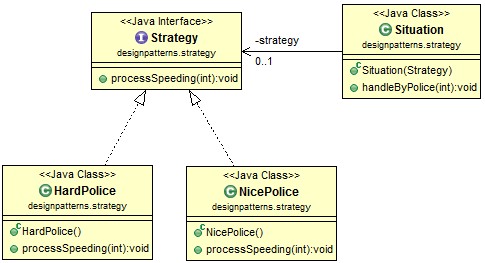
Strategy pattern is also called policy pattern.

Here is a story about Strategy pattern. Suppose Mike sometimes speeds when driving, but he doesn't always do that. He may be stopped by a police officer. It's possible that the police is very nice, who would let him go without any ticket or with a simple warning. (Let call this kind of police "NicePolice".) Also it's possible that he may be caught by a hard police and gets a ticket. (Let's call this kind of police "HardPolice".) He doesn't know what kind of police would stop him, until he actually gets caught, that is, run-time. This is the whole point of Strategy pattern.

Strategy Pattern Class Diagram

[](http://www.programcreek.com/wp-content/uploads/2011/01/strategy-pattern-class-diagram.jpg)

Strategy Pattern Java Code

Define a interface Strategy, which has one method processSpeeding()

|  |
| --- |
| **public** **interface** Strategy {  *//defind a method for police to process speeding case.*  **public** **void** processSpeeding(**int** speed);  } |

Now we have two kinds of police officers.

|  |
| --- |
| **public** **class** NicePolice **implements** Strategy{  @Override  **public** **void** processSpeeding(**int** speed) {  System.out.println("This is your first time, be sure don't do it again!");  }  } |
| **public** **class** HardPolice **implements** Strategy{  @Override  **public** **void** processSpeeding(**int** speed) {  System.out.println("Your speed is "+ speed+ ", and should get a ticket!");  }  } |

Define a situation in which a police officer will be involved to process speeding.

|  |
| --- |
| **public** **class** Situation {  **private** Strategy strategy;    **public** Situation(Strategy strategy){  **this**.strategy = strategy;  }    **public** **void** handleByPolice(**int** speed){  **this**.strategy.processSpeeding(speed);  }  } |

Finally, try the result.

|  |
| --- |
| **public** **class** Main {  **public** **static** **void** main(String args[]){  HardPolice hp = **new** HardPolice();  NicePolice ep = **new** NicePolice();    *// In situation 1, a hard officer is met*  *// In situation 2, a nice officer is met*  Situation s1 = **new** Situation(hp);  Situation s2 = **new** Situation(ep);    *//the result based on the kind of police officer.*  s1.handleByPolice(10);  s2.handleByPolice(10);  }  } |

Output is:

Your speed is 10, and should get a ticket!

This is your first time, be sure don't do it again!

You can compare this pattern with[State pattern](http://www.programcreek.com/2011/07/java-design-pattern-state/) which is very similar. The major difference is that State pattern involves changing the behavior of an object when the state of the object changes while Strategy pattern is mainly about using different algorithm at different situation.

Strategy Pattern in JDK

1). Java.util.Collections#sort(List list, Comparator < ? super T > c)  
2). java.util.Arrays#sort(T[], Comparator < ? super T > c)

The sort method use different Comparator in different situations. To know more about this method, check out [Deep Understanding of Arrays.sort()](http://www.programcreek.com/2013/11/arrays-sort-comparator/).

[Java Design Pattern: Visitor](http://www.programcreek.com/2011/05/visitor-design-pattern-example/)

Visitor pattern is a design pattern commonly used in the parser of a compiler, such as Eclipse JDT AST Parser.

Basically, there are two interfaces - Visitor and Element - in Visitor pattern.

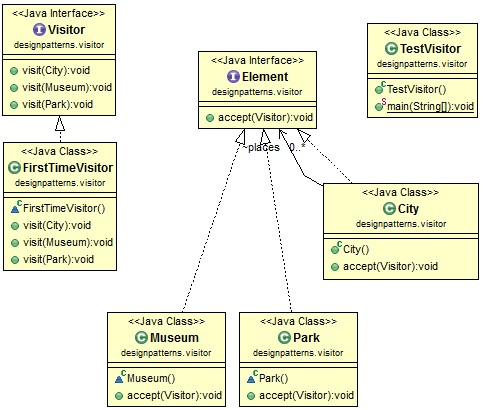
Visitor pattern story

In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming) and [software engineering](https://en.wikipedia.org/wiki/Software_engineering), 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 structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. It is one way to follow the [open/closed principle](https://en.wikipedia.org/wiki/Open/closed_principle).

In essence, the visitor allows one to add new [virtual functions](https://en.wikipedia.org/wiki/Virtual_function) to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through [double dispatch](https://en.wikipedia.org/wiki/Double_dispatch).

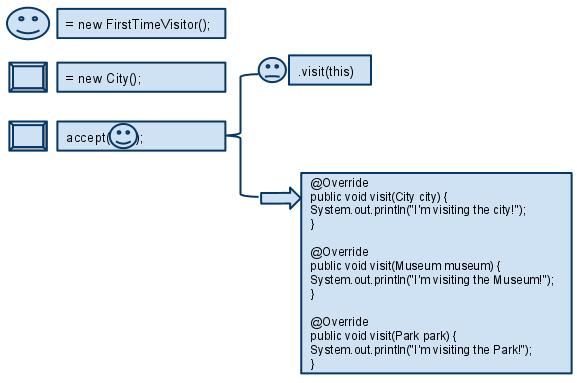
Suppose a first time visitor comes to New York City. He want to visit the city and the city accepts his visit. Once the visitor starts visit, it automatically visit everything, and he doesn't need to call a method when he wants to go to a museum. The travel is a package!

Visitor pattern class diagram

[](http://www.programcreek.com/wp-content/uploads/2011/05/visitor-pattern-class-diagram.jpg)

Steps of Visitor pattern

This diagram shows the steps of visiting.

[](http://www.programcreek.com/wp-content/uploads/2011/05/VisitorPatternWorkFlow.jpg)

The working process is like the following:

1. A visitor FirstTimeVisitor and an element City are created.  
2. Program starts with "City accept a visitor".  
3. The accept method in City defines that let this visitor to visit.  
4. Accepted visitor call it's overloaded method "visit" to visit this City.

Visitor pattern Java code

|  |
| --- |
| **import** java.util.ArrayList;    **interface** Visitor {  **public** **void** visit(City city);  **public** **void** visit(Museum museum);  **public** **void** visit(Park park);  }    **class** FirstTimeVisitor **implements** Visitor {    @Override  **public** **void** visit(City city) {  System.out.println("I'm visiting the city!");  }    @Override  **public** **void** visit(Museum museum) {  System.out.println("I'm visiting the Museum!");  }    @Override  **public** **void** visit(Park park) {  System.out.println("I'm visiting the Park!");  }  }    **interface** Element {  **public** **void** accept(Visitor visitor);  }    **class** City **implements** Element {    ArrayList<Element> places = **new** ArrayList<Element>();    **public** City() {  places.add(**new** Museum());  places.add(**new** Park());  }    @Override  **public** **void** accept(Visitor visitor) {  System.out.println("City is accepting visitor.");  visitor.visit(**this**);    **for** (Element e : places) {  e.accept(visitor);  }  }  }    **class** Museum **implements** Element {  @Override  **public** **void** accept(Visitor visitor) {  System.out.println("Museum is accepting visitor.");  visitor.visit(**this**);  }  }    **class** Park **implements** Element {  @Override  **public** **void** accept(Visitor visitor) {  System.out.println("Park is accepting visitor.");  visitor.visit(**this**);  }    }    **public** **class** TestVisitor {  **public** **static** **void** main(String[] args) {  FirstTimeVisitor visitor = **new** FirstTimeVisitor();  City city = **new** City();  city.accept(visitor);  }  } |

Output:

City is accepting visitor.  
I'm visiting the city!  
Museum is accepting visitor.  
I'm visiting the Museum!  
Park is accepting visitor.  
I'm visiting the Park!

Visitor pattern in JDK

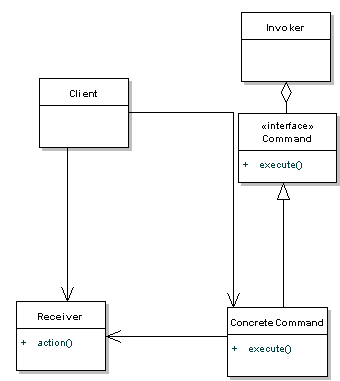
javax.lang.model.element.AnnotationValue obviously use Visitor pattern, but it is not commonly used in regular projects.

### The Command Pattern

The Command pattern is known as a **behavioural**pattern,as it's used to manage algorithms, relationships and responsibilities between objects. Thedefinition of Command provided in the original Gang of Four book on DesignPatterns states:

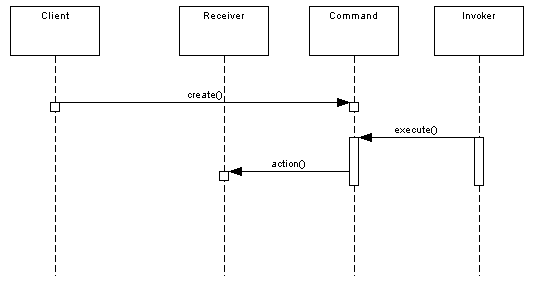
**Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations**

So what does this mean in a class diagram?



**Command** declares an interface for all commands, providing a simple **execute()**method which asks the **Receiver**of the command to carry out an operation. The **Receiver**has the knowledge of what to do to carry out the request.  The **Invoker**holds a command and can get the **Command**to execute a request by calling the execute method. The **Client** creates **ConcreteCommand**s and sets a **Receiver**for the command. The **ConcreteCommand**defines a binding between the action and the receiver. When the **Invoker**calls execute the ConcreteCommand will run one or more actions on the Receiver.

The following sequence diagram shows the relationship in a clearer way:



If this all seems a bit confusing right now, hang on for the code example later on in the article.

### When Would I Use This Pattern?

The Command Pattern is useful when:

* A history of requests is needed
* You need callback functionality
* Requests need to be handled at variant times or in variant orders
* The invoker should be decoupled from the object handling the invocation.

You'll see command being used a lot when you need to have multiple undo operations, where a stack of the recently executed commands are maintained. To implement the undo, all you need to do is get the last Command in the stack and execute it's undo() method.

You'll also find Command useful for wizards, progress bars, GUI buttons and menu actions, and other transactional behaviour.

### So How Does It Work In Java?

Let's use a remote control as the example. Our remote is the center of home automation and can control everything. We'll just use a light as an example, that we can switch on or off, but we could add many more commands.