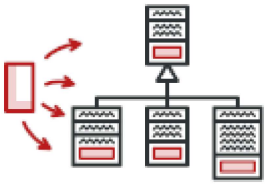

Visitor in C++

Visitor is a behavioral design pattern that allows adding new behaviors to existing class hierarchy without altering any existing code.

Read why Visitors can't be simply replaced with method overloading in our article **Visitor and Double Dispatch**.

Learn more about Visitor

Complexity:

Popularity:

Usage examples: Visitor isn't a very common pattern because of its complexity and narrow applicability.

Conceptual Example

This example illustrates the structure of the **Visitor** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```
/**
 * The Visitor Interface declares a set of visiting methods that correspond to
 * component classes. The signature of a visiting method allows the visitor to
 * identify the exact class of the component that it's dealing with.
 */
```

```
class ConcreteComponentA;
class ConcreteComponentB;

class Visitor {
 public:
  virtual void VisitConcreteComponentA(const ConcreteComponentA *element) const = 0;
  virtual void VisitConcreteComponentB(const ConcreteComponentB *element) const = 0;
};

/**
 * The Component interface declares an `accept` method that should take the base
 * visitor interface as an argument.
 */

class Component {
 public:
  virtual ~Component() {}
  virtual void Accept(Visitor *visitor) const = 0;
};

/**
 * Each Concrete Component must implement the `Accept` method in such a way that
 * it calls the visitor's method corresponding to the component's class.
 */
class ConcreteComponentA : public Component {
  /**
   * Note that we're calling `visitConcreteComponentA`, which matches the
   * current class name. This way we let the visitor know the class of the
   * component it works with.
   */
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentA(this);
  }
  /**
   * Concrete Components may have special methods that don't exist in their base
   * class or interface. The Visitor is still able to use these methods since
   * it's aware of the component's concrete class.
   */
  std::string ExclusiveMethodOfConcreteComponentA() const {
    return "A";
  }
};

class ConcreteComponentB : public Component {
  /**
   * Same here: visitConcreteComponentB => ConcreteComponentB
   */
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentB(this);
  }
```

```
  std::string SpecialMethodOfConcreteComponentB() const {
    return "B";
  }
};

/**
 * Concrete Visitors implement several versions of the same algorithm, which can
 * work with all concrete component classes.
 *
 * You can experience the biggest benefit of the Visitor pattern when using it
 * with a complex object structure, such as a Composite tree. In this case, it
 * might be helpful to store some intermediate state of the algorithm while
 * executing visitor's methods over various objects of the structure.
 */
class ConcreteVisitor1 : public Visitor {
 public:
  void VisitConcreteComponentA(const ConcreteComponentA *element) const override {
    std::cout << element->ExclusiveMethodOfConcreteComponentA() << " + ConcreteVisitor1\n";
  }

  void VisitConcreteComponentB(const ConcreteComponentB *element) const override {
    std::cout << element->SpecialMethodOfConcreteComponentB() << " + ConcreteVisitor1\n";
  }
};

class ConcreteVisitor2 : public Visitor {
 public:
  void VisitConcreteComponentA(const ConcreteComponentA *element) const override {
    std::cout << element->ExclusiveMethodOfConcreteComponentA() << " + ConcreteVisitor2\n";
  }
  void VisitConcreteComponentB(const ConcreteComponentB *element) const override {
    std::cout << element->SpecialMethodOfConcreteComponentB() << " + ConcreteVisitor2\n";
  }
};
/**
 * The client code can run visitor operations over any set of elements without
 * figuring out their concrete classes. The accept operation directs a call to
 * the appropriate operation in the visitor object.
 */
void ClientCode(std::array<const Component *, 2> components, Visitor *visitor) {
  // ...
  for (const Component *comp : components) {
    comp->Accept(visitor);
  }
  // ...
}

int main() {
  std::array<const Component *, 2> components = {new ConcreteComponentA, new ConcreteComponen
  std::cout << "The client code works with all visitors via the base Visitor interface:\n";
  ConcreteVisitor1 *visitor1 = new ConcreteVisitor1;
  ClientCode(components, visitor1);
```

```
  std::cout << "\n";
  std::cout << "It allows the same client code to work with different types of visitors:\n";
  ConcreteVisitor2 *visitor2 = new ConcreteVisitor2;
  ClientCode(components, visitor2);

  for (const Component *comp : components) {
    delete comp;
  }
  delete visitor1;
  delete visitor2;

  return 0;
}
```

Output.txt: Execution result

```
The client code works with all visitors via the base Visitor interface:
A + ConcreteVisitor1
B + ConcreteVisitor1

It allows the same client code to work with different types of visitors:
A + ConcreteVisitor2
B + ConcreteVisitor2
```