VISITOR PATTERN

Separating the algorithm from the elements

Inside A Complex System

Multiple Classes
Interaction between each classes
Each classes have multiple similar operations
Some have distinct operations
The needs to add/remove operations to classes on the run!

Could we make this more manageable ? less costly ?

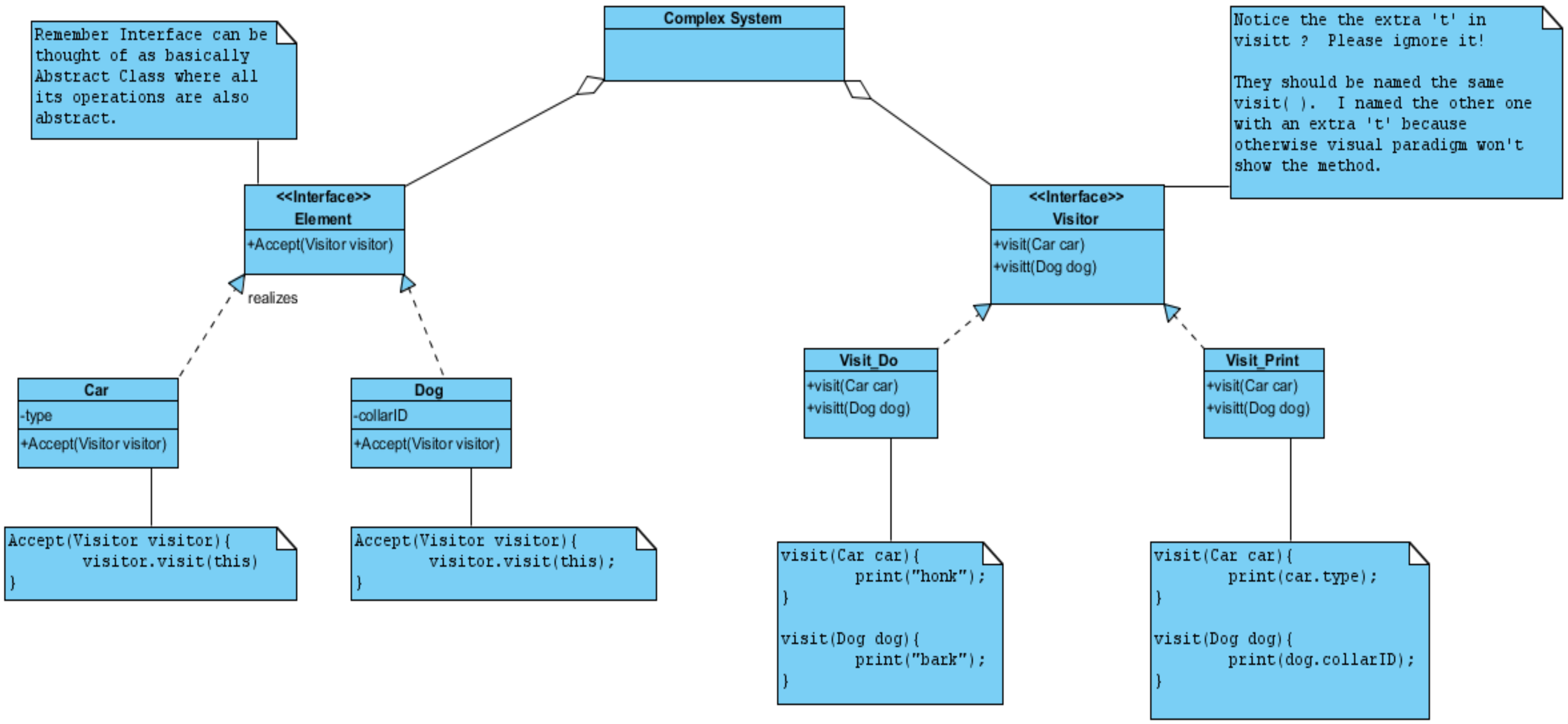
Yes! Split The Structures Into 2 Parts

*We pull out the operations and put it into its own structure, we called this structure the Visitor Class.

*We could then reorganize the elements into a more manageable structure. This way the relations between elements can be expressed more clearly.

Here, what I meant with elements, would be something like car, houses, numbers, cats, dogs, superman, kryptonite.

Operations, would be methods the elements have such as the bark( ) method in a dog, or transform( ) method in a car. Again, we pull out these methods and put it in a new structure we called the visitor.



Remember Interface can be thought of as basically Abstract Class where all its operations are also abstract.

Notice the the extra 't' in visitt ? Please ignore it!

They should be named the same visit( ). I named the other one with an extra 't' because otherwise visual paradigm won't show the method.

```
Accept(Visitor visitor){
        visitor.visit(this)
}
```

```
Accept(Visitor visitor){
        visitor.visit(this);
}
```

```
visit(Car car){
        print("honk");
}

visit(Dog dog){
        print("bark");
}
```

```
visit(Car car){
        print(car.type);
}

visit(Dog dog){
        print(dog.collarID);
}
```

The Elements The Visitors

Now in the main part of the system

Instead of doing the traditional:

```
Dog dog = new Dog();
dog.bark();
dog.getcollarID();
```

You'd be doing:

```
Dog dog = new Dog();
dog.accept(new visit_do());
dog.accept(new visit_print());
```

Instead of calling operation by accessing with '.' symbol. You instantiate an object of the operation and you passed it in to the accept method of the element.

the accept method can be passed object of visit_do or visit_print because these 2 objects are child of the abstract Visitor class

Java Example

```
interface Visitor{
      void visit(Car car);
      void visit(Dog dog);
}

public class Visit_Do implements Visitor{
   public void visit(Car car){
      System.out.println("Do Honk()");
   }

   public void visit(Dog dog){
      System.out.println("Do Bark()");
   }
}

public class Visit_Print implements Visitor{
    public void visit(Car car){
      System.out.println("car type");
    }

    public void visit(Dog dog){
      system.out.println("dog collarID");
    }
}
```

```
interface Elements{
      void accept(Visitor visitor);
}
public class Dog implements Elements{
   public void accept(Visitor visitor){
      visitor.visit(this);
   }
}
public class Car implements Elements{
   public void accept(Visitor visitor){
      visitor.visit(this);
   }
}
-------------------------------------------------------------
public class Example_MAIN{
   public static void main(String[] args){
      Car car = new Car();
      Dog dog = new Dog();
      car.accept(new Visit_Print());
      car.accept(new Visit_Do());
   }
}
```

Advantages of this design

*Clarity

By pulling the classes operations away. The classes elements can be expressed more simply & clearly.

*Efficiency

What if you have tons of operations but only used 1 of them at a time? declaring the operations and the elements in 1 class means wasted space when instantiating object of that class.

*The ability to add operations, even on runtime

wait... if we add new concrete classes to the visitor structure don't we need to recompile the whole thing? Nope, you put the visitor classes in a separate file and it can be compiled by itself. Linked to the main part of your program (e.g. via Class path -cp in java).

Disadvantages of this design

*You may need to break Encapsulation
What if your concrete visitor classes need to access an attributes of your elements? you would be forced to declare the attribute visibility scope into public!

*Difficult to implement on an already running system
You need to do significant restructuring of your classes