**using** System;

**using** System.Collections.Generic;

**namespace** **RefactoringGuru**.DesignPatterns.Visitor.Conceptual

{

// The Component interface declares an `accept` method that should take the

// base visitor interface as an argument.

**public** **interface** **IComponent**

{

**void** Accept(IVisitor visitor);

}

// 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.

**public** **class** **ConcreteComponentA** : IComponent

{

// 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(IVisitor visitor)

{

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.

**public** **string** ExclusiveMethodOfConcreteComponentA()

{

**return** "A";

}

}

**public** **class** **ConcreteComponentB** : IComponent

{

// Same here: VisitConcreteComponentB => ConcreteComponentB

**public** **void** Accept(IVisitor visitor)

{

visitor.VisitConcreteComponentB(**this**);

}

**public** **string** SpecialMethodOfConcreteComponentB()

{

**return** "B";

}

}

// 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.

**public** **interface** **IVisitor**

{

**void** VisitConcreteComponentA(ConcreteComponentA element);

**void** VisitConcreteComponentB(ConcreteComponentB element);

}

// 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** : IVisitor

{

**public** **void** VisitConcreteComponentA(ConcreteComponentA element)

{

Console.WriteLine(element.ExclusiveMethodOfConcreteComponentA() + " + ConcreteVisitor1");

}

**public** **void** VisitConcreteComponentB(ConcreteComponentB element)

{

Console.WriteLine(element.SpecialMethodOfConcreteComponentB() + " + ConcreteVisitor1");

}

}

**class** **ConcreteVisitor2** : IVisitor

{

**public** **void** VisitConcreteComponentA(ConcreteComponentA element)

{

Console.WriteLine(element.ExclusiveMethodOfConcreteComponentA() + " + ConcreteVisitor2");

}

**public** **void** VisitConcreteComponentB(ConcreteComponentB element)

{

Console.WriteLine(element.SpecialMethodOfConcreteComponentB() + " + ConcreteVisitor2");

}

}

**public** **class** **Client**

{

// 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.

**public** **static** **void** ClientCode(List<IComponent> components, IVisitor visitor)

{

**foreach** (**var** **component** **in** components)

{

component.Accept(visitor);

}

}

}

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

List<IComponent> components = **new** List<IComponent>

{

**new** ConcreteComponentA(),

**new** ConcreteComponentB()

};

Console.WriteLine("The client code works with all visitors via the base Visitor interface:");

**var** **visitor1** = **new** ConcreteVisitor1();

Client.ClientCode(components,visitor1);

Console.WriteLine();

Console.WriteLine("It allows the same client code to work with different types of visitors:");

**var** **visitor2** = **new** ConcreteVisitor2();

Client.ClientCode(components, visitor2);

}

}

}