Visitor Pattern

Introduction

The Visitor Design Pattern is a behavioral design pattern that allows you to add new operations to a collection of objects without changing the objects themselves. Imagine you have different types of documents in your system like PDF files, Word documents, and text files. Without the Visitor pattern if you want to perform operations like exporting, printing, or compressing you would need to add these methods directly to each document class. This means modifying existing classes every time you need a new operation which violates the Open/Closed Principle. The Visitor pattern solves this by creating separate visitor classes for each operation. These visitors visit each document type and perform the operation specific to that type just like how a tax auditor visits different businesses and applies tax rules specific to each business type, a visitor visits different object types and applies operations specific to each type. The pattern is called Visitor because it involves visitors traversing through an object structure and performing operations on each element they encounter.

Problem Statement

Consider a scenario where you have different types of document classes in your system such as PDFDocument, WordDocument, and TextDocument. Your application needs to perform various operations on these documents like exporting, printing, compressing, and validating. Without the Visitor pattern, you would need to add each operation method directly to every document class. This means if you have 3 document types and 5 operations, you must implement 15 methods across your document classes. With 5 document types and 10 operations you need 50 methods. Without Visitor Pattern you need to create separate methods like exportPDF(), exportWord(), exportText(), printPDF(), printWord(), printText(), etc. This approach leads to massive code duplication because similar operation logic is repeated across multiple document classes. For example all document classes will have similar structure for operations, even though the actual implementation differs for each document type. When you want to add a new operation like generatePreview(), you must modify all document classes by adding this method to each one. Similarly adding a new document type requires implementing all operations in that class.

This tight coupling between document types and operations makes the system very difficult to extend and maintain. Every time you add a new operation, you must touch and modify every document class in your system. If you need to change how an operation works, you must update that logic in multiple classes. This violates the

Open/Closed Principle because classes should be open for extension but closed for modification. The document classes become bloated with operation methods that mix concerns - the document class should represent the document structure and data, not all possible operations that can be performed on it. If operations need to access private members of document classes, you must break encapsulation by exposing internal details. The classes become overly complex with too many responsibilities, violating the Single Responsibility Principle.

Solution

The Visitor Design Pattern solves this problem by separating the operations from the object structure. Instead of adding operation methods directly to document classes, it creates two independent hierarchies that work together through a double dispatch mechanism. One hierarchy is created for document types i.e. PDFDocument, WordDocument and TextDocument, all implementing a common Element interface that defines an accept(Visitor) method. Another separate hierarchy is created for operations i.e. ExportVisitor, PrintVisitor, CompressVisitor, each implementing a common Visitor interface that defines visit methods for each document type.

The document classes don't directly implement operation methods. Instead each document class only implements the accept(Visitor) method that accepts a visitor and calls the visitor's appropriate visit method passing itself as an argument. For example, PDFDocument.accept(visitor) calls visitor.visitPDF(this). The visitor classes define how each operation is performed for each specific document type. When you want to perform an operation on a document, you create a visitor instance and pass it to the document's accept method. The document then delegates the actual operation to the visitor, which knows how to handle that specific document type.

This separation allows adding new operations by creating new visitor classes without touching document classes, and adding new document types by updating the visitor interface and implementing accept methods in the new document class. The pattern uses double dispatch - the visitor's visit method is selected based on both the visitor type and the document type. This loose coupling makes the system easy to extend and maintain. Changes to operation behavior only require updating the visitor classes, and changes to document structure only affect the document classes. Related operations are grouped together in visitor classes making the code more organized and maintainable.

Code

Code implementation of Chain of Responsibility pattern:
https://github.com/prateek27/design-patterns-java/tree/main/design-patterns/src/main/java/org/prateek/BehaviouralPatterns/VisitorPattern

Use Cases

The Visitor Design Pattern is most useful when you have a stable object structure but frequently need to add new operations to those objects. If the object structure changes rarely but operations change frequently, Visitor pattern provides the perfect solution.

Visitor pattern should be used when there is a need to perform operations on objects of different types in a collection without modifying the object classes. This is particularly important when operations need to be added without changing existing classes, following the Open/Closed Principle. Consider a document management system where document types remain stable but new operations like exporting, printing, compressing, validating, and encrypting are added frequently.

The pattern is highly beneficial when you want to separate concerns and group related operations together. If multiple operations need to work on the same set of object types, Visitor pattern allows grouping all operation logic for a specific object type in one place, making the code more organized and maintainable. This is useful when operations need to share state or work together.

Visitor pattern works well when operations need access to private members of object classes. Instead of breaking encapsulation by exposing internal details, the pattern allows operations to be defined externally while still having access to object internals through the accept method mechanism. The pattern is also suitable when you want to avoid type checking or casting when processing collections of different object types.

However, the Visitor pattern should not be used when the object structure changes frequently. If new object types are added often, you must update the Visitor interface and all existing visitor implementations each time, which can be costly. The pattern is not suitable for systems where object types are unstable or change frequently.

The pattern is also not appropriate when there are only a few operations that rarely change. If operations are stable and unlikely to be added, the overhead of maintaining visitor classes and the double dispatch mechanism is not justified. Simple methods in object classes would be more straightforward. The pattern should only be used when

there is genuine need for frequent addition of operations without modifying object classes.

Applications

Compiler and AST Traversal: Compilers and interpreters use Visitor pattern extensively for traversing Abstract Syntax Trees (AST). The AST structure with nodes like statements, expressions, and declarations remains stable, while operations like type checking, code generation, optimization, and pretty printing are added frequently. Visitors like TypeCheckerVisitor, CodeGeneratorVisitor, and OptimizerVisitor traverse the AST and perform operations specific to each node type without modifying the node classes. Tools like Java Compiler, Python AST module, and TypeScript compiler use this pattern.

File System Operations: File system libraries use Visitor pattern for directory traversal and file operations. File system structures with files and directories remain stable, while operations like searching, copying, archiving, and virus scanning are added frequently. Visitors like SearchVisitor, CopyVisitor, and ArchiveVisitor can traverse file system trees and perform operations on each file and directory without modifying the file system classes. Applications like file managers, backup tools, and antivirus software use this pattern.

Document Processing Systems: Document processing frameworks use Visitor pattern for processing document elements. Document structures with elements like paragraphs, images, tables, and headers remain stable, while operations like rendering, exporting, printing, and converting are added frequently. Visitors like RenderVisitor, ExportVisitor, and PrintVisitor can process document elements and perform operations specific to each element type without modifying element classes. Tools like HTML parsers, Markdown processors, and word processors use this pattern.

Expression Evaluation: Mathematical and logical expression evaluators use Visitor pattern for processing expression trees. Expression structures with nodes like numbers, variables, operators, and functions remain stable, while operations like evaluation, simplification, differentiation, and validation are added frequently. Visitors like EvaluatorVisitor, SimplifierVisitor, and DifferentiatorVisitor can traverse expression trees and perform operations on each node without modifying node classes. Applications like calculators, formula processors, and symbolic math systems use this pattern.

XML and JSON Processing: XML and JSON parsers use Visitor pattern for processing parsed elements. Parsed document structures with elements, attributes, and text nodes

remain stable, while operations like validation, transformation, querying, and serialization are added frequently. Visitors like ValidatorVisitor, TransformerVisitor, and QueryVisitor can traverse parsed structures and perform operations on each element without modifying parser classes. Libraries like DOM parsers, JSON processors, and configuration readers use this pattern.

Reporting Systems: Report generation systems use Visitor pattern for processing report components. Report structures with sections like headers, tables, charts, and footers remain stable, while operations like rendering, exporting, formatting, and validation are added frequently. Visitors like RenderVisitor, ExportVisitor, and FormatVisitor can process report components and perform operations specific to each component type without modifying component classes. Business intelligence tools and report generators use this pattern.

Advantages and Disadvantages

Advantages

Visitor pattern provides strong separation between object structure and operations. This separation allows both to be developed, tested and maintained independently. Changes in operations do not affect the object classes and vice versa, making the codebase more stable and easier to manage.
The pattern enables easy addition of new operations without modifying existing classes. New operations can be added by creating new visitor classes without touching the object structure, following the Open/Closed Principle where classes are open for extension but closed for modification. This allows the system to evolve by adding new functionality without breaking existing code.

Visitor pattern groups related operations together in visitor classes, making the code more organized and maintainable. All logic for a specific operation across different object types is contained in one visitor class, making it easier to understand, test and modify operation behavior. Operations can also share state and work together within a visitor.

The pattern allows operations to access private members of object classes without breaking encapsulation. Instead of exposing internal details through getter methods, the accept method mechanism allows visitors to access object internals while maintaining proper encapsulation. This protects the internal structure from external dependencies.

Visitor pattern eliminates the need for type checking or casting when processing collections of different object types. The double dispatch mechanism automatically selects the correct visit method based on both visitor type and object type, making code cleaner and less error-prone. This improves type safety and code readability.

The pattern improves code reusability as visitors can be shared across multiple object structures. It also makes operations testable independently since each visitor can be tested in isolation without requiring the full object structure.

Disadvantages

Visitor pattern increases overall complexity of the codebase. It introduces additional classes, interfaces and the double dispatch mechanism which can make the system harder to understand initially. For simple scenarios, this added complexity may not be justified.

The pattern adds a level of indirection through the double dispatch mechanism. This can make the code slightly harder to follow and debug as method calls pass through the accept method and then to the visitor's visit method. There is also a minor performance overhead due to this indirection.

Visitor pattern requires careful upfront design to identify the right separation between object structure and operations. If the object structure is not stable or if operations are unlikely to be added, the pattern may not provide the intended benefits and could make the system more complicated than necessary.

When new object types are added, you must update the Visitor interface to include a visit method for the new type, and all existing visitor implementations must implement this new method. This can be costly and violates the Open/Closed Principle from the visitor's perspective. If object types change frequently, this overhead becomes significant.

For systems with only a few operations or operations that rarely change, Visitor pattern is overkill. The overhead of maintaining visitor classes, the Visitor interface, and the double dispatch mechanism is not worth it when simpler approaches would suffice. The pattern should only be used when there is genuine need for frequent addition of operations without modifying object classes.