## Visitor

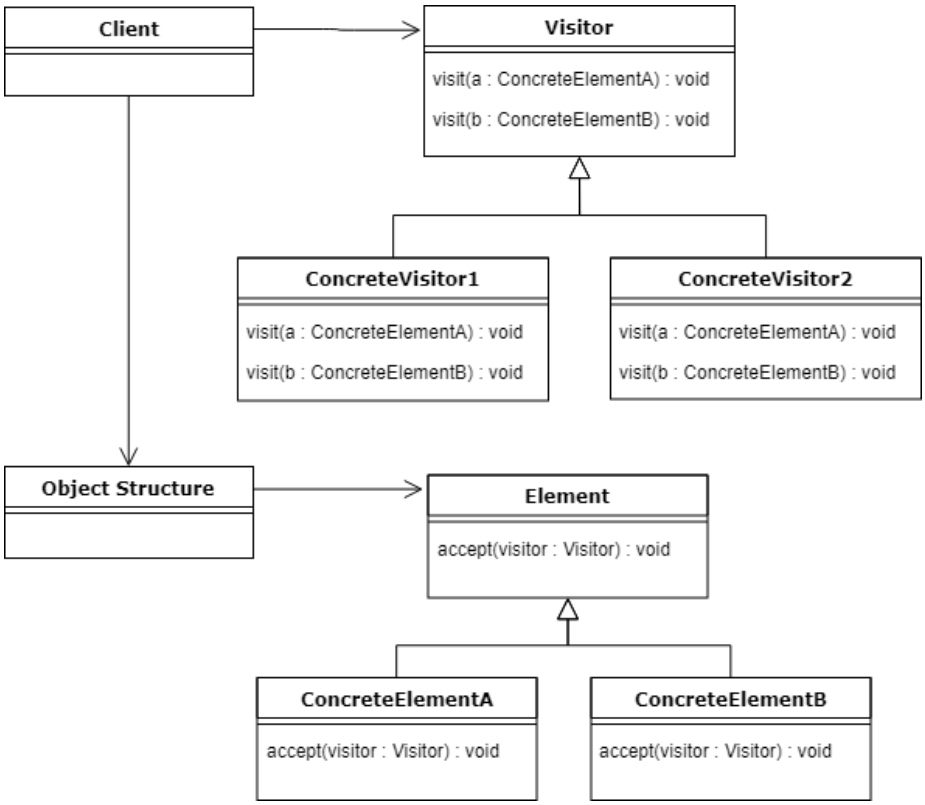


Figure 1: "UML of Visitor Design Pattern"

### Intent

1. define a new operation to be performed on the elements of an **object structure**, without changing the classes of the **elements**.
2. **define two class hierarchies**:
   - one for the **elements** being operated on, contained in the **object structure** and
   - one for the **visitors** defining **operations** on the **elements**.
3. defining a new operation consists of subclassing the **visitor** hierarchy.

### Applicability

1. when an **object structure** contains many classes of **elements** with differing interfaces, upon which we want to perform **operations** that depend

on their **concrete classes**.
2. when distinct and unrelated operations need to be performed on **elements** contained in an **object structure**, and we want to avoid polluting their classes with these operations, we can define these operations in a separate **visitor hierarchy** and use them only in applications using the **object structure** and requiring these operations.
3. when the **element hierarchy rarely changes**, but we often want to define **new operations** over it.

### Participants

1. `Visitor` (*abstract class/interface*):
   - defines the **root visitor** of the **visitor hierarchy**.
   - declares a `visit()` **operation** for each `ConcreteElement` class in the **element hierarchy**.
   - the **operation's signature** identifies the `ConcreteElement`'s class that **accepts** the **visit** request from the **visitor**.
   - the **visitor** can **access** the `ConcreteElement` **directly** through its particular interface.
2. `ConcreteVisitor` (*concrete class*):
   - implements each operation declared by `Visitor`.
   - each operation implements a **fragment** of the **algorithm** defined for the corresponding `ConcreteElement` in the **element hierarchy**.
   - provides the **context** for the **algorithm** and stores its **local state**, accumulating usually during the **object structure's elements traversal**.
3. `Element` (*abstract class/interface*):
   - defines the **root element** of the **element hierarchy**.
   - declares an `accept()` **operation** that takes a `Visitor` as an **argument**.
4. `ConcreteElement`(*concrete class*): implements the `accept()` **operation** according to how it should be **visited** and by which `ConcreteVisitor`.
5. `ObjectStructure`:
   - a structure that can **enumerate** its **elements**.
   - may provide a high-level interface to allow the `Visitor` to **visit** its **elements**.
   - may be a **composite** object (*cf.* **Composite Design Pattern**) or a **collection** (*e.g. list, set, ...*)

### Collaborations and UML interaction diagram

1. a **client** must create a `ConcreteVisitor` and then **traverse** the **object structure**, **visiting** each `ConcreteElement` therein.

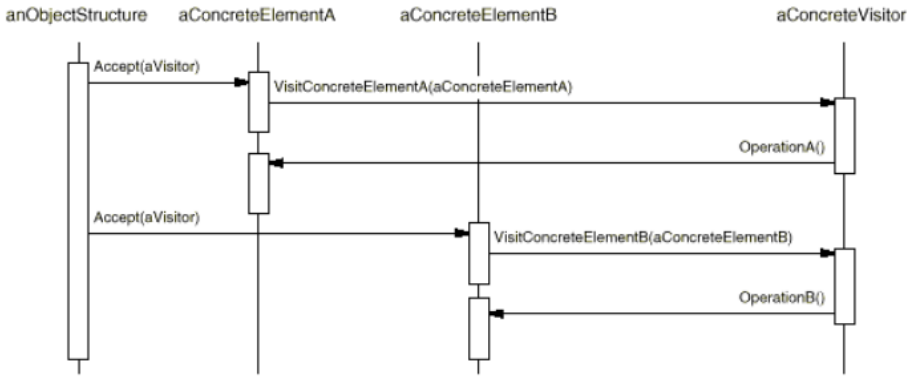


Figure 2: "UML Interaction Diagram of Visitor Design Pattern"

2. when a `ConcreteElement` **accepts** the **visit** , it calls the `visit()` **operation** that corresponds to its class.
3. the `ConcreteElement` supplies itself as an **argument** to `visit()` operation to let the `ConcreteVisitor` access its **state**, if necessary.

**Pros**

1. **adding new operations is easy**: simply adding a **new** `ConcreteVisitor` class to the **visitor hierarchy** and implementing the `visit()` operation for each `ConcreteElement` class in the **object structure's elements hierarchy**.
2. **gathering related operations and separating unrelated ones**:
   - **related behavior** isn't spread over the `ConcreteElement` classes of the **object structure's elements hierarchy**, but rather **localized** in a single `ConcreteVisitor` class defining the behavior for different `ConcreteElement` classes.
   - **unrelated sets of behavior** are **partitioned** across different `ConcreteVisitor` classes.
   - this simplifies the definition of `ConcreteElement` classes and the algorithms defining operations over them in the `ConcreteVisitor` classes, where algorithm-related data structures can be hidden in the `ConcreteVisitor` classes.
3. **visiting across class hierarchies**:
   - **iterator**: can **visit elements** in an **elements hierarchy**, but it cannot work across **element hierarchies** with elements having different types, and therefore **elements visited by an iterator** must all **share a common parent class**.
   - **visitor**: can **visit elements** that don't necessarily have a **common**

**parent class**, and therefore doesn't have the restrictions of an **iterator**.

4. **accumulating state**:
   - `ConcreteVisitor`s can **accumulate state** as they **visit** each `ConcreteElement` class in the **object structure's elements hierarchy**.
   - without a `ConcreteVisitor`, the **state** would have to either:
     1. be passed as **extra arguments** to the **operations** that perform the **traversal of the elements hierarchy**, or
     2. appear as **global variables**.

**Cons**

1. **adding new concrete elements to the object structure's elements hierarchy is hard**:
   - adding a new `ConcreteElement` class requires **declaring a new corresponding** `visit()` **operation** in `Visitor` which can:
     1. have a **default implementation** inherited by all `ConcreteVisitor` classes.
     2. be **abstract** and must be implemented accordingly by all `ConcreteVisitor` classes.
   - **consequence**: if new `ConcreteElement` classes are **constantly being added** to the **object structure's elements hierarchy**, the **visitors hierarchy** becomes **harder to maintain**, and it becomes **easier** just to **define operations directly** on the **object structure's elements hierarchy**.
2. **breaking encapsulation**: the `Visitor` design pattern approach assumes that the `ConcreteElement` interface **allows the visitors to do their job**, by providing **public operations** that access a `ConcreteElement`'s **internal state**, which may **compromise** its **encapsulation**.

**Implementation issues**

**`Visitor` uses the double-dispatch technique**

**Single dispatch**

1. **definition**: two criteria determine which **operation** will fulfill a **request** and therefore gets **executed**:
   - the **name** of the **request**;
   - the **type** of the **receiver**.
2. **example**: a `GenerateCode` **request** on a `VariableRefNode` **receiver** will **execute** `VariableRefNode::GenerateCode()`, whereas a

`GenerateCode` **request** on a `AssignmentNode` **receiver** will **execute** `AssignmentNode::GenerateCode()`.

### Double dispatch

1. **definition**: two criteria determine which **operation** will fulfill a **request** and therefore gets **executed**:
   - the **name** of the **request**;
   - the **types** of **two receivers**.
2. **example**: `accept()` is a **double-dispatch operation** since its meaning depends on **two types**: the `ConcreteVisitor` class and the `ConcreteElement` class it **visits**.
3. **pro**: instead of **statically binding an operation** to the `Element` **interface**, we can **consolidate** the **operations** in a `Visitor` and use `accept()` to do the **binding** at **run-time**.

### The object structure traversal responsibility

1. **principle**: since a `ConcreteVisitor` must **visit** each `ConcreteElement` in an **object structure's elements hierarchy**, the **traversal responsibility** can be delegated to any of the following:
   - the **object structure**
   - a **separate iterator object**, or
   - the `ConcreteVisitor`.
2. **traversal through the object structure**:
   - if **object structure** is a **collection**: call `accept()` **iteratively** on each `ConcreteElement` in the **elements hierarchy**.
   - if **object structure** is a **composite**: call `accept()` **recursively** on the **elements hierarchy**.
3. **traversal through a separate iterator object**: it's a lot like the **first case**, but it will **not cause double-dispatching** as the iterator will **invoke** `visit()` on the `ConcreteVisitor` with `ConcreteElement` as an **argument**, rather than invoking `accept()` on `ConcreteElement` with `ConcreteVisitor` as an **argument**.
4. **in the visitor**:
   - **pro**: create a **complex traversal** that depends on the results of the **operations** on the **object structure**.
   - **con**: **duplicate** the **traversal code** in each `ConcreteVisitor` for each **aggregate** `ConcreteElement`.

### Example

```
package behavioral.visitor;

/**
```

```java
 * a Product abstract class that plays the role of Element
 * in the Visitor Design pattern.<br/>
 * Product is the root class of the products hierarchy.
 * It implements the Visitable interface to allow its
 * subclasses to be visitable. The visitor accepting behavior
 * is implemented by its subclasses, each according to its type.
 * @author anonbnr
 */
public abstract class Product implements Visitable {
    /* ATTRIBUTES */
    /**
     * The product's name.
     */
    private String name;

    /**
     * The product's price.
     */
    private double price;

    /* CONSTRUCTORS */
    /**
     * Creates a Product named name having price.
     * @param name The name of the Product to create.
     * @param price The price of the Product to create.
     */
    public Product(String name, double price) {
        this.setName(name);
        this.setPrice(price);
    }

    /* METHODS */
    /**
     * Gets this Product's name.
     * @return this Product's name.
     */
    public String getName() {return this.name;}

    /**
     * Sets name as this Product's name.
     * @param name The value to set this Product's name.
     */
    public void setName(String name) {this.name = name;}

    /**
     * Gets this Product's price.
```

```
     * @return this Product's price.
     */
    public double getPrice() {return this.price;}

    /**
     * Sets price as this Product's price.
     * @param price The value to set this Product's price.
     */
    public void setPrice(double price) {this.price = price;}
}

package behavioral.visitor;

/**
 * a Liquor concrete class that plays the role of a ConcreteElement
 * in the Visitor Design pattern.<br/>
 * Liquor is a Visitable product that can be visited
 * by a ConcreteVisitor to implement a specific operation
 * on it without changing its structure.
 * @author anonbnr
 *
 */
public class Liquor extends Product {
    /* CONSTRUCTORS */
    /**
     * Creates a Liquor named name having price.
     * @param name The name of the Liquor to create.
     * @param price The price of the Liquor to create.
     */
    public Liquor(String name, double price) {
        super(name, price);
    }

    /* METHODS */
    /**
     * Allows visitor to visit this Liquor
     */
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

package behavioral.visitor;

/**
```

```
 * a Necessity concrete class that plays the role of a ConcreteElement
 * in the Visitor Design pattern.<br/>
 * Necessity is a Visitable Product that can be visited
 * by a ConcreteVisitor to implement a specific operation
 * on it without changing its structure.
 * @author anonbnr
 *
 */
public class Necessity extends Product {
    /* CONSTRUCTORS */
    /**
     * Creates a Necessity having price.
     * @param name The name of the Necessity to create.
     * @param price The price of the Necessity to create.
     */
    public Necessity(String name, double price) {
        super(name, price);
    }

    /* METHODS */
    /**
     * Allows visitor to visit this Necessity
     */
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

package behavioral.visitor;

/**
 * a Tobacco concrete class that plays the role of a ConcreteElement
 * in the Visitor Design pattern.<br/>
 * Tobacco is a Visitable product that can be visited
 * by a ConcreteVisitor to implement a specific operation
 * on it without changing its structure.
 * @author anonbnr
 *
 */
public class Tobacco extends Product {
    /* CONSTRUCTORS */
    /**
     * Creates a Tobacco having price.
     * @param name The name of the Tobacco to create.
     * @param price The price of the Tobacco to create.
```

```
     */
    public Tobacco(String name, double price) {
        super(name, price);
    }

    /* METHODS */
    /**
     * Allows visitor to visit this Tobacco
     */
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

package behavioral.visitor;
/**
 * a Visitable interface declaring the accept() method
 * in the Visitor Design pattern.<br/>
 * The interface could be used to specify different ways
 * of accepting visitors and should be implemented by
 * visitable objects.
 * @author anonbnr
 */
public interface Visitable {
    /* METHODS */
    /**
     * Allows a visitable object to accept visitor.
     * @param visitor A Visitor visiting this visitable object.
     */
    public void accept(Visitor visitor);
}

package behavioral.visitor;

/**
 * A Visitor interface that plays the role of Visitor
 * in the Visitor Design pattern.<br/>
 * It provides the visiting operations that
 * are to be implemented by all concrete visitors
 * for every concrete product in the visitable Products hierarchy.
 * @author anonbnr
 * @see Product
 */
public interface Visitor {
    /* METHODS */
```

```
    /**
     * Visits a Liquor.
     * @param liquor The Liquor to visit.
     */
    void visit(Liquor liquor);

    /**
     * Visits a Tobacco.
     * @param tobacco The Tobacco to visit.
     */
    void visit(Tobacco tobacco);

    /**
     * Visits a Necessity.
     * @param necessity The Necessity to visit.
     */
    void visit(Necessity necessity);
}
```

```
package behavioral.visitor;

/**
 * A TaxVisitor concrete class that plays the role
 * of a ConcreteVisitor in the Visitor design pattern.<br/>
 * It's a visitor which allows to compute taxes on different
 * concrete products, namely Liquor, Tobacco, and Necessity objects.
 * @author anonbnr
 */
public class TaxVisitor implements Visitor {

    /* ATTRIBUTES */
    /**
     * The current visited product's computed tax.
     */
    protected double computedTax;

    /**
     * The current visited product's tax ratio.
     */
    protected double taxRate;

    /* METHODS */
    /**
     * Gets the computed tax for the currently visited product
     * @return the computed tax for the currently visited product
     */
```

```java
public double getComputedTax() {
    return computedTax;
}

/**
 * Gets the taxRate for the currently visited product.
 * @return the taxRate for the currently visited product.
 */
public double getTaxRate() {
    return taxRate;
}

/**
 * Computes the tax for product using taxRate.
 * @param product The visited Product for which we wish
 * to compute the tax.
 * @param taxRate The tax rate used to compute the tax for
 * product.
 */
protected void computeTax(Product product) {
    System.out.println(product.getClass().getSimpleName() + " item: Price with Tax");
    computedTax = product.getPrice() * (1 + taxRate);
}

/**
 * Computes the tax for a Liquor product.
 */
@Override
public void visit(Liquor liquor) {
    taxRate = 0.18;
    computeTax(liquor);
}

/**
 * Computes the tax for a Tobacco product.
 */
@Override
public void visit(Tobacco tobacco) {
    taxRate = 0.32;
    computeTax(tobacco);
}

/**
 * Computes the tax for a Necessity product.
 */
@Override
```

```
    public void visit(Necessity necessity) {
        taxRate = 0;
        computeTax(necessity);
    }
}
```

```
package behavioral.visitor;

/**
 * A TaxHolidayVisitor concrete class that plays the role
 * of a ConcreteVisitor in the Visitor design pattern.<br/>
 * It's a visitor which allows to compute holiday taxes on different
 * concrete products, namely Liquor, Tobacco, and Necessity objects.
 * @author anonbnr
 */
public class TaxHolidayVisitor extends TaxVisitor {

    /* METHODS */
    @Override
    public void visit(Liquor liquor) {
        taxRate = 0.10;
        computeTax(liquor);
    }

    @Override
    public void visit(Tobacco tobacco) {
        taxRate = 0.30;
        computeTax(tobacco);
    }
}
```

```
package behavioral.visitor;

import java.util.Arrays;
import java.util.List;

/**
 * a Test class for the Visitor design pattern.
 * @author anonbnr
 */
public class Test {

    public static void main(String[] args) {
        List<Product> products = Arrays.asList(
                new Necessity("Milk", 3.47),
                new Liquor("Vodka", 11.99),
```

```
                new Tobacco("Cigar", 19.99)
        );

        System.out.println("Tax prices\n==========");
        computeTaxForProducts(products, new TaxVisitor());

        System.out.println("Holiday Tax prices\n==================");
        computeTaxForProducts(products, new TaxHolidayVisitor());
    }

    /**
     * Computes the tax for each product of products, using taxVisitor.
     * @param products The list of products whose taxes we wish to compute.
     * @param taxVisitor The TaxVisitor instance used to compute the tax for
     * each product in products.
     */
    private static void computeTaxForProducts(List<Product> products,
                TaxVisitor taxVisitor) {
        for (Product product: products) {
            product.accept(taxVisitor);
            System.out.println(taxVisitor.getComputedTax() + "\n");
        }
    }
}
```

**Output**

```
Tax prices
==========
Necessity item: Price with Tax
3.47

Liquor item: Price with Tax
14.1482

Tobacco item: Price with Tax
26.3868

Holiday Tax prices
==================
Necessity item: Price with Tax
3.47

Liquor item: Price with Tax
13.189000000000002
```

```
Tobacco item: Price with Tax
25.987
```