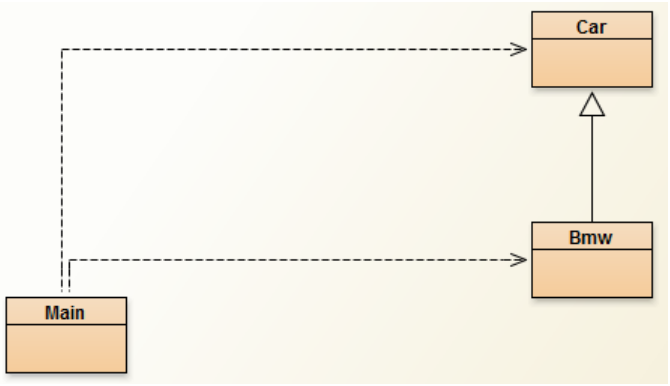
Tecniche di Progettazione: Design Patterns

GoF: Visitor

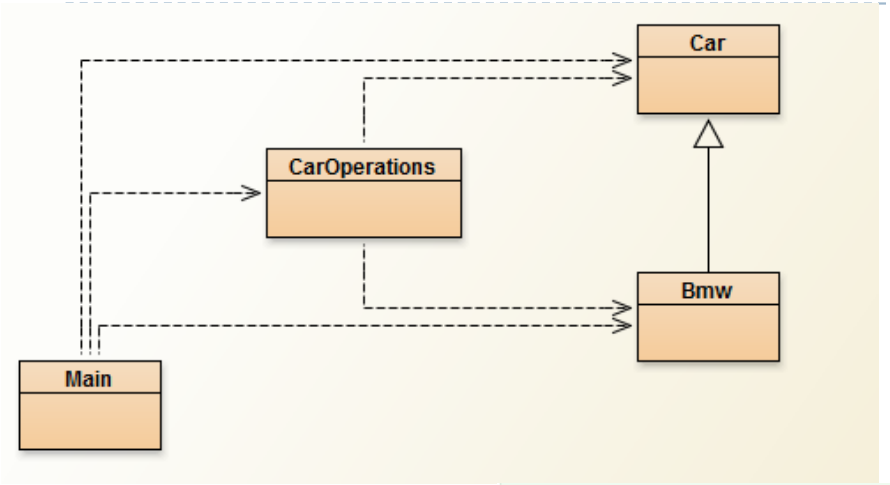
Polymorphism reminded: overriding and dynamic binding




```
public class Car {
   public void doVroom(){System.out.println("vroom");}
}
public class Bmw extends Car {
   public void doVroom(){System.out.println("VROOM");}
}
```

```
public class Main{
    public static void main(String[] args) {
        Car bmw = new Bmw();
        bmw.doVroom(); //calls Bmw.doVroom()
    }
}
```

Polymorphism reminded: overloading and static dispatch



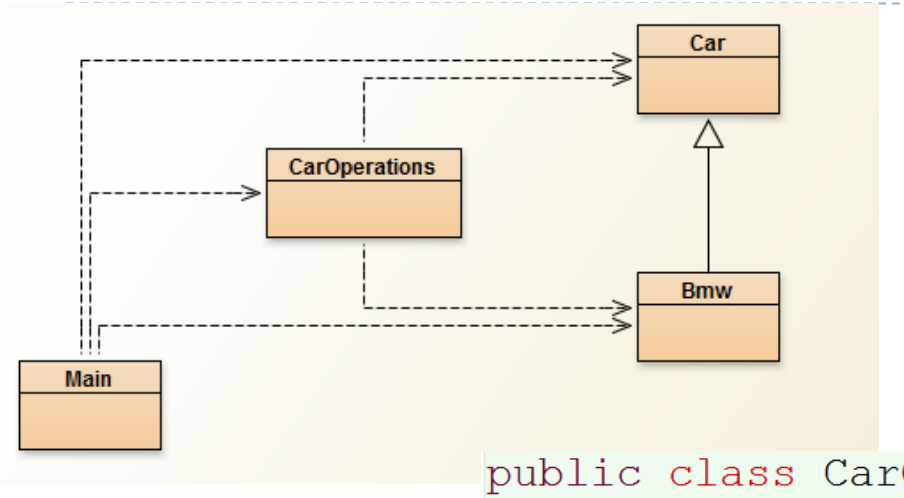

```
public class CarOperations {

   void doWroom(Car car){System.out.println("wroom");}
   void doWroom(Bmw car){System.out.println("WROOM");}

}
```

```
public class Main{
    public static void main(String[] args) {
        Car bmw = new Bmw();
        CarOperations carops = new CarOperations();
        carops.doWroom(bmw); //calls CarOperations.doWroom(Car car)
```

Polymorphism reminded: overloading may need downcasting

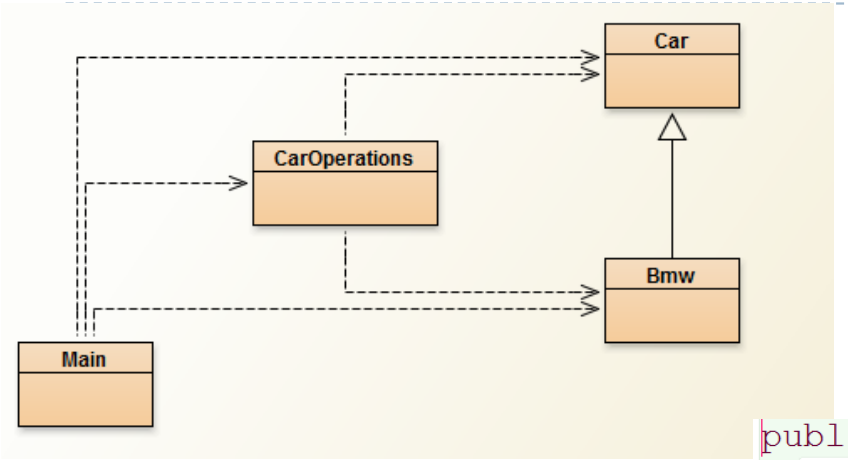



```
public class CarOperations {

    void doWroom(Car car){System.out.println("wroom");}
    void doWroom(Bmw car){System.out.println("WROOM");}

}
public class Main{
    public static void main(String[] args) {
        Car bmw = new Bmw();
        CarOperations carops = new CarOperations();
        carops.doWroom((Bmw)bmw); //calls CarOperations.doWroom(Bmw car)
        }
}
```

Polymorphism reminded:
overloading with instanceof and downcasting



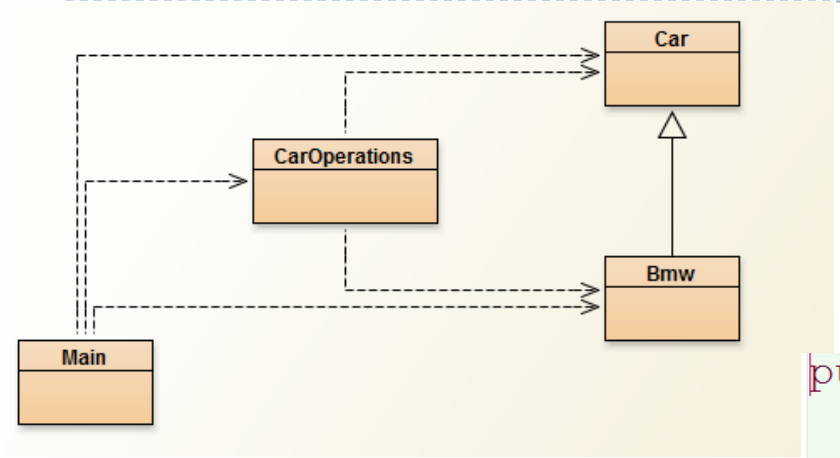

```
public class CarOperations {

    void doWroom(Car car){System.out.println("wroom");}
    void doWroom(Bmw car){System.out.println("WROOM");}

}
```

```
public class Main{
    public static void main(String[] args) {
        Car bmw = new Bmw();
        CarOperations carops = new CarOperations();
        if (bmw instanceof Bmw)
            carops.doWroom((Bmw)bmw);
        else if (bmw instanceof Car)
            carops.doWroom(bmw);
        }
}
```

Polymorphism reminded:
with instanceof and no overloading




```
public class CarOperations {
    void doWroom(Car car){
        if (car instanceof Bmw )
            System.out.println("WROOM");
        else if (car instanceof Car )
            System.out.println("wroom");
```

```
public class Main{
    public static void main(String[] args) {
        Car bmw = new Bmw();
        CarOperations carops = new CarOperations();
        carops.doWroom(bmw);
        }
}
```

Visitor

- With the pattern Visitor we can avoid this:

```
public class CarOperations {
   void doWroom(Car car){
        if (car instanceof Bmw )
              System.out.println("WROOM");
        else if (car instanceof Car )
              System.out.println("wroom");
    }
   }
```

- And the pattern has also a more general intent

Visitor Pattern

- Intent
  - Lets you define new operations 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.

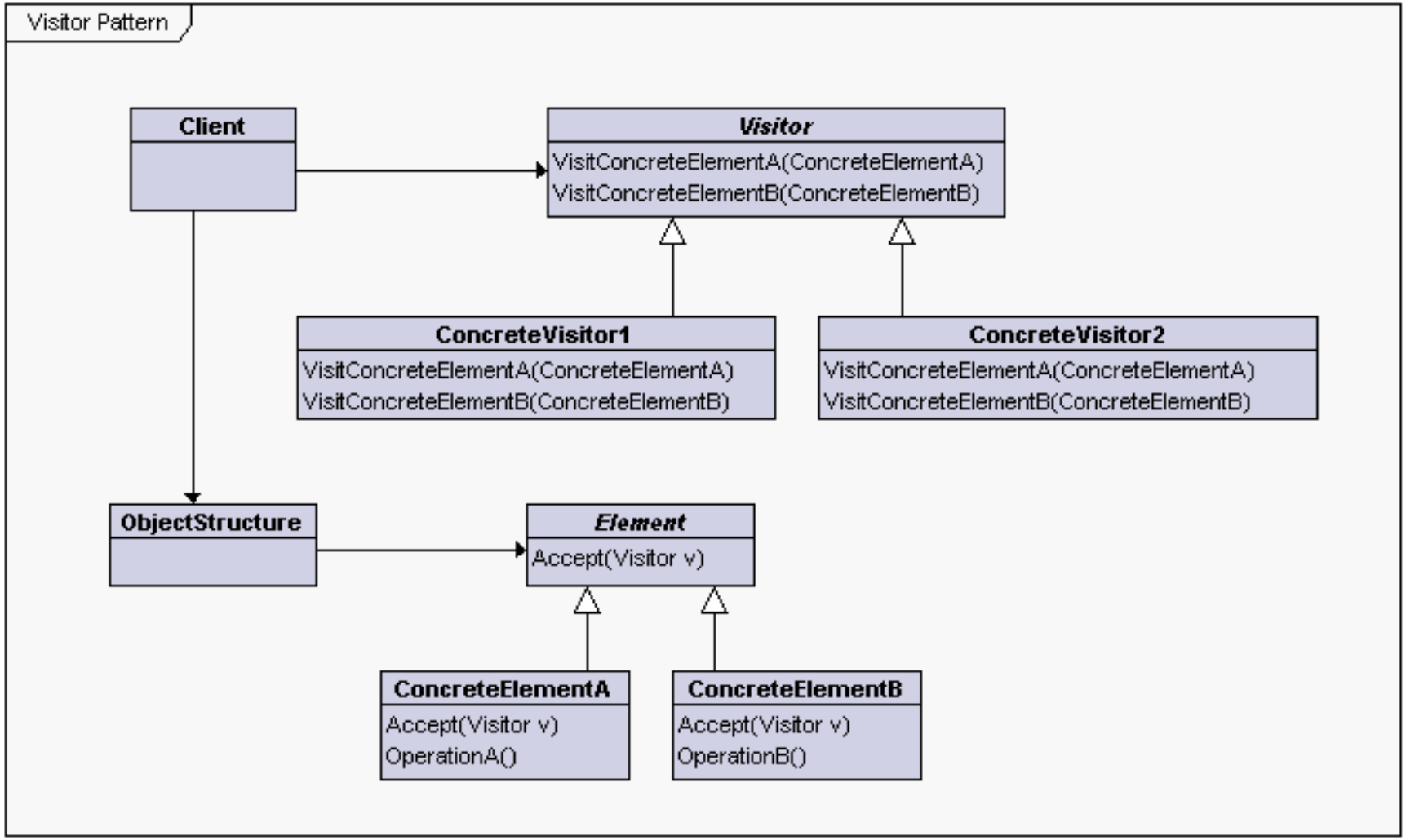
```
myOperation(Base b) {
        if (b instanceof ChildA){
                        // Perform task for child type A.
        } else if (b instanceof ChildB){
                        // Perform task for child type B.
        } else if (b instanceof ChildC){
                        // Perform task for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

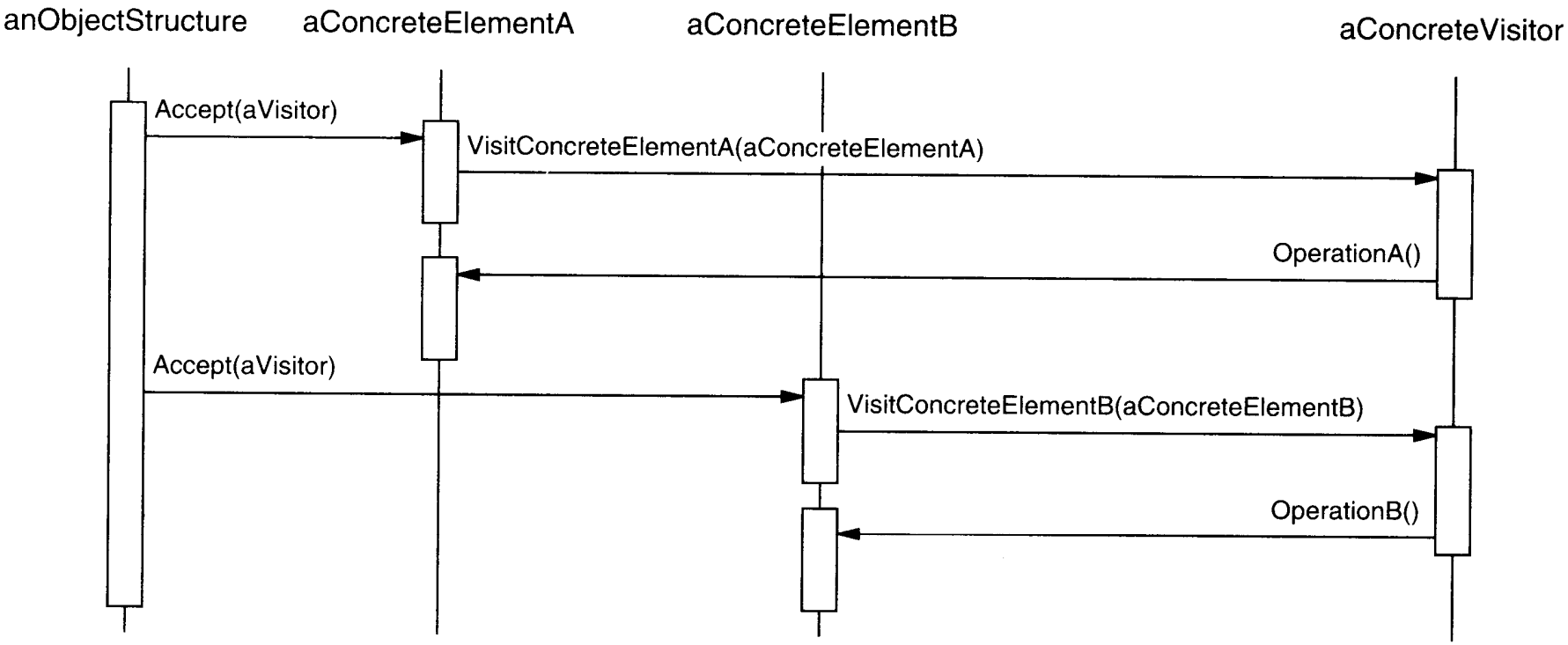
Structure


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

```
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); }
}
```

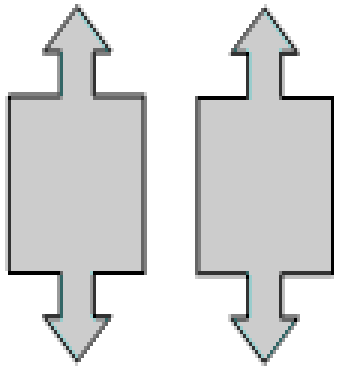
```
public interface Visitor {
   public void visit(X x);
   public void visit(Y y);
}
public class ConcreteVisitor {
   public void visit(X x) { ... }
   public void visit(Y y) { ... }
}

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.



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

double polymorphism

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

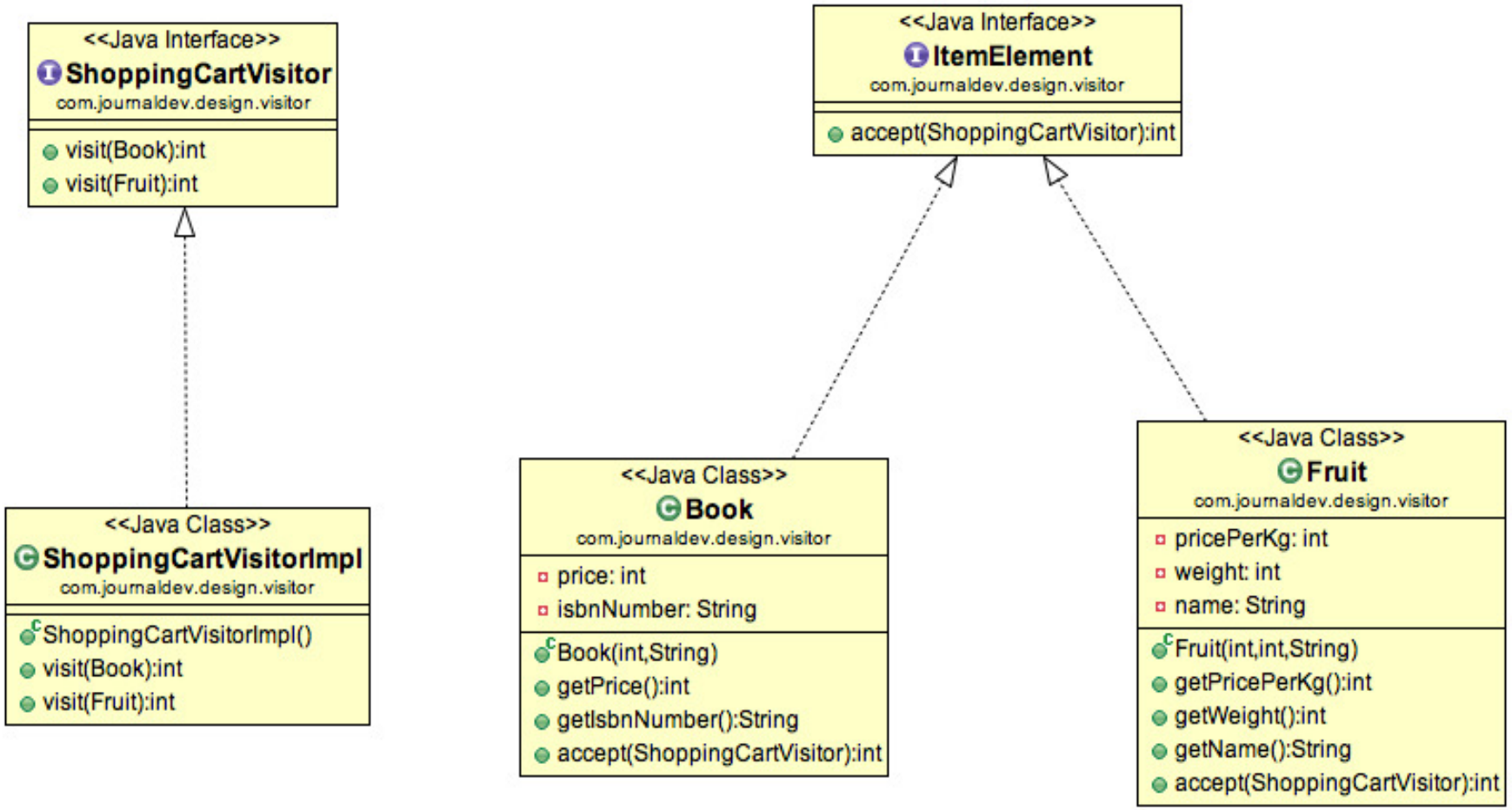
5) The accept() method calls visit() on the object it was passed.

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: write the code



Exercise (may be homework, unless you develop another example of application of visitor)

Given the following classes

class Red {…}

class Blue {…}

class Green {…}

Assume an array "colors" contains mixed objects from the above 3 classes. We want to the following:

1. To find number of Red, Blue, and Green objects, respectively, in the array.
2. To create another array "sortedColors" which contain the same objects as in "colors" but sorted according to their colors in red-blue-green order.

Using Visitor pattern to implement the above two operations. You need to write abstract and concrete Visitor classes, modify Red, Blue, Green classes, and write a main program to use a Visitor object to perform each of the operations.

Game of Life

- Group project due in some weeks

- The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent.

Game of Life

Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.

- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any dead cell with exactly three live neighbours becomes a live cell.

Game of Life

- The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed?births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the one before). The rules continue to be applied repeatedly to create further generations.