Visitor in C++

Back to **Visitor** description

Visitor design pattern

1. Add an `accept(Visitor)` method to the "element" hierarchy
2. Create a "visitor" base class w/ a `visit()` method for every "element" type
3. Create a "visitor" derived class for each "operation" to do on "elements"
4. Client creates "visitor" objects and passes each to `accept()` calls

```cpp
#include <iostream>
#include <string>
using namespace std;

// 1. Add an accept(Visitor) method to the "element" hierarchy
class Element
{
  public:
    virtual void accept(class Visitor &v) = 0;
};

class This: public Element
{
  public:
     /*virtual*/void accept(Visitor &v);
    string thiss()
    {
        return "This";
    }
};

class That: public Element
{
  public:
     /*virtual*/void accept(Visitor &v);
    string that()
    {
        return "That";
    }
};

class TheOther: public Element
{
  public:
     /*virtual*/void accept(Visitor &v);
    string theOther()
    {
        return "TheOther";
    }
};

// 2. Create a "visitor" base class w/ a visit() method for every "element" type
class Visitor
{
  public:
    virtual void visit(This *e) = 0;
    virtual void visit(That *e) = 0;
    virtual void visit(TheOther *e) = 0;
```

```
};

 /*virtual*/void This::accept(Visitor &v)
{
  v.visit(this);
}

 /*virtual*/void That::accept(Visitor &v)
{
  v.visit(this);
}

 /*virtual*/void TheOther::accept(Visitor &v)
{
  v.visit(this);
}

// 3. Create a "visitor" derived class for each "operation" to do on "elements"
class UpVisitor: public Visitor
{
     /*virtual*/void visit(This *e)
    {
        cout << "do Up on " + e->thiss() << '\n';
    }
     /*virtual*/void visit(That *e)
    {
        cout << "do Up on " + e->that() << '\n';
    }
     /*virtual*/void visit(TheOther *e)
    {
        cout << "do Up on " + e->theOther() << '\n';
    }
};

class DownVisitor: public Visitor
{
     /*virtual*/void visit(This *e)
    {
        cout << "do Down on " + e->thiss() << '\n';
    }
     /*virtual*/void visit(That *e)
    {
        cout << "do Down on " + e->that() << '\n';
    }
     /*virtual*/void visit(TheOther *e)
    {
        cout << "do Down on " + e->theOther() << '\n';
    }
};
```

```
int main()
{
  Element *list[] =
  {
    new This(), new That(), new TheOther()
  };
  UpVisitor up; // 4. Client creates
  DownVisitor down; //    "visitor" objects
  for (int i = 0; i < 3; i++)
  //    and passes each
    list[i]->accept(up);
  //    to accept() calls
  for (i = 0; i < 3; i++)
    list[i]->accept(down);
}
```

Output

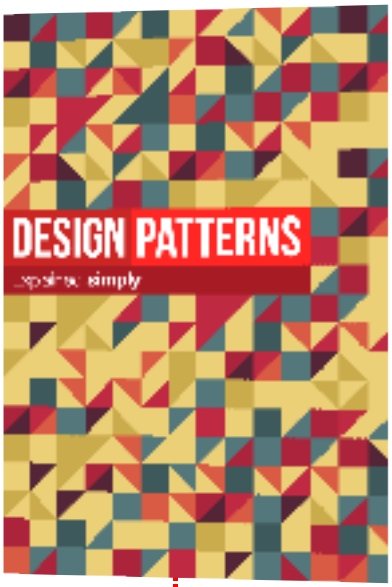
```
do Up on This               do Down on This
do Up on That               do Down on That
do Up on TheOther           do Down on TheOther
```

Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.



Learn more

Code examples

Java	**Visitor in Java**	**Visitor in Java: Double dispatch (within a single hierarchy)**	**Visitor in Java**	**Visitor in Java**
C++	**Visitor in C++: Before and after**	**Visitor in C++**	**Visitor in C++: Recovering lost type information**	
PHP	**Visitor in PHP**			
Delphi	**Visitor in Delphi**			

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us