Allows us to add extra behavior to entire hierarchy of classes

Need to define a new operation on an entire class hierarchy

Do not want to keep modifying every class in the hierarchy

Need access to the non-common aspects of classes in the hierarchy

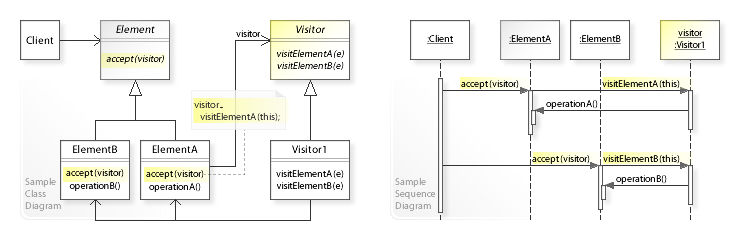
So Visitor is a pattern where a component (visitor) is allowed to traverse the entire inheritance hierarchy, implemented by propagating a single visit () method throughout the entire hierarchy.

The **visitor** [design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) is a way of separating an [algorithm](https://en.wikipedia.org/wiki/Algorithm) from an [object](https://en.wikipedia.org/wiki/Object_(computer_science)) structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. It is one way to follow the [**open/closed principl**e](https://en.wikipedia.org/wiki/Open/closed_principle).

**GOF Definition**

*Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates*

**Class Diagram**



**What problems can the Visitor design pattern solve?**

* It should be possible to define a new operation for (some) classes of an object structure without changing the classes.

When new operations are needed frequently and the object structure consists of many unrelated classes, it's inflexible to add new subclasses each time a new operation is required because distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change.

**What solution does the Visitor design pattern describe?**

* Define a separate (visitor) object that implements an operation to be performed on elements of an object structure.
* Clients traverse the object structure and call a *dispatching operation accept (visitor)* on an element — that "dispatches" (delegates) the request to the "accepted visitor object". The visitor object then performs the operation on the element ("visits the element").

## Design participants/components

The participant classes in this pattern are:

**Visitor** – This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.

**ConcreteVisitor** – For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.

**Visitable** – is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.

**ConcreteVisitable** – Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

**Visitor vs Strategy**

**Visitor** is for when you have a family of classes and you need to add new functionality to every class in that family but not touch the classes themselves (or wish to have that new functionality all defined in one place - the visitor)

**Strategy** is for when you have a family of classes that need to be able to do something in order to work properly (such as sort some objects they contain) but you want the client or your dependency injection to tell them which way to go about doing that.

The main difference is that the *Strategy Pattern* encapsulates a single group of related behaviors, while the *Visitor Pattern* encapsulates multiple such groups.

* **You should use the *Strategy Pattern* when you need to encapsulate a behavior** - If you have a family of algorithms and you need to choose among them at run time, you should use Strategy Pattern. This is very common: it happens every time you [program to an interface](https://stackoverflow.com/q/383947/335858).
* **You should use the *Visitor Pattern* to implement *double dispatch*** - If you have a group of algorithms that need to be virtual in relation to more than one object. This is far less common, in part because it is much harder to implement.

**Visitor**  
  
The intent of the visitor pattern is to add operations to a class which has rich data structure. If we are not able to identify all the operations in for the class but we expect in future we may need to add method for the class, but it is not advisable to alter the class, so to be on safer side we can put a visitor and share the data to the visitor class  
  
**Strategy**  
  
The intent of strategy pattern is to define a family of algorithm, encapsulate each one, and make them interchangeable. Let’s the algorithm vary independently from the clients that use it. This pattern is used to add different algorithm nothing to do with the data in the class

**What is Double Dispatch?**

Double Dispatch is a simple idea: **change behavior also depending on the caller's class, and not only on the called class**.

### Double Dispatch vs Method Overloading

**Double dispatch determines the method to invoke at runtime based both on the receiver type and the argument types**.

Java doesn't support double dispatch.

**Note that double dispatch is often confused with method overloading, which is not the same thing**. Method overloading chooses the method to invoke based only on compile-time information, like the declaration type of the variable.