



Behavioral Design Patterns: Visitor

by Emmanouil Gkatzouras MVB · Feb. 06, 19 · Java Zone · Tutorial

Delivering modern software? Atomist automates your software delivery experience.

Our last pattern of the behavioral design patterns is going to be the visitor pattern.

We use the visitor pattern when we want to make it possible to define a new operation for classes of an object structure without changing the classes.

Imagine the scenario of software that executes HTTP requests to an API. Most HTTP APIs out there have certain limits and allow a specific number of requests to be executed per minute. We might have a different class that executes requests and also takes into consideration the business logic with regards to the APIs that they interact.

In case we want to inspect those calls and print some information or persist request related information to the database, the visitor pattern could be a good fit.

We will start with the visitor interface.

```
1 package com.gkatzouras.design.behavioural.visitor;
2
3 public interface Visitor {
4 }
```

This interface will not specify any methods; however, interfaces that extend it will contain methods visit with specific types to visit. We do this in order to be able to have loosely-coupled visitor implementations (or even composition based visitors).

Then, we shall implement the visitable interface.

```
1 package com.gkatzouras.design.behavioural.visitor;
2
3 public interface Visitable {
4
5     void accept(T visitor);
6
7 }
```

Based on the above, we shall create our request execution classes, which are visitable.

```
1 package com.gkatzioura.design.behavioural.visitor;
2
3 public class LocationRequestExecutor implements Visitable {
4
5     private int successfulRequests = 0;
6     private double requestsPerMinute = 0.0;
7
8     public void executeRequest() {
9         /**
10          * Execute the request and change the successfulRequests and requestsPerMi
11          */
12     }
13
14     @Override
15     public void accept(LocationVisitor visitor) {
16         visitor.visit(this);
17     }
18
19     public int getSuccessfulRequests() {
20         return successfulRequests;
21     }
22
23     public double getRequestsPerMinute() {
24         return requestsPerMinute;
25     }
26
27 }
```

```
1 package com.gkatzioura.design.behavioural.visitor;
2
3 public class RouteRequestExecutor implements Visitable {
4
5     private int successfulRequests = 0;
6     private double requestsPerMinute = 0.0;
7
8     public void executeRequest() {
9         /**
10          * Execute the request and change the successfulRequests and requestsPerMi
11          */
12     }
13
14     @Override
15     public void accept(RouteVisitor visitor) {
16         visitor.visit(this);
17     }
18 }
```

```
17     }
18
19     public int getSuccessfulRequests() {
20         return successfulRequests;
21     }
22
23     public double getRequestsPerMinute() {
24         return requestsPerMinute;
25     }
26 }
```

And then, we shall add the visitor interfaces for these type of executors

```
1 package com.gkatzioura.design.behavioural.visitor;
2
3 public interface LocationVisitor extends Visitor {
4
5     void visit(LocationRequestExecutor locationRequestExecutor);
6 }
```

```
1 package com.gkatzioura.design.behavioural.visitor;
2
3 public interface RouteVisitor extends Visitor {
4
5     void visit(RouteRequestExecutor routeRequestExecutor);
6 }
```

The last step would be to create a visitor that implements the above interfaces.

```
1 package com.gkatzioura.design.behavioural.visitor;
2
3 public class RequestVisitor implements LocationVisitor, RouteVisitor {
4
5     @Override
6     public void visit(LocationRequestExecutor locationRequestExecutor) {
7
8     }
9
10    @Override
11    public void visit(RouteRequestExecutor routeRequestExecutor) {
12
13    }
14 }
```

So, let's put em all together.

```
1 package com.gkatzioura.design.behavioural.visitor;
2
3 public class VisitorMain {
4
5     public static void main(String[] args) {
6         final LocationRequestExecutor locationRequestExecutor = new LocationRequestExecutor();
7         final RouteRequestExecutor routeRequestExecutor = new RouteRequestExecutor();
8         final RequestVisitor requestVisitor = new RequestVisitor();
9
10        locationRequestExecutor.accept(requestVisitor);
11        routeRequestExecutor.accept(requestVisitor);
12    }
13 }
```

That's it! You can find the source code on [GitHub](#).

Start automating your delivery right there on your own laptop, today! Get the open source Atomist Software Delivery Machine.

Like This Article? Read More From DZone



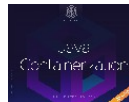
**Behavioral Design Patterns:
Observer**



**Observer Design Pattern in a
Nutshell**




**Behavioral Design Patterns:
Interpreter**



**Free DZone Refcard
Java Containerization**

Topics: [JAVA](#) , [BEHAVIORAL DESIGN PATTERNS](#) , [DESIGN PATTERNS](#) , [VISITOR](#) , [VISITOR DESIGN PATTERN](#) , [TUTORIAL](#) , [DESIGN PATTERN TUTORIAL](#) , [SAMPLE CODE](#)

Published at DZone with permission of Emmanouil Gkatziouras , DZone MVB. [See the original article here.](#) 

Opinions expressed by DZone contributors are their own.

Java Partner Resources

Gateway to Data Driven Operation & Digital Transformation

data

|

Learn more about Kotlin

testBrains

|

Advanced Linux Commands [Cheat Sheet]

Red Hat Developer Program

|

Streamline and Automate Your Local Development Flow

DevOps

|