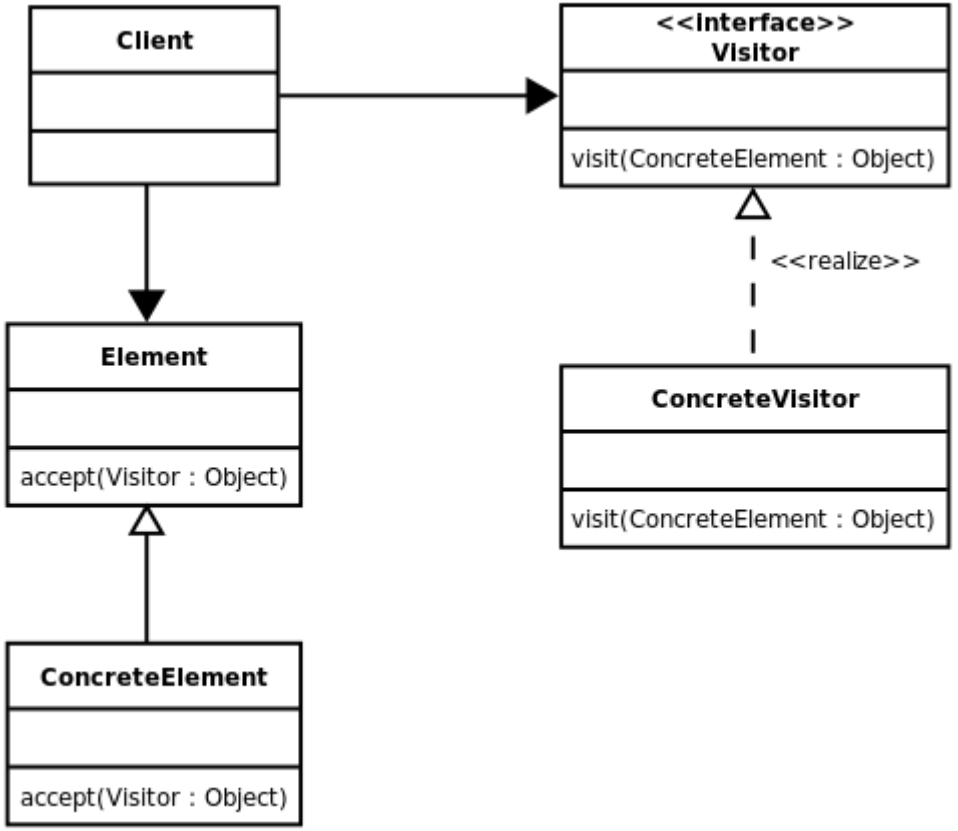
# Visitor


Used to traverse a Composite structure whose items may be completely different types and have completely different behaviors. It can add new functionality to the Composite without requiring behavior changes to the Composite itself. This is a direct application of the Open/Closed Principle.

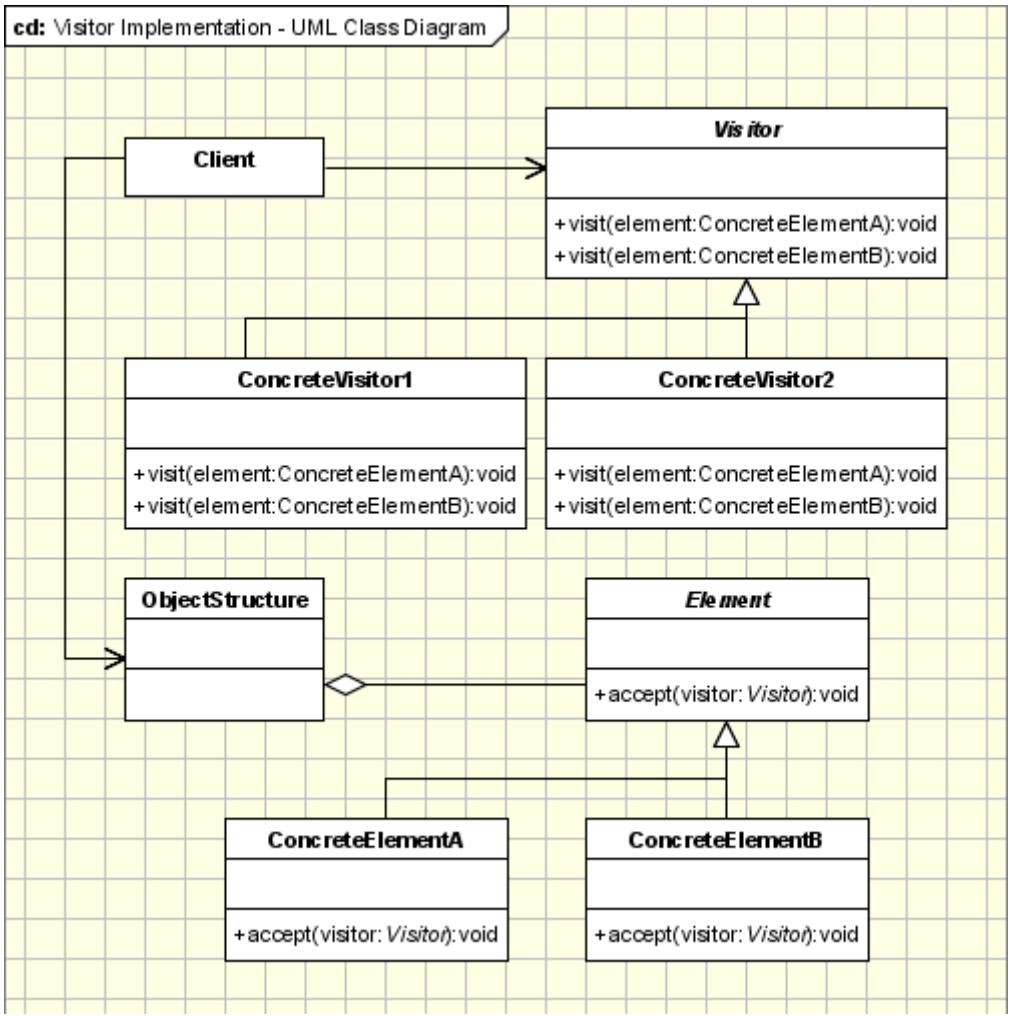
# Visitor

- Visitor is a behavioral pattern since it allows you to add new functionality to an existing Composite structure
- Considered to be one of the more complex patterns in use due to its structure
- Similar feel to Iterator, but Iterator is typically used to traverse a Composite that contains related types internally.
- Furthermore, Iterator does not typically add behavior to the Composite

# Visitor UML


Client
<<Interface>>
Visitor
visit(ConcreteElement : Object)
<<realize>>
Element
accept(Visitor : Object)
ConcreteVisitor
visit(ConcreteElement : Object)
ConcreteElement
accept(Visitor : Object)

# Second Visitor UML



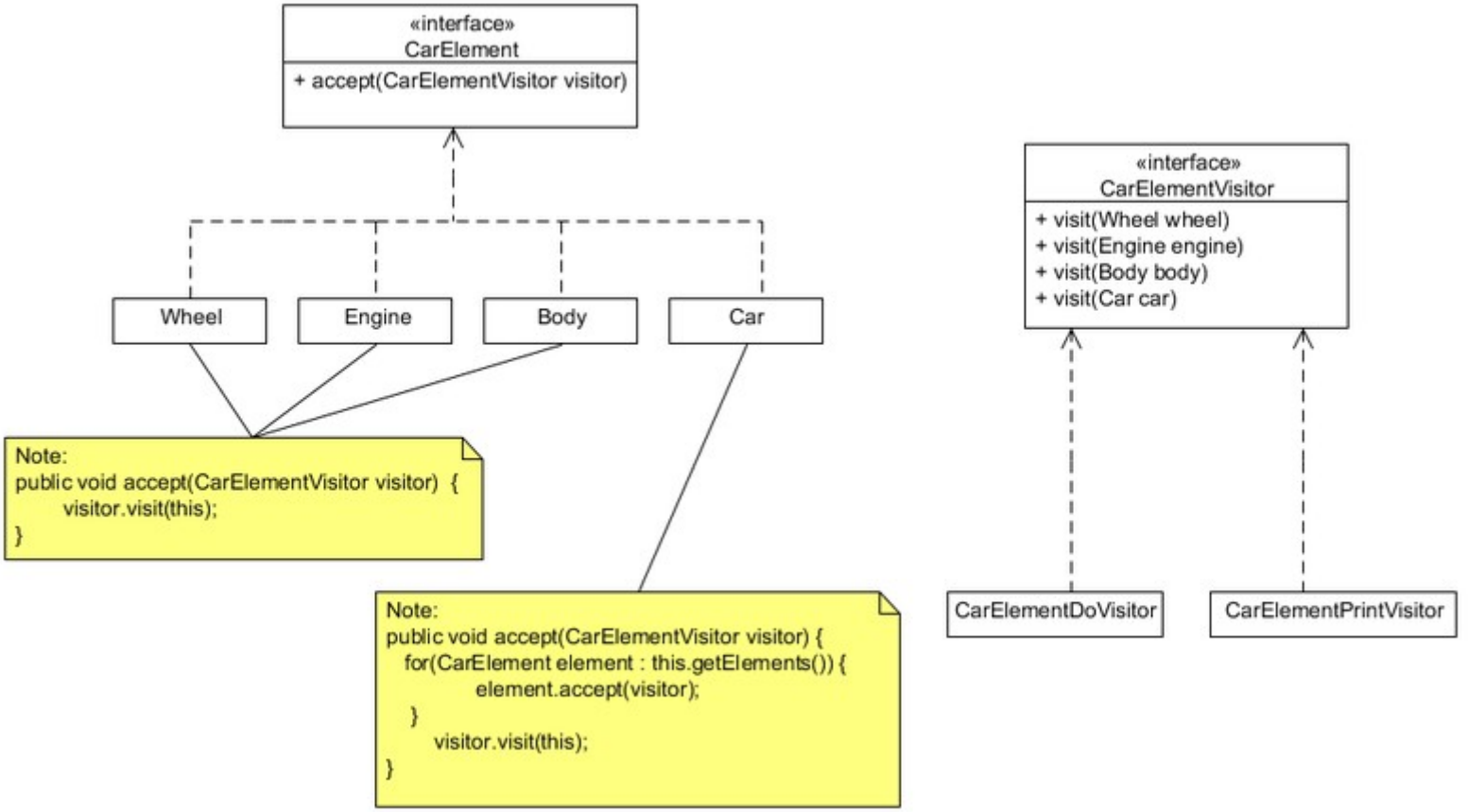
# UML Entities

- Visitor interface contains a list of methods that each Concrete Visitor implements. These methods are designed specifically with the underlying components of the ObjectStructure in mind. Generically the methods are named visit.
- The Element interface provides (typically) a single accept method that each ConcreteElement defines. The usual code in the implemented accept method is as follows: visitor.visit(this)
- NOTE: The fact that the elements allow access to an external entity violates encapsulation principles. That said, the visitor may still only access the public items presented by the element.

# Visitor Concerns

- Visitor does violate encapsulation principles, so you must be ok with allowing an external entity to work 'directly' with the underlying components (which are usually hidden from the outside world) of a Composite.
- If the Composite is updated to include another element, the Visitor interface and all ConcreteVisitor classes need updated. This shows there is coupling with this pattern. It is somewhat similar in nature to the coupling you have in the Bridge Pattern (if the abstraction changes, the implementations must be updated).

# Wikipedia Example



# Wikipedia Example Code

```
interface ICarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

interface ICarElement {
    void accept(ICarElementVisitor visitor); //
CarElements have to provide accept().
}
```

# Wikipedia Example Code

```
class Wheel implements ICarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
```

# Wikipedia Example Code

```
public void accept(ICarElementVisitor visitor) {
        /*
         * accept(ICarElementVisitor) in Wheel implements
         * accept(ICarElementVisitor) in ICarElement, so the call
         * to accept is bound at run time. This can be considered
         * the first dispatch. However, the decision to call
         * visit(Wheel) (as opposed to visit(Engine) etc.) can be
         * made during compile time since 'this' is known at compile
         * time to be a Wheel. Moreover, each implementation of
         * ICarElementVisitor implements the visit(Wheel), which is
         * another decision that is made at run time. This can be
         * considered the second dispatch.
         */
        visitor.visit(this);
    }
}
```

# Wikipedia Example Code

```
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

# Wikipedia Example Code

```
class Car implements ICarElement {
    ICarElement[] elements;

    public Car() {
        //create new Array of elements
        this.elements = new ICarElement[] { new Wheel("front left"),
            new Wheel("front right"), new Wheel("back left") ,
            new Wheel("back right"), new Body(), new Engine() };
    }

    public void accept(ICarElementVisitor visitor) {
        for(ICarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}
```

# Wikipedia Example Code

```
class CarElementPrintVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}
```

# Wikipedia Example Code

```
class CarElementDoVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + "
wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}
```

# Wikipedia Example Code

```
public class VisitorDemo {
    public static void main(String[] args) {
        ICarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

# Wikipedia Example Code: Output

```
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

# Concluding Thoughts

- Visitor helps enforce Single Responsibility Principle and allows Composite object it visits to just worry about its true purpose.
- Visitor applies the Open/Closed principle because the Composite does not have to be modified (other than during initial design accepting a Visitor).
- Visitors are objects so can have state. This is very useful in cases where the action performed on a given part of the Composite depends on previous actions which may have modified state.

# Other Thoughts

- Visitor solves the double dispatch problem
  - Single dispatch: implemented via virtual methods. All OOP languages support this.
  - Given this:

```
public class SpaceShip {
  public virtual string GetShipType() {
   return "SpaceShip";
  }
}

public class ApolloSpacecraft : SpaceShip {
  public virtual string GetShipType() {
    return "ApolloSpacecraft";
  }
}
```

# Other Thoughts

- Execute this

```
SpaceShip ship = new ApolloSpacecraft();
Console.WriteLine(ship.GetShipType());
```

- Output is: ApolloSpacecraft
- Method to execute is chosen at runtime based solely on the actual type of ship (underlying object)
- Double dispatch from above example (see next slides)

# Double Dispatch

- What if we add Asteroids?

```
public class Asteroid {
  public virtual void CollideWith(SpaceShip ship) {
    Console.WriteLine("Asteroid hit a SpaceShip");
  }
  public virtual void CollideWith(ApolloSpacecraft ship) {
    Console.WriteLine("Asteroid hit an ApolloSpacecraft");
  }
};

public class ExplodingAsteroid : Asteroid {
  public override void CollideWith(SpaceShip ship) {
    Console.WriteLine("ExplodingAsteroid hit a SpaceShip");
  }
  public override void CollideWith(ApolloSpacecraft ship) {
    Console.WriteLine("ExplodingAsteroid hit an ApolloSpacecraft");
  }
};
```

# Double Dispatch

Now add this code:

```
Asteroid theAsteroid = new Asteroid();
ExplodingAsteroid theExplodingAsteroid = new
ExplodingAsteroid();
SpaceShip theSpaceShip = new SpaceShip();
ApolloSpacecraft theApolloSpacecraft = new ApolloSpacecraft();
```

# Double Dispatch

And this code:

```
theAsteroid.CollideWith(theSpaceShip);
theAsteroid.CollideWith(theApolloSpacecraft);
theExplodingAsteroid.CollideWith(theSpaceShip);
theExplodingAsteroid.CollideWith(theApolloSpacecraft);
```

Which prints:

```
Asteroid hit a SpaceShip
Asteroid hit an ApolloSpacecraft
ExplodingAsteroid hit a SpaceShip
ExplodingAsteroid hit an ApolloSpacecraft
```

This is as expected

# Double Dispatch

Now let's do the following:

```
// Note the different data types!
Asteroid theExplodingAsteroidRef = new ExplodingAsteroid();
SpaceShip theApolloSpacecraftRef = new ApolloSpacecraft();
theExplodingAsteroidRef.CollideWith(theApolloSpacecraftRef);
```

What does it print? If you guessed 'ExplodingAsteroid hit an ApolloSpacecraft'

...

# Double Dispatch

- You guessed WRONG!
- The actual output is: ExplodingAsteroid hit a Spaceship
- The problem is that C# (and Java, C++, etc) only supports single dispatch. The method chosen in the above example is only based on theExplodingAsteroidRef, not theExplodingAsteroidRef and theApolloSpacecraftref (which would be double dispatch)
- Visitor solves this problem by allowing the ConcreteVisitor to specifically call the underlying method of the ConcreteElement

# Final Comments

- Visitor is a behavioral pattern that allows you to visit each element of a Composite structure.
- The elements visited can be very different, but the visitor is designed to deal with each in the desired way
- Visitor is coupled with the Composite on which it operates, but the tradeoff of this coupling and encapsulation violation is worth the gain in added functionality and in leaving the Composite itself untouched.

# The Pattern is much better than the Movie :-)