**Applicability**

1. Perform the same kind of operation on a set of **different object types** in a complex structure
2. Keep your main classes **clean and focused** on their main jobs by moving extra behaviors into **separate visitor classes**
3. Only **apply** an operation **to some classes** in a hierarchy and ignore the rest

**Other realworld problem**

Mobile App Settings Sync and Diagnostics

*Context*

In a large **mobile app** (banking or fitness), you have a modular settings system:

* NotificationSetting
* ThemeSetting
* SecuritySetting
* PrivacySetting
* LocationSetting

Each type of setting:

* Has different **data formats**
* Needs to be **synced** with cloud or local storage
* May require **logging or diagnostics**
* Might be conditionally sent to different servers based on privacy settings

*How Visitor Pattern Solves It*

You create a SettingVisitor interface:

A screen shot of a computer

AI-generated content may be incorrect.

Then define visitors like:

* CloudSyncVisitor: pushes data to the cloud
* LocalSaveVisitor: writes settings to local storage
* DiagnosticVisitor: logs current setting values for support
* AnalyticsVisitor: collects anonymized data for analysis

🡪 Each setting only needs to implement an accept(visitor) method

🡪 You can add new features without touching any existing setting classes

**Relative patterns**

1. Visitor + Composite

* **Composite** structures let you build **trees of objects**
* **Visitor** helps you perform operations **across the entire structure**, **without modifying node classes**

1. **Visitor + Iterator**

* **Iterator** is used to **traverse elements** of a collection

**🡪 Iterator** handles **how you go through the structure**  
**🡪 Visitor** handles **what to do with each element,** especially when elements are **not of the same type**

1. **Visitor + Command**

* **Command pattern** encapsulates a **request or action as an object** allowing you to:
* Queue actions
* Undo/redo operations
* Log operations
* Execute actions later
* If you need to:
* Perform **different operations** on **different object types**
* And then **encapsulate those operations** for queuing, undoing, or delaying

🡪 Visitor **decides what to do based on object type**

**🡪** Command **wraps how and when the action is performed**

**Quizzes**

**1. What is the main purpose of the Visitor pattern?**

A. To allow objects to change their behavior based on their internal state  
B. To define a new operation without changing the classes of the elements it operates on  
C. To ensure only one instance of a class exists  
D. To manage dependencies between objects

**2. Which principle does the Visitor pattern help follow?**

A. Liskov Substitution Principle  
B. Interface Segregation Principle  
C. Open/Closed Principle  
D. Dependency Inversion Principle

**3. What must each element in the object structure do to work with a Visitor?**

A. Inherit from the Visitor class  
B. Implement a visit() method  
C. Provide an accept() method that takes a Visitor  
D. Store a list of Visitor objects

**4. In which scenario is the Visitor pattern most useful?**

A. When you want to create many instances of a class  
B. When you need to ensure thread safety  
C. When you want to perform many unrelated operations on objects in a class hierarchy  
D. When you want to observe changes in object states

**5. How does the Visitor pattern relate to the Composite pattern?**

A. Visitor pattern helps to build the tree structure  
B. Visitor can traverse the Composite tree and apply different logic to different node types  
C. Composite allows Visitor to change the structure dynamically  
D. They cannot be used together