Tecniche di Progettazione: Design Patterns

GoF: Visitor

Visitor Pattern

- Intent
  - Lets you define a new operation without changing the classes on which they operate.
- Motivation
  - Allows for increased functionality of a class(es) while streamlining base classes.
  - A primary goal of designs should be to ensure that base classes maintain a minimal set of operations.
  - Encapsulates common functionality in a class framework.

Visitor Pattern

- Motivation (cont)
  - Visitors avoid type casting that is required by methods that pass base class pointers as arguments. The following code describes how a typical class could expand the functionality of an existing composite.

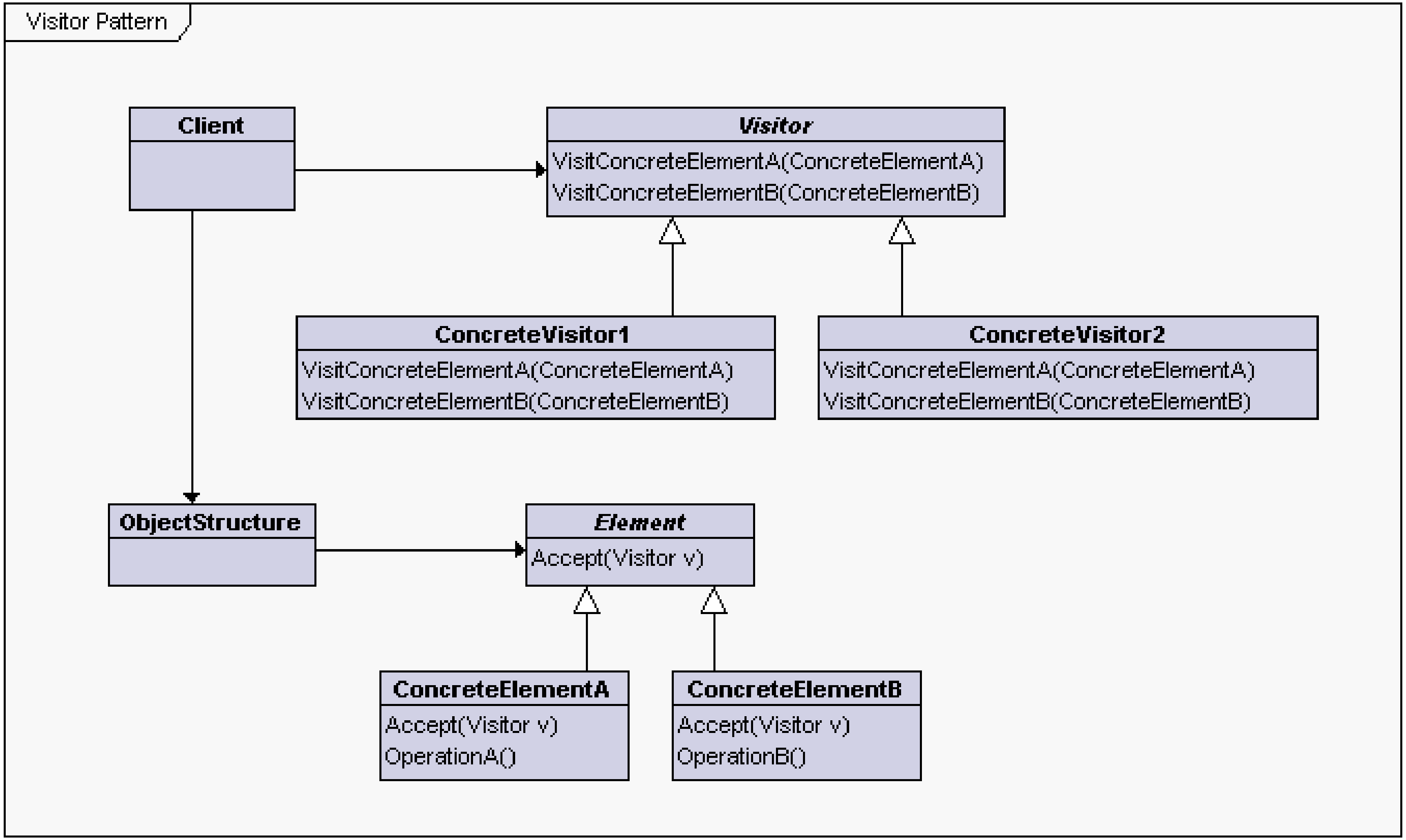
```
myOperation(Base b) {
        if (b instanceof ChildA){
                        // Perform task for child type A.
        } else if (b instanceof ChildB){
                        // Perform task for child type B.
        } else if (b instanceof ChildC){
                        // Perform task for child type C.
        }
}
```

Single vs double dispatch

- **double dispatch** is a mechanism that dispatches a function call to different concrete functions depending on:
  - the runtime types of two objects involved in the call.

- With **single dispatch** the operation that is executed depends on: the name of the request, and the type of the receiver.

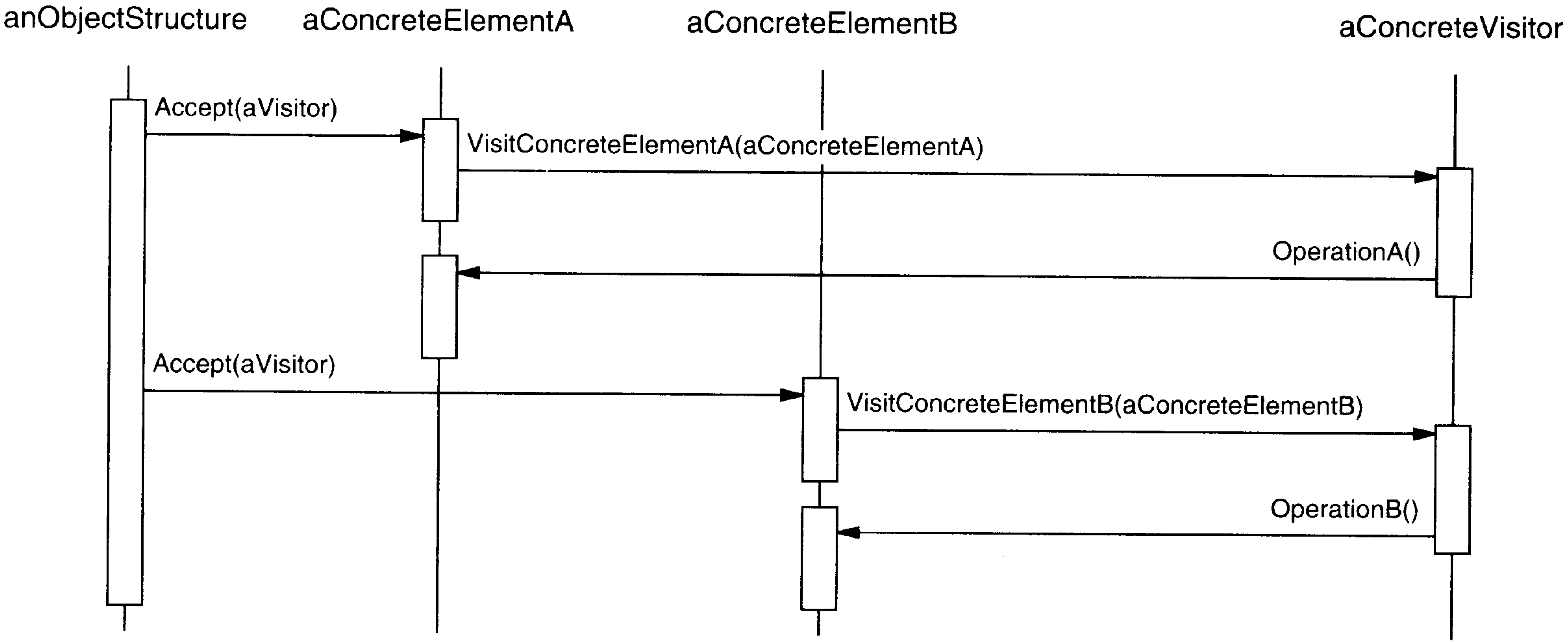
Structure


Visitor Pattern
Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor v)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()

Visitor Pattern: Participants

- Visitor
  - Declares a Visit Operation for each class of Concrete Elements in the object structure.
- Concrete Visitor
  - Implements each operation declared by Visitor.
- Element
  - Defines an Accept operation that takes the visitor as an argument.
- Concrete Element
  - Implements an accept operation that takes the visitor as an argument.
- Object Structure
  - Can enumerate its elements.
  - May provide a high level interface to all the visitor to visit its elements.
  - May either be a composite or a collection.

Visitor Pattern: Collaborations



Visitor Pattern: Applicability

- When an object structure contains many classes of objects with different interfaces and you want to perform functions on these objects that depend on their concrete classes.
- When you want to keep related operations together by defining them in one class.
- When the class structure rarely change but you need to define new operations on the structure.
- When many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations

Visitor Pattern: Consequences

- Makes adding new operations easier.
- Collects related functionality.
- Adding new Concrete Element classes is difficult.
- Can "visit" across class types, unlike iterators.
- Accumulates states as they visit elements.
- May require breaking object encapsulation to support the implementation.

Static or Dynamic binding

```
public interface Visitor {
   public void visitX(X x);
   public void visitY(Y y);
}
public class ConcreteVisitor {
   public void visitX(X x) { … }
   public void visitY(Y y) { … }
}

public abstract class XY {
   public abstract void accept(Visitor v);
}
public class X extends XY {
   public void accept(Visitor v) { v.visitX(this);  }
}
public class Y extends XY {
   public void accept(Visitor v) { v.visitY(this);  }
}
```

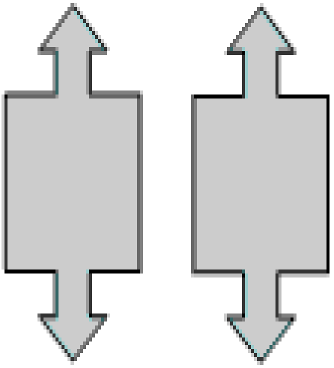
```
public interface Visitor {
   public void visit(X x);
   public void visit(Y y);
}
public class ConcreteVisitor {
   public void visit(X x) { … }
   public void visit(Y y) { … }
}

public abstract class XY {
   public abstract void accept(Visitor v);
}
public class X extends XY {
   public void accept(Visitor v) { v.visit(this);}
}
public class Y extends XY {
  public void accept(Visitor v) { v.visit(this);}
}
```

Visitor

1) Start with an inheritance hierarchy to which you would like to add new operations without the need to modify existing code.

2) Add an "accept(Visitor)" method to this existing hierarchy.



double polymorphism

3) Define a new second hierarchy called "Visitor" that has as many "visit" methods as the first hierarchy has derived classes.

4) The client calls accept() on an instance of the first hierarchy, and passes an instance of the second hierarchy.

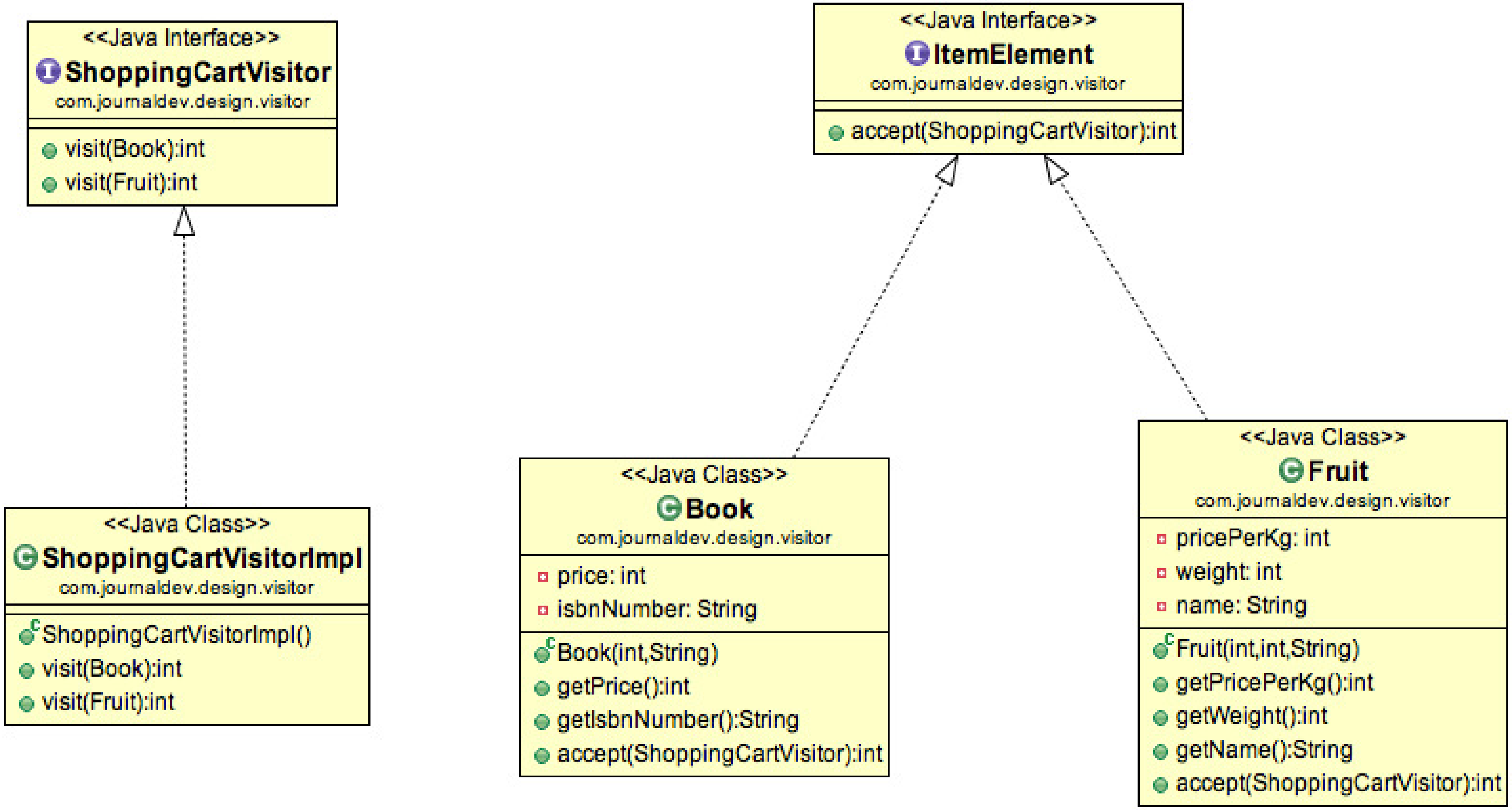
5) The accept() method calls visit() on the object it was passed.

6) The magic of dynamic binding (applied twice) vectors flow of control to the right piece of code based on the type of **two** objects.

Visitor: Related Patterns

- Composites
  - Visitors can be used to apply an operation over an object structure defined by the composite pattern.
- Interpreter
  - Visitors may be applied to do the interpretation.

Example



<<Java Interface>>
ShoppingCartVisitor
com.journaldev.design.visitor
visit(Book):int
visit(Fruit):int
<<Java Interface>>
ItemElement
com.journaldev.design.visitor
accept(ShoppingCartVisitor):int
<<Java Class>>
ShoppingCartVisitorImpl
com.journaldev.design.visitor
ShoppingCartVisitorImpl()
visit(Book):int
visit(Fruit):int
<<Java Class>>
Book
com.journaldev.design.visitor
price: int
isbnNumber: String
Book(int,String)
getPrice():int
getIsbnNumber():String
accept(ShoppingCartVisitor):int
<<Java Class>>
Fruit
com.journaldev.design.visitor
pricePerKg: int
weight: int
name: String
Fruit(int,int,String)
getPricePerKg():int
getWeight():int
getName():String
accept(ShoppingCartVisitor):int