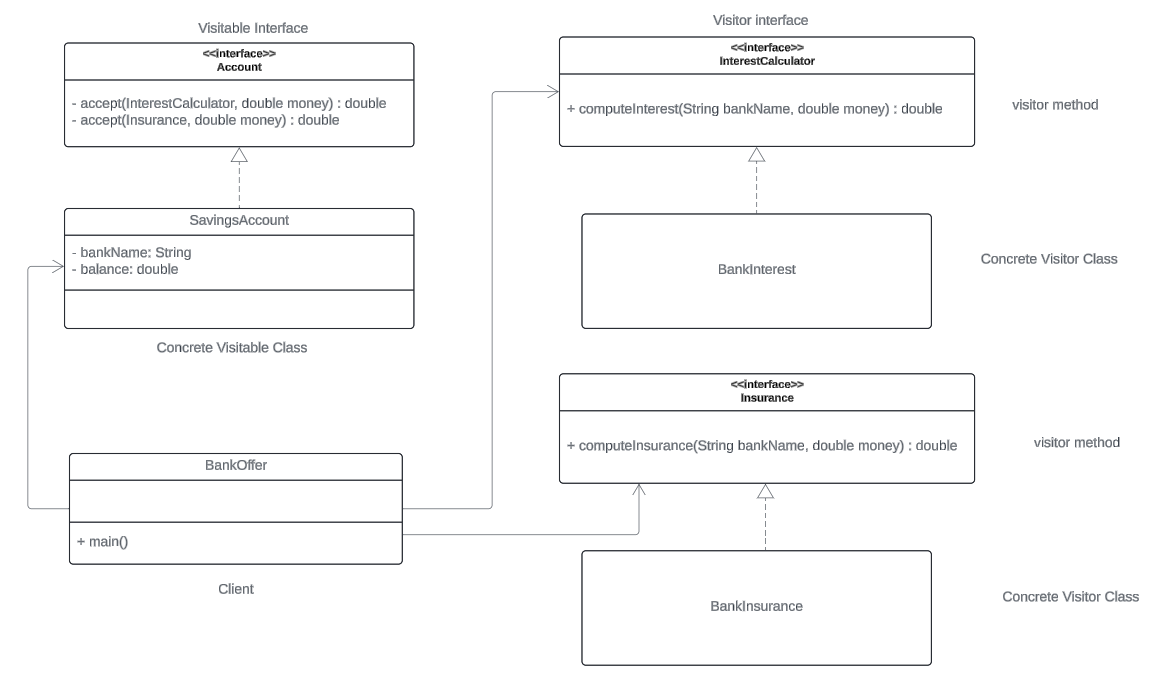
Problem Scenario

Imagine you're considering depositing money in one of three banks: Cimb, SeaBank, or GoTyme. Each bank offers a unique interest rate and potential insurance benefits to entice you. Here's a breakdown of their offerings:

* **Cimb:**
  + Interest rate: 2.6%
  + Insurance: Up to a maximum of ₱250,000, proportional to your account balance (minimum balance of ₱50,000 required to qualify for insurance)
* **SeaBank:**
  + Interest rate: 5%
  + Insurance: Up to ₱1,000,000, proportional to your account balance (minimum balance of ₱50,000 required to qualify for insurance)
* **GoTyme:**
  + Interest rate: 4.5%
  + Insurance: No insurance offered

Implement a visitor design pattern to apply the open/closed principle and promote loose coupling.

UML Class Diagram



Implementation Codes

See this link :  <https://github.com/JerryEsperanza/visitorPattern.git>