# Visitor Design Pattern Example

Last Modified: August 21, 2021

**Design patterns** are used to solve the problems which occur in a pattern, we all know that, right? Also we know that **behavioral design patterns** are design patterns that identify common communication patterns between objects. One of such behavioral patterns is visitor pattern, which we are going to learn about in this post.

If you have been in working on a application which manage plenty of products, then you can easily relate with this problem:

> "You need to add a new method to a hierarchy of classes, but the act of adding it might be painful or damaging to the design."

So clearly, you **want a hierarchy of objects to modify their behavior but without modifying their source code**. How to do that? To solve this problem, visitor pattern comes into picture.

**Sections in this post:**

## Visitor pattern introduction

According to Wikipedia, the **visitor design pattern** is a way of separating an 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** (one of **SOLID design principles**).

Above design flexibility allows to add methods to any object hierarchy without modifying the code written for hierarchy. Rather **double dispatch** mechanism is used to implement this facility. Double dispatch is a special mechanism that dispatches a function call to different concrete functions depending on the runtime types of two objects involved in the call.

## Design participants/components

The participants 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.

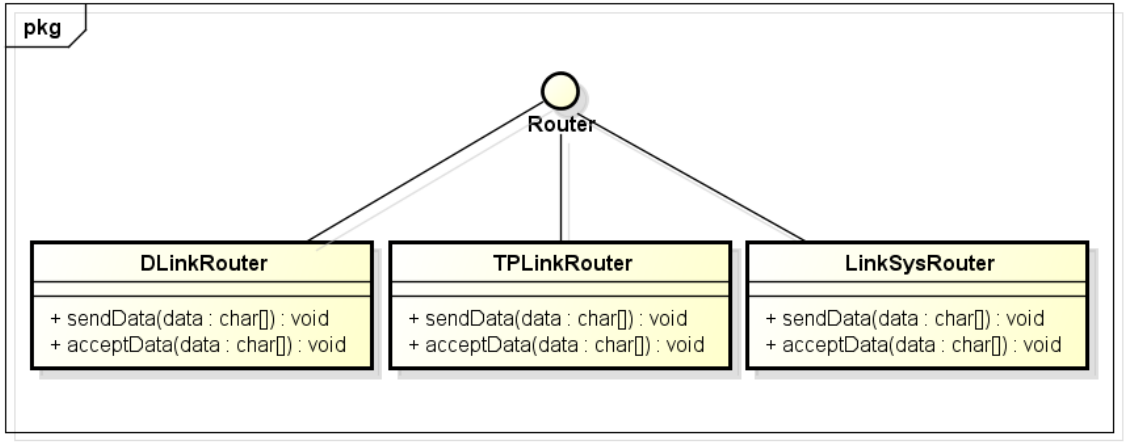
**ConcreteVisitable** – Those classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

## Sample problem to solve

An example is always better than long theory. So let's have one here also for visitor design pattern.

Suppose we have an application which manage routers in different environments. Routers should be capable of sending and receiving char data from other nodes and application should be capable of configuring routers in different environment.

Essentially, design should flexible enough to support the changes in way that **routers can be configured for additional environments in future, without much modifications in source code**.




**Existing routers in application**

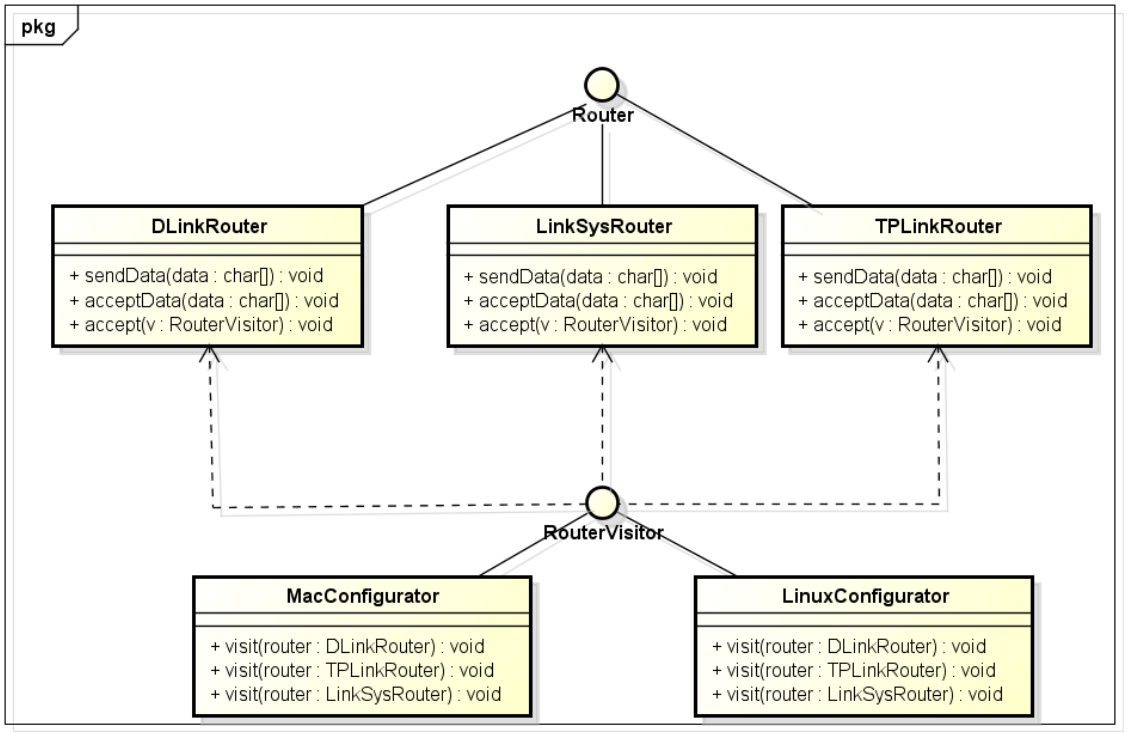
We have above 3 types of routers and we need to write code for them. **One way to solve this problem**, is to define methods like configureForWinodws() and configureForLinux() in Router.java interface and implement them in different products, cause each will have it's own configuration setting and procedure.

But the problem with above approach is that each time a new environment is introduced, whole router's hierarchy will have to be compiled again. Not acceptable solution. So what is it which can prevent this situation?

## Proposed solution using Visitor pattern

Visitor pattern is good fit for these types of problems where you want to introduce a new operation to hierarchy of objects, without changing its structure or modifying them. In this solution, we will implement **double dispatch technique by introducing two methods i.e. accept() and visit() methods**. Method accept(), will be defined in router's hierarchy and and visit methods will be on visitors level.

Whenever a new environment need to be added, a new visitor will be added into visitors hierarchy and that needs to implement visit() method for all the available routers and that's all.




**Solution using visitor pattern**

In above class diagram, we have routers configured for Mac and Linux operating systems. If we need to add the capability for windows also, then I do not need to change any class, just define a new visitor WindowsConfigurator and implement the visit() methods defined in RouterVisitor interface. It will provide the desired functionality without any further modification.

## Implementation code

Let's look at the source code of different files involved into above discussed problem and solution.

### Router.java

```
public interface Router
{
    public void sendData(char[] data);
    public void acceptData(char[] data);

    public void accept(RouterVisitor v);
}
```

### DLinkRouter.java

```
public class DLinkRouter implements Router{

    @Override
    public void sendData(char[] data) {
    }

    @Override
    public void acceptData(char[] data) {
    }

    @Override
    public void accept(RouterVisitor v) {
        v.visit(this);
    }
}
```

### LinkSysRouter.java

```
public class LinkSysRouter implements Router{
```

```
    @Override
    public void sendData(char[] data) {
    }

    @Override
    public void acceptData(char[] data) {
    }

    @Override
    public void accept(RouterVisitor v) {
        v.visit(this);
    }
}
```

### TPLinkRouter.java

```
public class TPLinkRouter implements Router{

    @Override
    public void sendData(char[] data) {
    }

    @Override
    public void acceptData(char[] data) {
    }

    @Override
    public void accept(RouterVisitor v) {
        v.visit(this);
    }
```

```
}
```

### RouterVisitor.java

```
public interface RouterVisitor {
    public void visit(DLinkRouter router);
    public void visit(TPLinkRouter router);
    public void visit(LinkSysRouter router);
}
```

### LinuxConfigurator.java

```
public class LinuxConfigurator implements RouterVisitor{

    @Override
    public void visit(DLinkRouter router) {
        System.out.println("DLinkRouter Configuration for Linux complete !!");
    }

    @Override
    public void visit(TPLinkRouter router) {
        System.out.println("TPLinkRouter Configuration for Linux complete !!");
    }

    @Override
    public void visit(LinkSysRouter router) {
        System.out.println("LinkSysRouter Configuration for Linux complete !!");
    }
}
```

**MacConfigurator.java**

```
public class MacConfigurator implements RouterVisitor{

    @Override
    public void visit(DLinkRouter router) {
        System.out.println("DLinkRouter Configuration for Mac complete !!");
    }

    @Override
    public void visit(TPLinkRouter router) {
        System.out.println("TPLinkRouter Configuration for Mac complete !!");
    }

    @Override
    public void visit(LinkSysRouter router) {
        System.out.println("LinkSysRouter Configuration for Mac complete !!");
    }
}
```

## How to use visitors in application code

To use above design, use the visitors in below given way. I have used the code in form of JUNIT testcases, you can change the code the way you feel correct into your case.

**TestVisitorPattern.java**

```
public class TestVisitorPattern extends TestCase
{
    private MacConfigurator macConfigurator;
```

```
    private LinuxConfigurator linuxConfigurator;
    private DLinkRouter dlink;
    private TPLinkRouter tplink;
    private LinkSysRouter linksys;

    public void setUp()
    {
        macConfigurator = new MacConfigurator();
        linuxConfigurator = new LinuxConfigurator();

        dlink = new DLinkRouter();
        tplink = new TPLinkRouter();
        linksys = new LinkSysRouter();
    }

    public void testDlink()
    {
        dlink.accept(macConfigurator);
        dlink.accept(linuxConfigurator);
    }

    public void testTPLink()
    {
        tplink.accept(macConfigurator);
        tplink.accept(linuxConfigurator);
    }

    public void testLinkSys()
    {
        linksys.accept(macConfigurator);
        linksys.accept(linuxConfigurator);
    }
}
```

```
Output:

DLinkRouter Configuration for Mac complete !!
DLinkRouter Configuration for Linux complete !!
LinkSysRouter Configuration for Mac complete !!
LinkSysRouter Configuration for Linux complete !!
TPLinkRouter Configuration for Mac complete !!
TPLinkRouter Configuration for Linux complete !!
```

## Sourcecode download

To download the sourcecode of above application, follow given link.

**Souececode download**

**Happy Learning !!**

**Share this:**

Twitter Facebook LinkedIn Reddit WhatsApp

## About Lokesh Gupta

A family guy with fun loving nature. Love computers, programming and solving everyday problems. Find me on Facebook and Twitter.

# Feedback, Discussion and Comments

Rishabh

April 6, 2018

what is TestCase class here and where it is create and it belongs to which package..

Muhammad

January 16, 2016

Nice talk Everybody! please i don’t know if somebody can help me to explain the Visitor design pattern.

Mladen

July 17, 2015

Just a thought. In Java 8 you could use default method in Router interface, so that you don't have to repeat implementation on all Router implementations.

```
public interface Router
{
public void sendData(char[] data);
public void acceptData(char[] data);

default public void accept(RouterVisitor v) {
v.visit(this);
}
}
```

Lokesh Gupta

July 17, 2015

Awesome thought !! A really useful update. Thanks for suggesting. But at this point, I would not like to update the existing source because Java 8 still is not adopted in most of the organizations for production (in my knowledge).

Marving

July 12, 2017

Nice post. Many Thanks.
One question. About comment Mladen

```
public interface Router
. . . . . .
default public void accept(RouterVisitor v)
{
v.visit(this); // at this moment this is type of is Router
}
```

And RouterVisitor hast NOT method visit(Router router)

```
public interface RouterVisitor {
public void visit(DLinkRouter router);
public void visit(TPLinkRouter router);
public void visit(LinkSysRouter router);
}
```

DLinkRouter is a Router
but
Router NOT is a DlinkRouter
NOT is a TPLinkRouter
NOT is a LinkSysRouter

What do you think about?

Rajendra J

May 14, 2015

Hi Lokesh,

I like all your topic discussions,
I want to know template pattern in detail (am writing here as it is also comes under behavioural pattern ).
Please discuss about 'Template pattern' if you have some time.
Thanks in advance.

Lokesh Gupta

May 14, 2015

Right now, I am very busy in my other projects. I will soon update on this.

java luver

February 27, 2015

looked around for good explanation/usecase of visitor design pattern. I must admit this is far the best example/explanation. Kudos to the author!

Vishi

February 24, 2014

RouterVisitor should take Router interface as dependency. You are using the concrete Router classes which violates the SOLID DI principle..

so RouterVisitor should have only 1 method visit(Router router)

In Each Concrete Configurator class, we can implement visit method with Reflection (for all RouterVisitor types)

viru

November 3, 2013

There is no main method and TestCase class..

Lokesh Gupta

November 4, 2013

```
public class TestVisitorPattern extends TestCase
```

Loganathan

October 24, 2013

If we introduced new router, again we need to modify all the visitor right since visit() method is getting Visitable class not interface. Again this will make big issue in OPEN/CLOSED principle. why should not we use interface and print the message based on the object type. please correct me if i m wrong.

Lokesh Gupta

October 25, 2013

Yes you are right. New router will have to be configured for each environment and it will change lots of classes, but isn't it how it should be done?? It forces the developer to handle new router in each supported environment.

Amit Das

July 31, 2019

Don't you think that if we don't use visitor here then also we are forcing the developer to handle new router in each supported environment.

Suresh

October 14, 2013

As per definition if we modify structure of any object that should not effect on others…in your example let i need to add one new method in DLinkRouter class then how we are going to handle that…?

> Lokesh Gupta
>
> October 14, 2013
>
> Definition is “add a new method to a hierarchy of classes”. Not for just a single class.

sasidhar

September 9, 2013

your explanation is simple and everybody will understand. Thanks!

Vitaly

September 9, 2013

In the picture you've got 2 LinkSys routers and no DLink

Lokesh Gupta

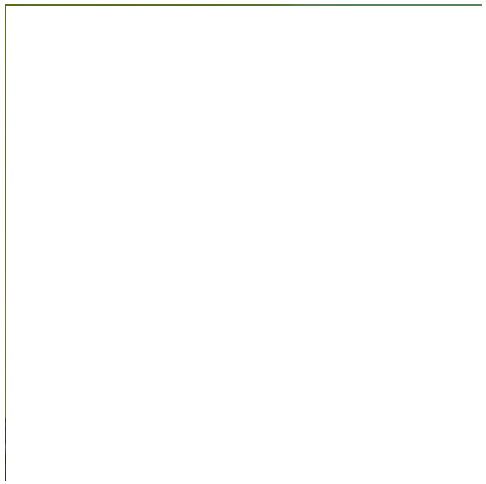
September 9, 2013

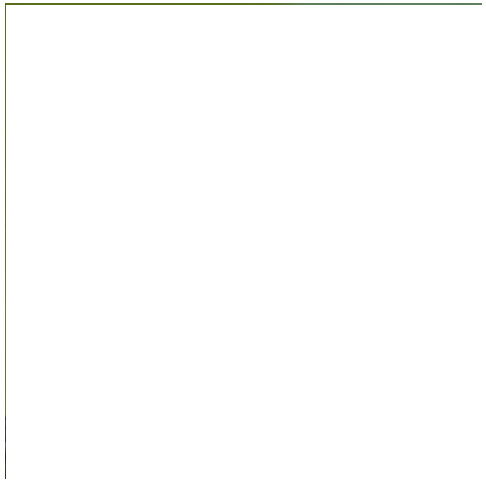
Corrected it.

## Comments are closed on this article!

### Search Tutorials

Type and Press ENTER...





## Meta Links

About Me

Contact Us

Privacy policy

Advertise

Guest and Sponsored Posts

## Recommended Reading

10 Life Lessons

Secure Hash Algorithms

How Web Servers work?

How Java I/O Works Internally?

Best Way to Learn Java

Java Best Practices Guide

Microservices Tutorial

REST API Tutorial

How to Start New Blog