Software Design and Architecture

Builder and Visitor Patterns

Design principles

High-level principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution Principle
- **I**nterface Segregation
- **D**ependency Inversion

Low-level principles

- Encapsulate what varies
- Program to interfaces, not implementations
- Favor composition over inheritance
- Strive for loose coupling

Builder

Behavioral Patterns
- observer
- strategy
- command
- Template
- null object
- state
- iterator

Creational Patterns
- factory method
- abstract factory
- singleton
- **builder**

Structural Patterns
- decorator
- adapter
- façade
- composite
- bridge

Problem

We want to isolate implementation of a complex object, so that its construction process is the same as of another complex object.

Example

```
/** "Product" */
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";
  public void setDough(String dough) {
    this.dough = dough; }
  public void setSauce(String sauce) {
    this.sauce = sauce; }
  public void setTopping(String topping) {
    this.topping = topping; }
}
```

```
Create HawaiianPizza
     Pizza pizza;
     pizza.setDough("cross");
     pizza.setSauce("mild");
     pizza.setTopping("ham+pineapple");
```

Example

```
/** "Product" */
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";
  public void setDough(String dough) {
    this.dough = dough; }
  public void setSauce(String sauce) {
    this.sauce = sauce; }
  public void setTopping(String topping) {
    this.topping = topping; }
}
```

```
Create SpicyPizza
     Pizza pizza;
     pizza.setDough("pan baked");
     pizza.setSauce("hot");
     pizza.setTopping("pepperoni+salami");
```

Example

```
Create HawaiianPizza
      Pizza pizza;
      pizza.setDough("cross");
      pizza.setSauce("mild");
      pizza.setTopping("ham+pineapple");
```

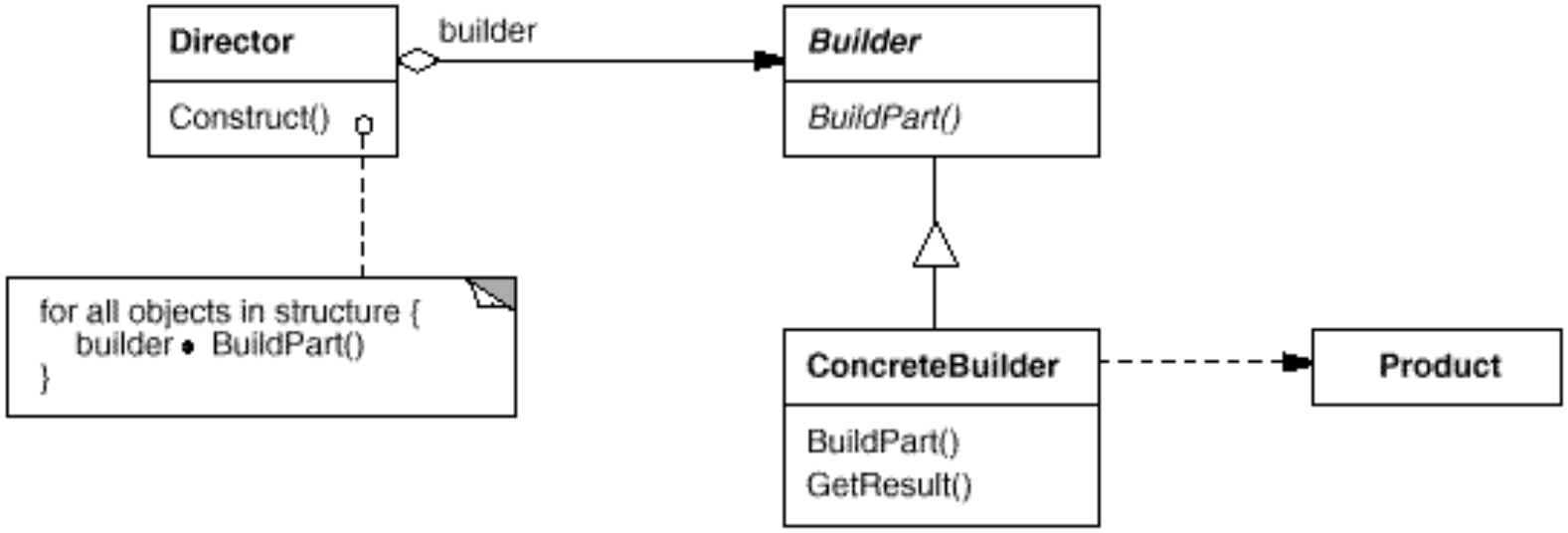
```
Create SpicyPizza
      Pizza pizza;
      pizza.setDough("pan baked");
      pizza.setSauce("hot");
      pizza.setTopping("pepperoni+salami");
```

We want to separate HawaiianPizza and SpicyPizza from their implementations. So that the same construction process can create different pizzas (representations).

Builder Pattern

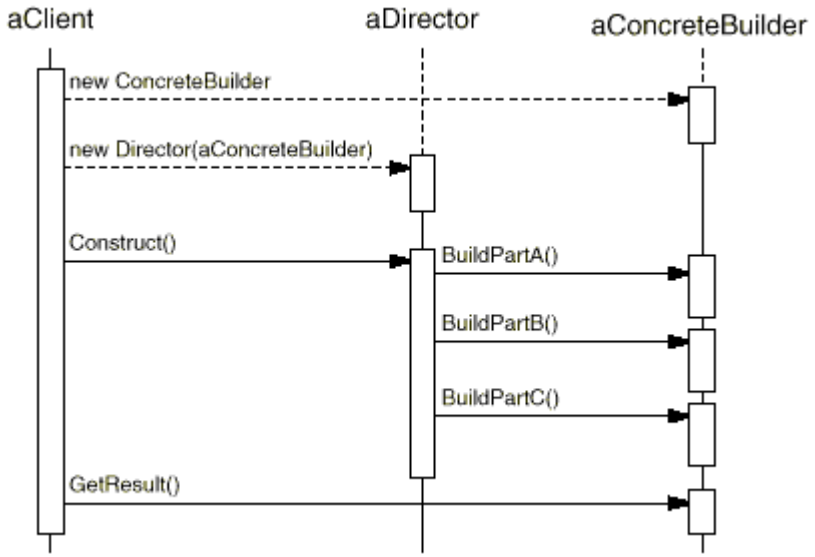
Separates the construction of a complex object from its representation so that the same construction process can create different representations.

Class Diagram



- **Builder** - specifies an abstract interface for creating parts of a **Product** object.
- **ConcreteBuilder -** constructs and assembles parts of the product by implementing the Builder interface. Also, it defines and keeps track of the representation it creates and provides an interface for retrieving the product .
- **Director** - constructs an object using the Builder interface.
- **Product** - represents the complex object under construction.

Sequence Diagram



- The client creates the *Director* object and configures it with the desired *Builder* object.
- *Director* notifies the builder whenever a part of the product should be built.
- *Builder* handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Pizza Builder Solution

```
/** "Product" */
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";
  public void setDough(String dough) {
    this.dough = dough; }
  public void setSauce(String sauce) {
    this.sauce = sauce; }
  public void setTopping(String topping) {
    this.topping = topping; }
}
```

Pizza Builder Solution

```
/** "Abstract Builder" */
 abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
      return pizza; }
    public void createNewPizzaProduct() {
      pizza = new Pizza(); }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
 }
```

Pizza Builder Solution

```
/** "ConcreteBuilder" */
 class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough()   {
      pizza.setDough("cross"); }
    public void buildSauce()   {
      pizza.setSauce("mild"); }
    public void buildTopping() {
      pizza.setTopping("ham+pineapple"); }
 }
 /** "ConcreteBuilder" */
 class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough()   {
      pizza.setDough("pan baked"); }
    public void buildSauce()   {
      pizza.setSauce("hot"); }
    public void buildTopping() {
      pizza.setTopping("pepperoni+salami"); }
 }
```

Pizza Builder Solution

```
/** "Director" */
class Waiter {
   private PizzaBuilder pizzaBuilder;
   public void setPizzaBuilder(PizzaBuilder pb) {
     pizzaBuilder = pb; }
   public Pizza getPizza() {
     return pizzaBuilder.getPizza(); }
   public void constructPizza() {
      pizzaBuilder.createNewPizzaProduct();
      pizzaBuilder.buildDough();
      pizzaBuilder.buildSauce();
      pizzaBuilder.buildTopping();
   }
}
```

Pizza Builder Solution

```
 /** A customer ordering a pizza. */
class BuilderExample {
   public static void main(String[] args) {
       Waiter waiter = new Waiter();
       PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
       PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

       waiter.setPizzaBuilder( hawaiianPizzaBuilder );
       waiter.constructPizza();

       Pizza pizza = waiter.getPizza();
   }
}
```

Visitor

Behavioral Patterns
- observer
- strategy
- command
- template
- state
- iterator
- **visitor**

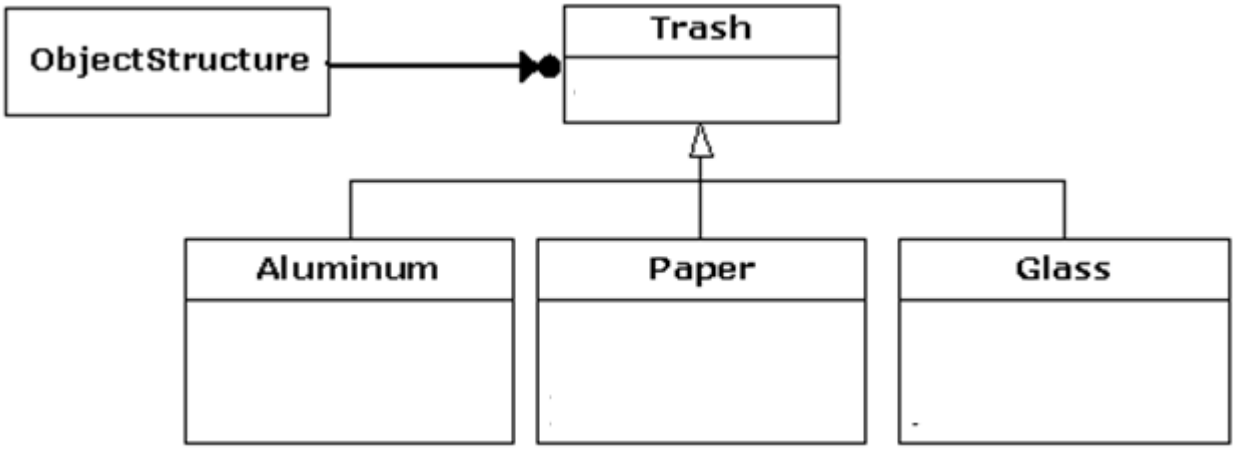
Creational Patterns
- factory method
- abstract factory
- singleton
- builder

Structural Patterns
- decorator
- adapter
- façade
- composite
- bridge

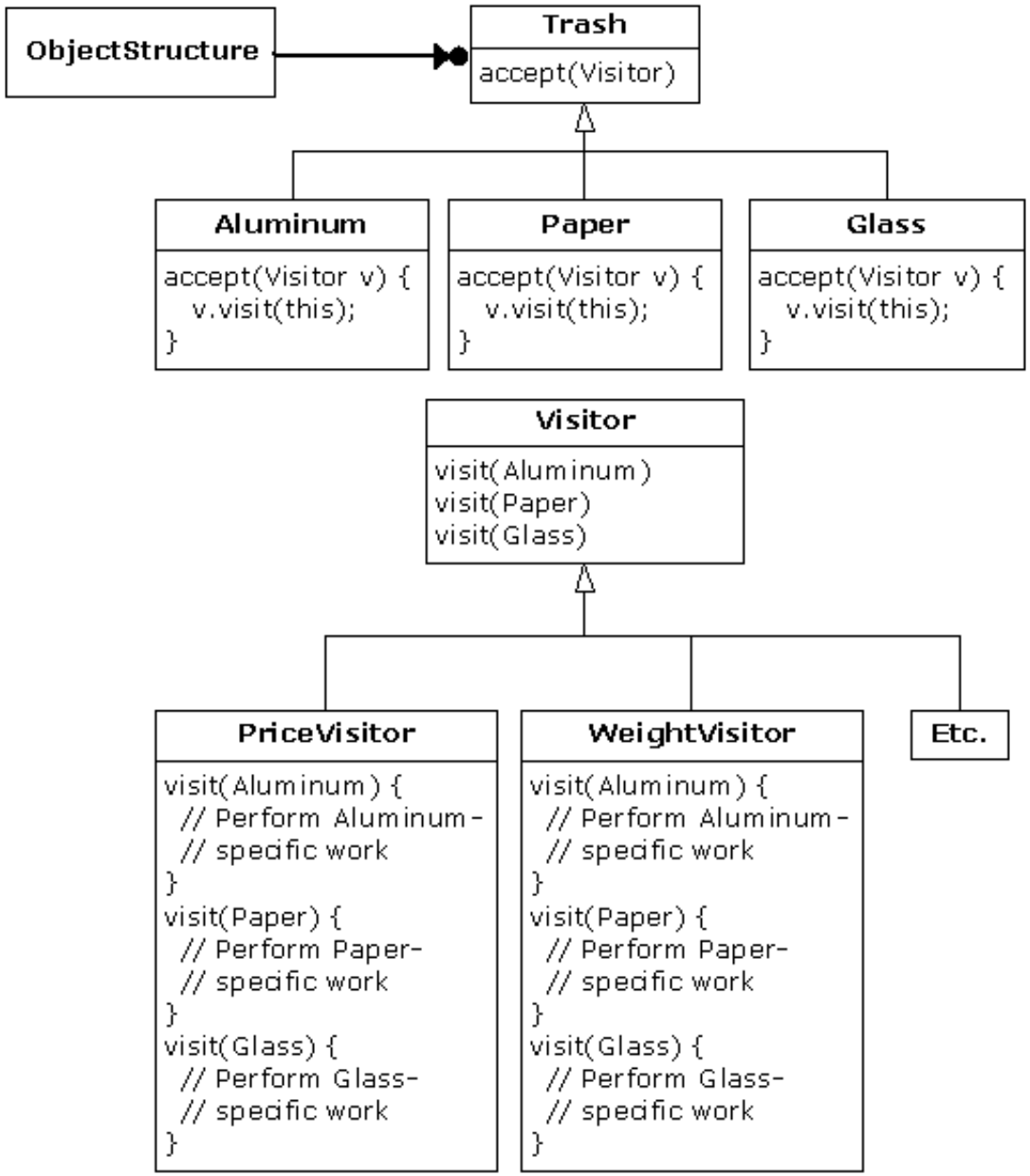
Problem

We want to perform new operations on concrete classes and we don’t want to “pollute” the classes with these operations.

Example



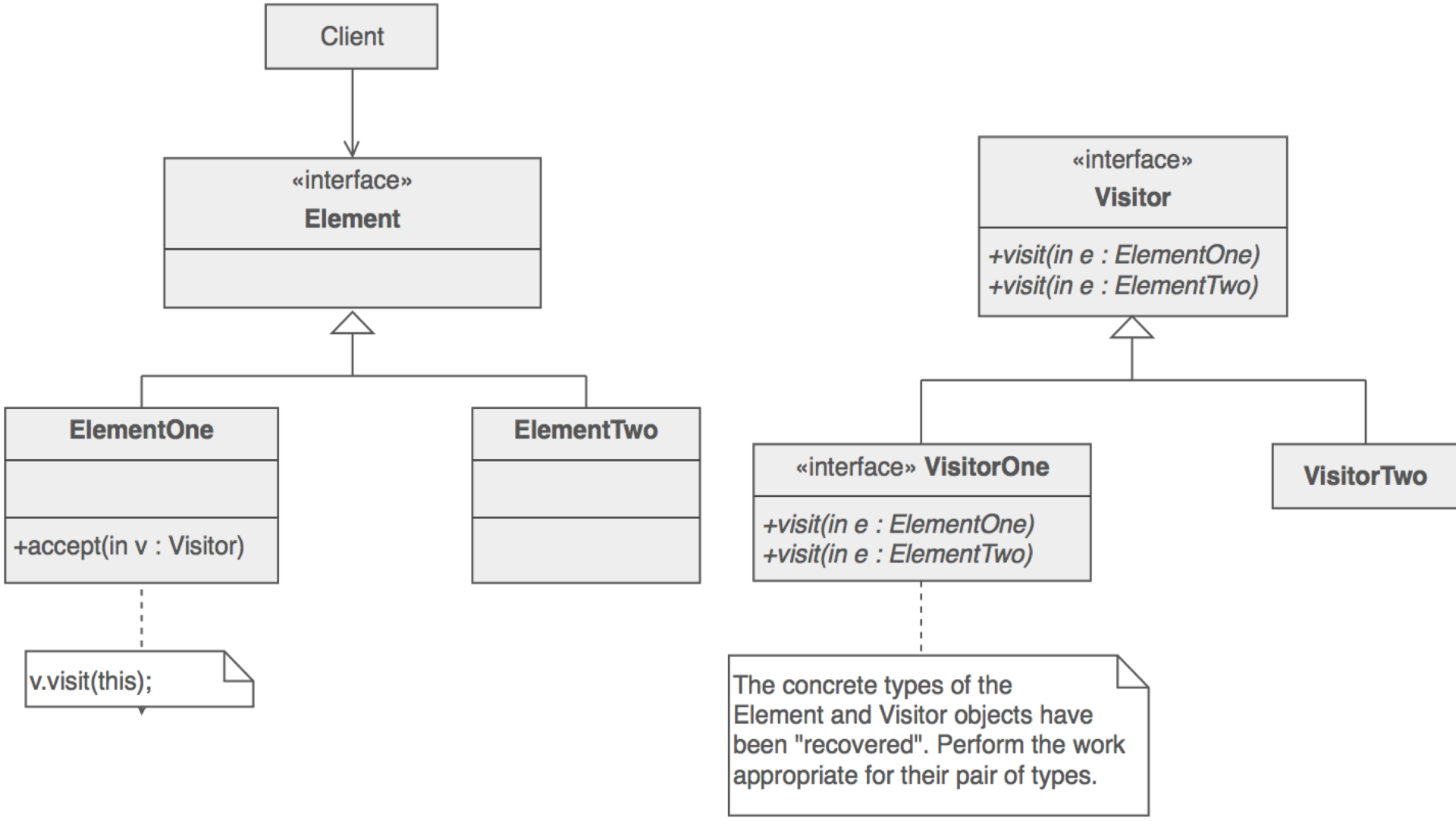
Example


ObjectStructure
Trash
accept(Visitor)
Aluminum
accept(Visitor v) {
v.visit(this);
}
Paper
accept(Visitor v) {
v.visit(this);
}
Glass
accept(Visitor v) {
v.visit(this);
}
Visitor
visit(Aluminum)
visit(Paper)
visit(Glass)
PriceVisitor
visit(Aluminum) {
// Perform Aluminum-
// specific work
}
visit(Paper) {
// Perform Paper-
// specific work
}
visit(Glass) {
// Perform Glass-
// specific work
}
WeightVisitor
visit(Aluminum) {
// Perform Aluminum-
// specific work
}
visit(Paper) {
// Perform Paper-
// specific work
}
visit(Glass) {
// Perform Glass-
// specific work
}
Etc.

Visitor Pattern

Allows for one or more operation to be applied to a set of objects at runtime, decoupling the operations from the object structure.

Class Diagram



Participants

Visitor – declares a visit operation for each class of ConcreteElement in the object structure

ConcreteVisitor – implements each operation declared by Visitor

Element – defines an Accept operation that takes a visitor as an argument

ConcreteElement – implements an Accept operation that takes a visitor as an argument

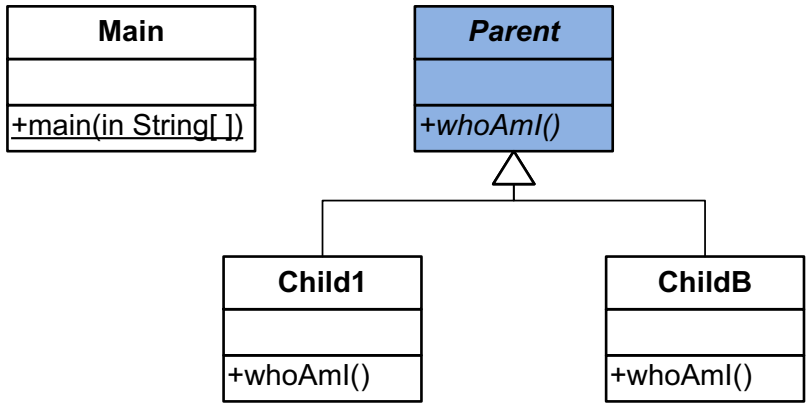
An Example

```
abstract class Parent {
   abstract void whoAmI();
}
```

```
class Child1 extends Parent {
   void whoAmI() {
      print("Child1");
   }
}
```

```
class Child2 extends Parent {
   void whoAmI() {
      print("Child2");
   }
}
```

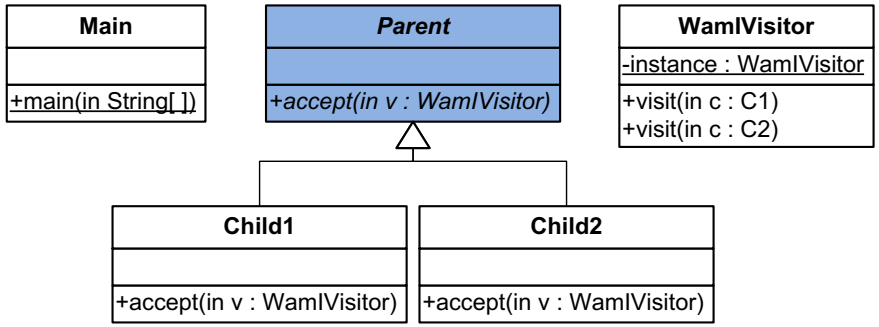
```
public static void main(String[] args) {
   Child1 c1 = new Child1();
   c1.whoAmI();
   Child2 c2 = new Child2();
   c2.whoAmI();
}
```



create a visitor for method whoAmI()

A Solution

```
abstract class Parent {
   abstract void accept(WamIVisitor v);
}
```




```
class Child1 extends Parent {
   void accept(WamIVisitor v) {
      v.visit(this);
   }
}
```

```
class Child2 extends Parent {
   void accept(WamIVisitor v) {
      v.visit(this);
   }
}
```

```
public static void main(String[] args) {
   Child1 c1 = new Child1();
   c1.accept(WamIVisitor.instance);
   Child2 c2 = new Child2();
   c2.accept(WamIVisitor.instance);
}
```

```
class WamIVisitor
   static WamIVisitor instance =
      new WamIVisitor();

   void visit(Child1 c) {
     print("Child1");
   }
   void visit(Child2 c) {
     print("Child2");
   }
}
```