

# Prototype, Visitor

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 31

# Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

# Head First Design Patterns

- Chapter 13 – Leftover Patterns

- Bridge
- Builder
- Flyweight
- Interpreter
- Chain of Responsibility
- Mediator
- Memento
- Prototype
- Visitor

# Prototype

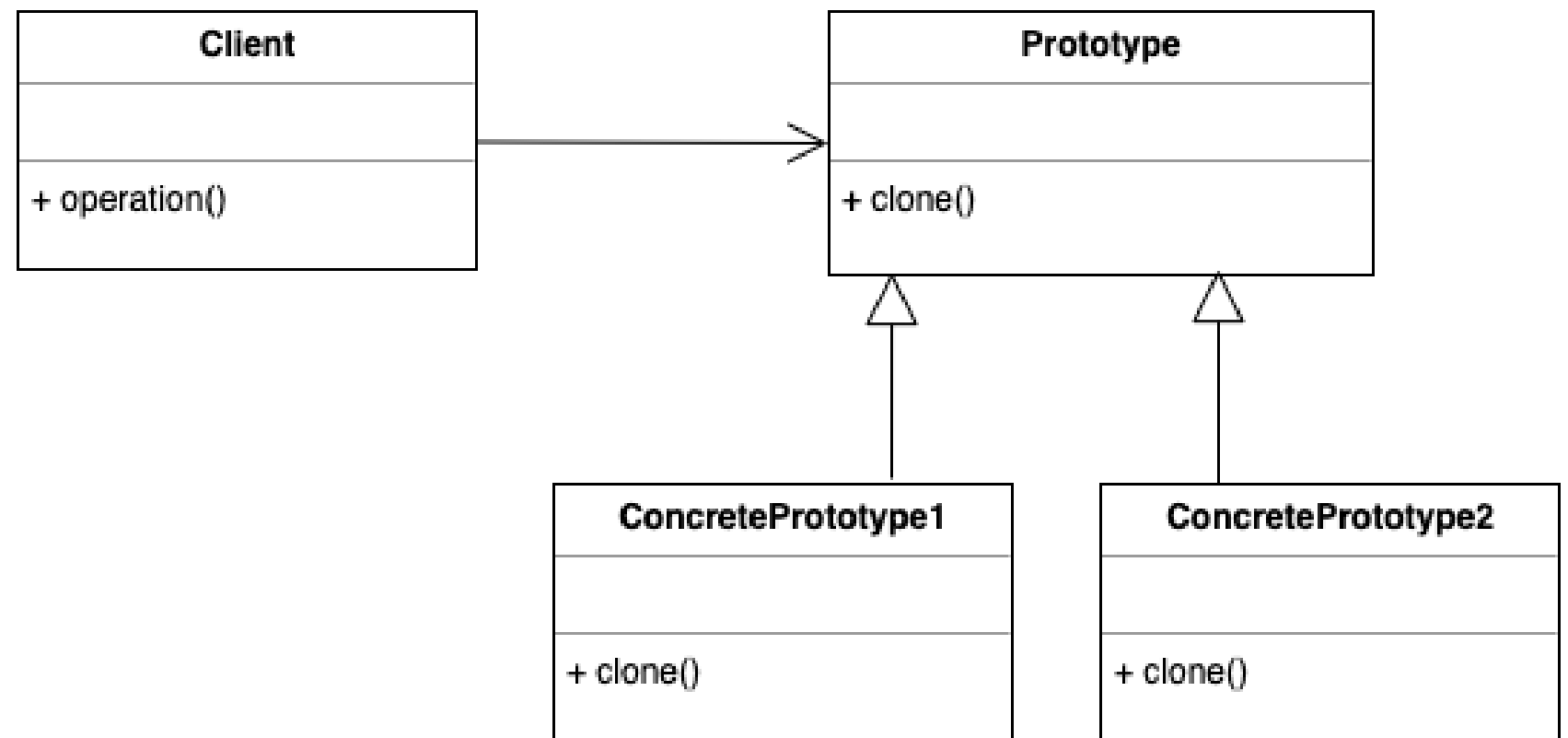
- The Prototype Pattern is used when creating an instance of a given class is either expensive or complicated
- Head First example:  
Back in the on-line RPG... It's about monsters.
  - Tons of different monsters
  - Monster characteristics that change based on landscape/setting
  - Advanced players can create their own monsters
  - ...

# Prototype Problem/Solution

- Problem: We want to decouple the code for the details of creating instances from code that needs to create instances on the fly
- Solution: Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes.
  - Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.
  - The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.
  - Used when we don't want to keep creating “new” instances of a class
- [https://sourcemaking.com/design\\_patterns/prototype](https://sourcemaking.com/design_patterns/prototype)

# Prototype Pattern: UML Class Diagram

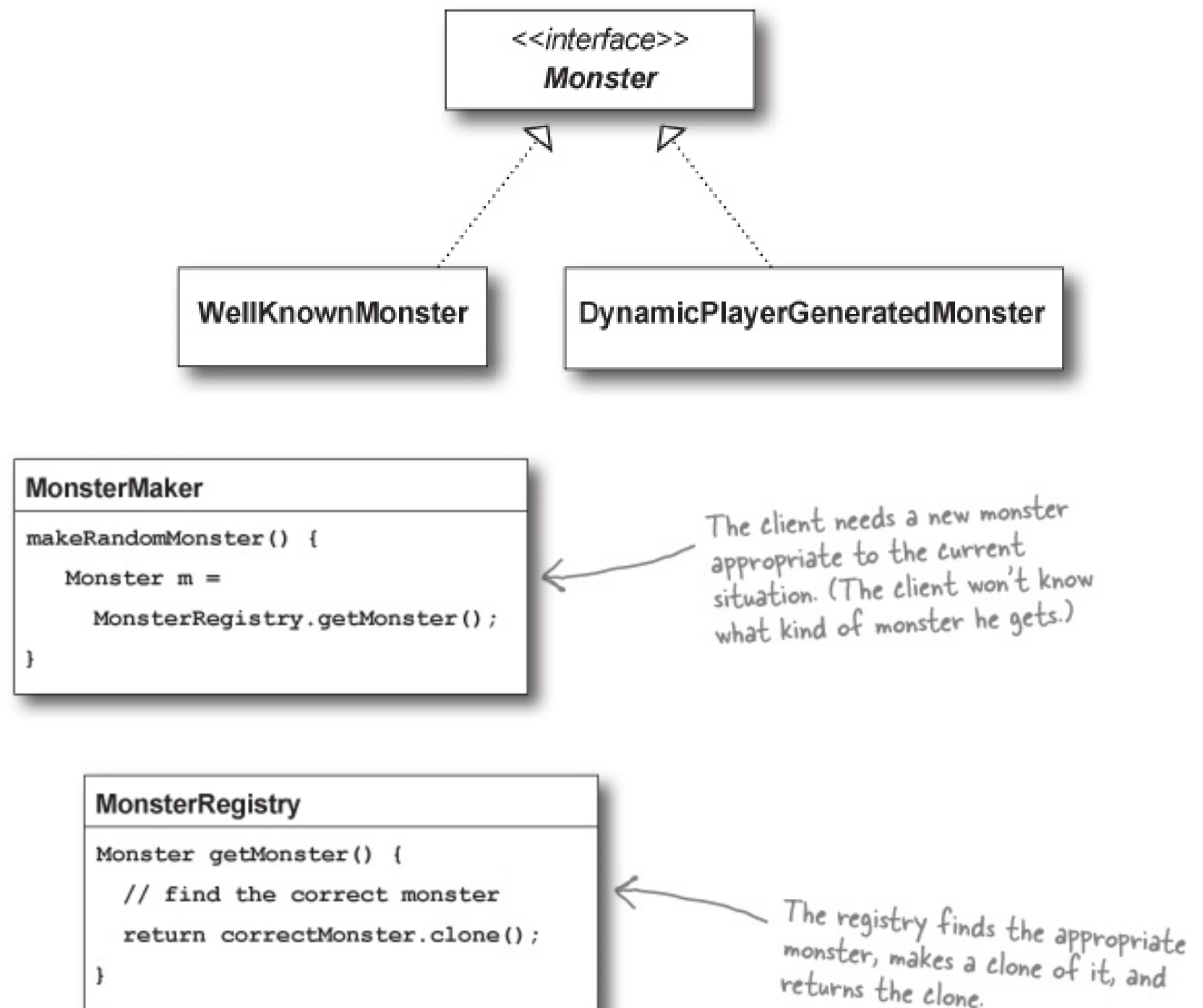
- The client is telling the prototype to clone itself and create an object
- Prototype is an interface and declares a method for cloning itself
- ConcretePrototype1 and ConcretePrototype2 implement the operation to clone themselves.



<https://www.baeldung.com/java-pattern-prototype>

# Prototype Applied

- Prototype Pattern allows you to make new instances by copying existing instances.
- In Java this typically means using the clone() method, or de-serialization when you need deep copies.
- A key aspect of this pattern is that the client code can make new instances without knowing which specific class is being instantiated



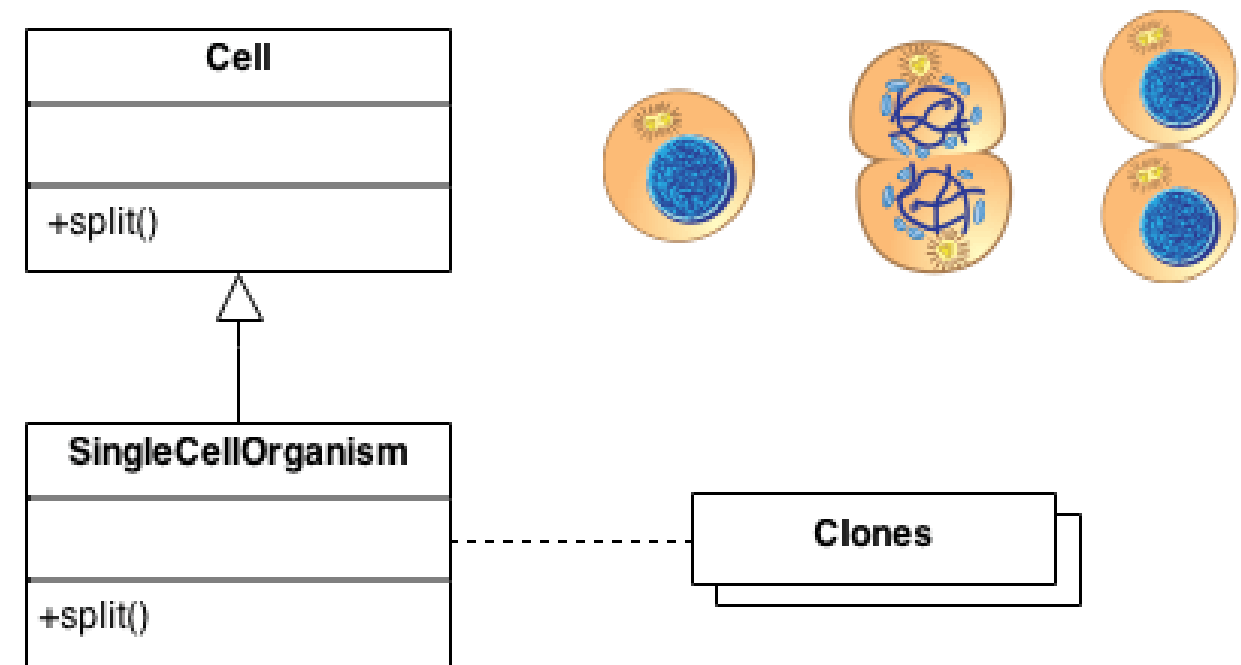
# Prototype Considered

- We can implement this pattern in Java is by using the clone() method; to do this, we'd implement the Cloneable interface
- When we're trying to clone, we should decide between making a shallow or a deep copy depending on requirements
- For example, if the class contains only primitive and immutable fields, we may use a shallow copy (not including referred objects)
- If it contains references to mutable fields, we should go for a deep copy (includes copies of objects found in original)
  - Could do that with copy constructors or serialization/deserialization.
- Python has copy and deepcopy functions for shallow/deep copies



# Another Prototype Example

- The Prototype pattern specifies the kind of objects to create using a prototypical instance
- Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself
- The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern
- When a cell splits, two cells of identical genotype result. In other words, the cell clones itself
- [https://sourcemaking.com/design\\_patterns/prototype](https://sourcemaking.com/design_patterns/prototype)



# Prototype in Java

```
public class Tree implements Cloneable {  
  
    // ...  
    @Override  
    public Tree clone() {  
        Tree tree = null;  
        try {  
            tree = (Tree) super.clone();  
        } catch (CloneNotSupportedException e) {  
            // ...  
        }  
        return tree;  
    }  
  
    // ...  
}
```

<https://www.baeldung.com/java-pattern-prototype>

```
public class TreePrototypesUnitTest {  
  
    @Test  
    public void TreeCloner() {  
        // ...  
  
        Tree tree = new Tree(mass, height);  
        tree.setPosition(position);  
        Tree anotherTree = tree.clone();  
        anotherTree.setPosition(otherPosition);  
  
        assertEquals(position, tree.getPosition());  
        assertEquals(otherPosition,  
            anotherTree.getPosition());  
    }  
}
```

# Prototype and other Patterns

- Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.
- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Factory Method: creation through inheritance. Prototype: creation through delegation.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- [https://sourcemaking.com/design\\_patterns/prototype](https://sourcemaking.com/design_patterns/prototype)

# Prototype and other Patterns

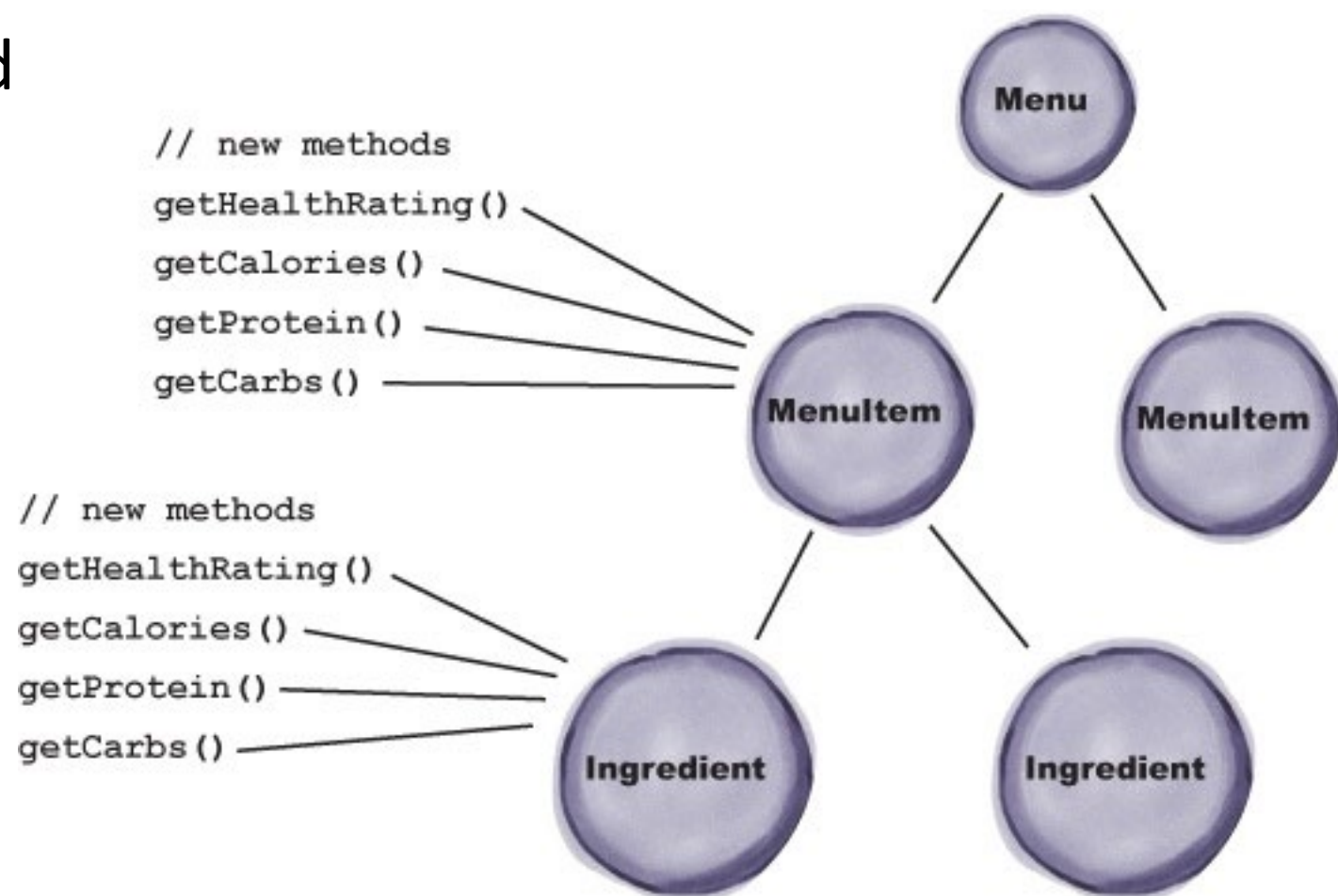
- Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require Initialize.
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.
- Prototype co-opts one instance of a class for use as a breeder of all future instances.
- Prototype is unique among the other creational patterns in that it doesn't require a class – only an object. Object-oriented languages like Self and Omega that do away with classes completely rely on prototypes for creating new objects.
- [https://sourcemaking.com/design\\_patterns/prototype](https://sourcemaking.com/design_patterns/prototype)

# Prototype – Key Points

- Hides the complexities of making new instances from the client
- Provides the option for the client to generate objects whose type is not known
- In some circumstances, copying an object can be more efficient than creating a new object
- Prototype should be considered when a system must create new objects of many types in a complex class hierarchy
- Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype.
- A drawback to using the Prototype is that making a copy of an object can sometimes be complicated

# Visitor

- The Visitor Pattern is used when you want to add capabilities to a composite of objects (and encapsulation is not important)
- Head First example:  
The diner example... Customers asking for nutritional information on meals and on ingredients.
- Initial thought is to add new methods to menu items and ingredients
- But concerns that this won't scale well with new items or new methods...

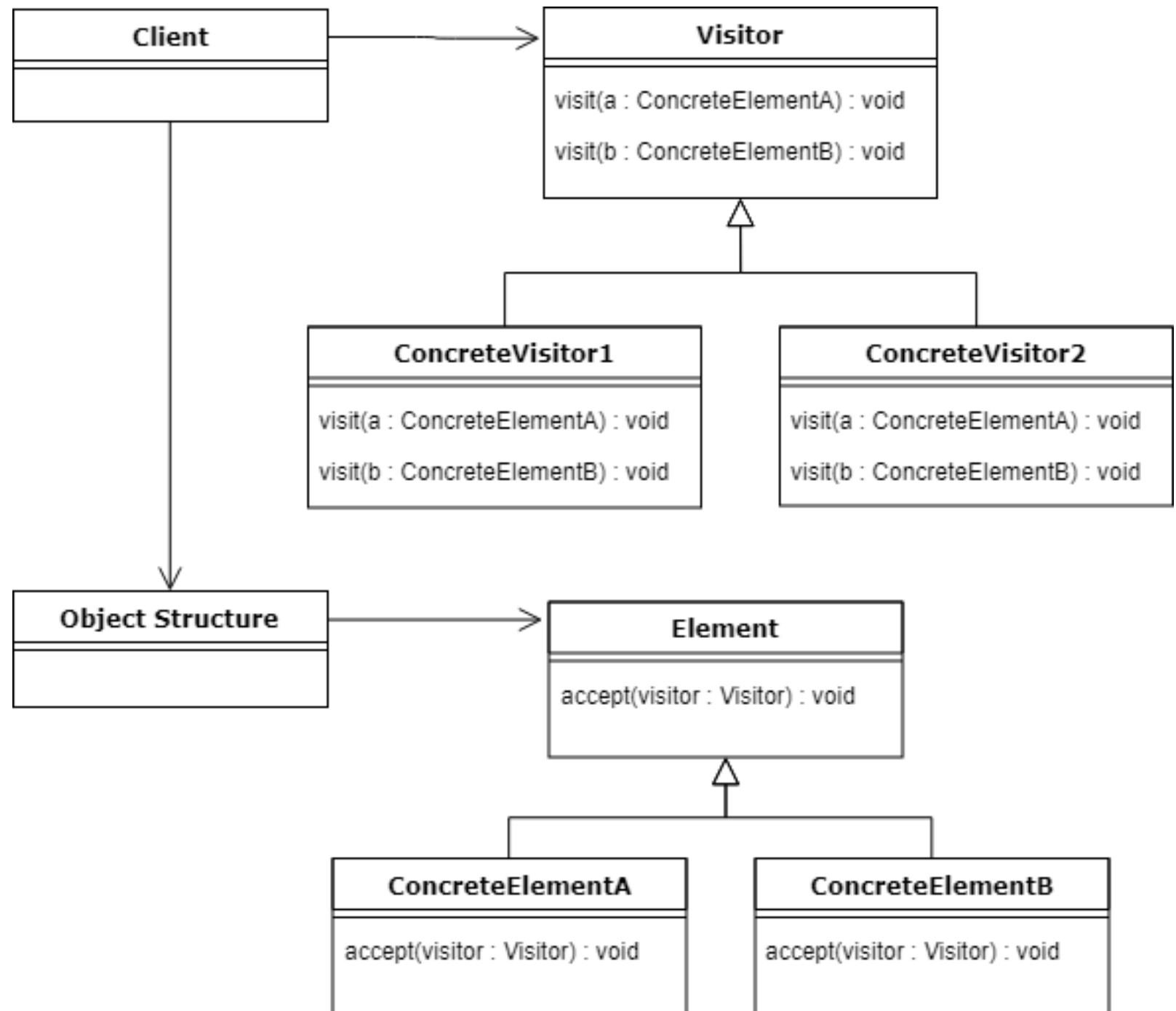


# Visitor Problem/Solution

- Problem: Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.
- You want to avoid "polluting" the node classes with these operations.
- And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.
- Solution: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- [https://sourcemaking.com/design\\_patterns/visitor](https://sourcemaking.com/design_patterns/visitor)

# Visitor Pattern: UML Class Diagram

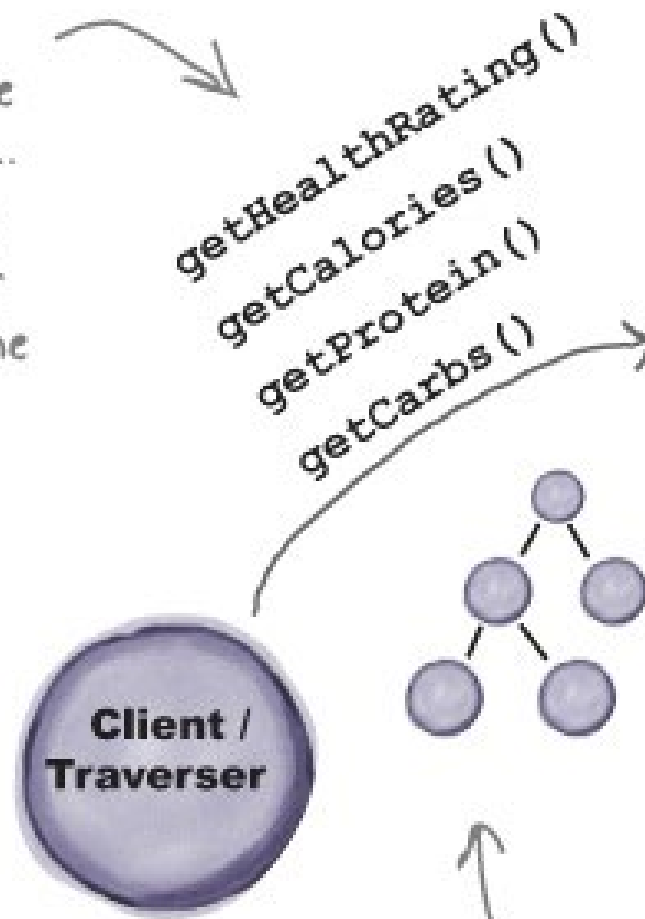
- Imagine that we have a composite object which consists of components; the object's structure is fixed – we either can't change it, or we don't plan to add new types of elements to the structure
- The Visitor design pattern adds a function which accepts the visitor class to each element of the structure
- That way our components will allow the visitor implementation to “visit” them and perform any required action on that element.
- In other words, we'll extract the algorithm which will be applied to the object structure from the classes.
- Consequently, we'll make good use of the Open/Closed principle as we won't modify the code, but we'll still be able to extend the functionality by providing a new Visitor implementation.



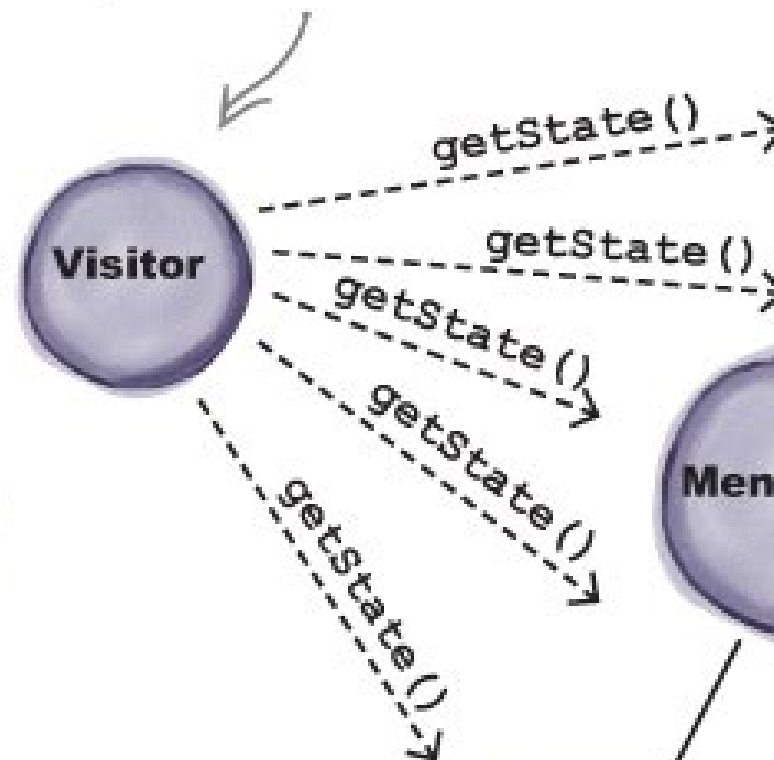


# Visitor applied

The Client asks the Visitor to get information from the Composite structure... New methods can be added to the Visitor without affecting the Composite.



The Visitor needs to be able to call `getState()` across classes, and this is where you can add new methods for the client to use.

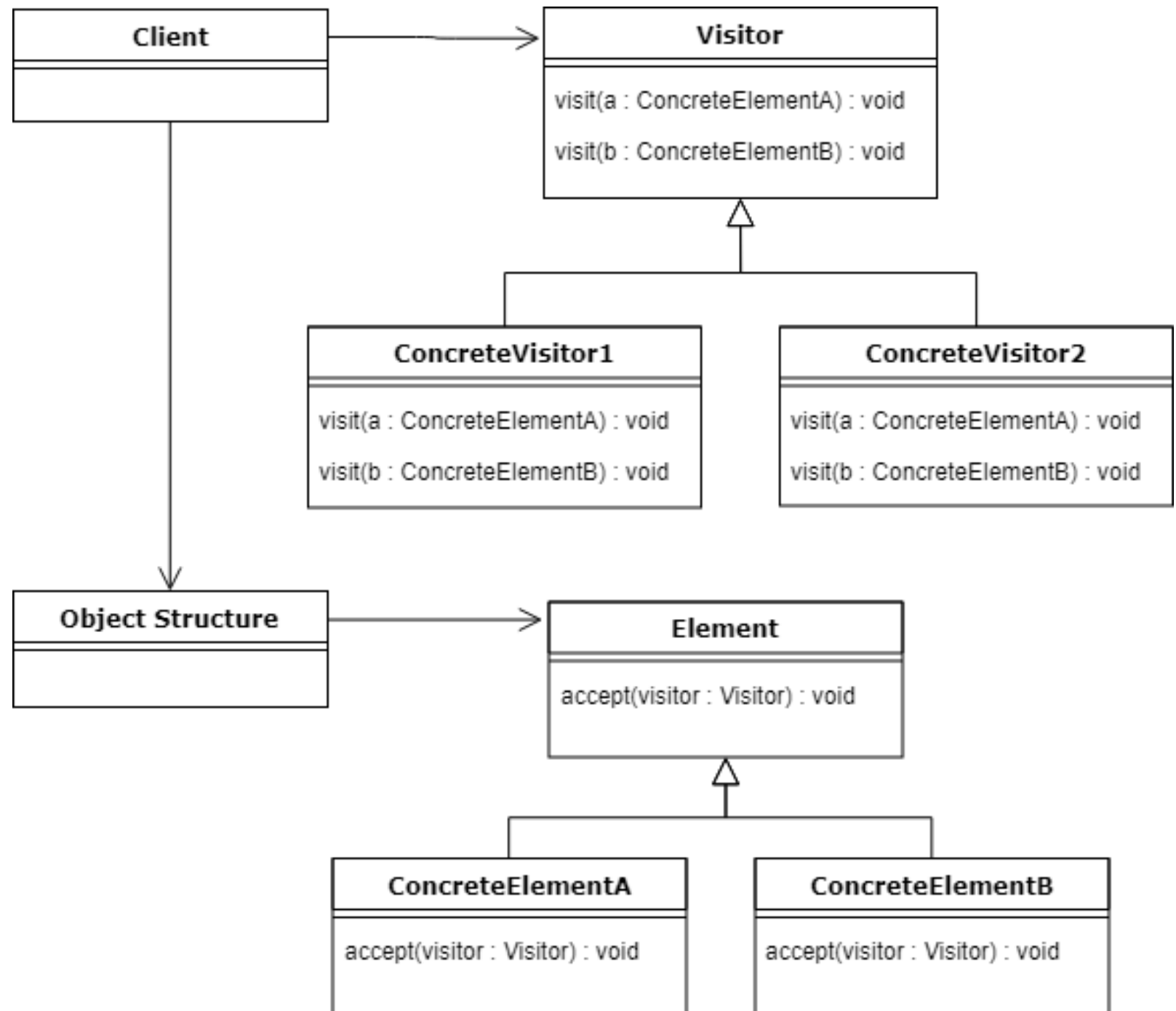


All these composite classes have to do is add a `getState()` method (and not worry about exposing themselves).

The Traverser knows how to guide the Visitor through the Composite structure.

# Visitor in Use

- When the client needs an operation to be performed, they create an instance of the Visitor object, traversing the composite structure, and calling the accept() method on each Element object, passing the Visitor object
- The accept() method causes flow of control to find the correct Element subclass
- Then when the visit() method is invoked, flow of control is vectored to the correct Visitor subclass. accept() dispatch plus visit() dispatch equals double dispatch

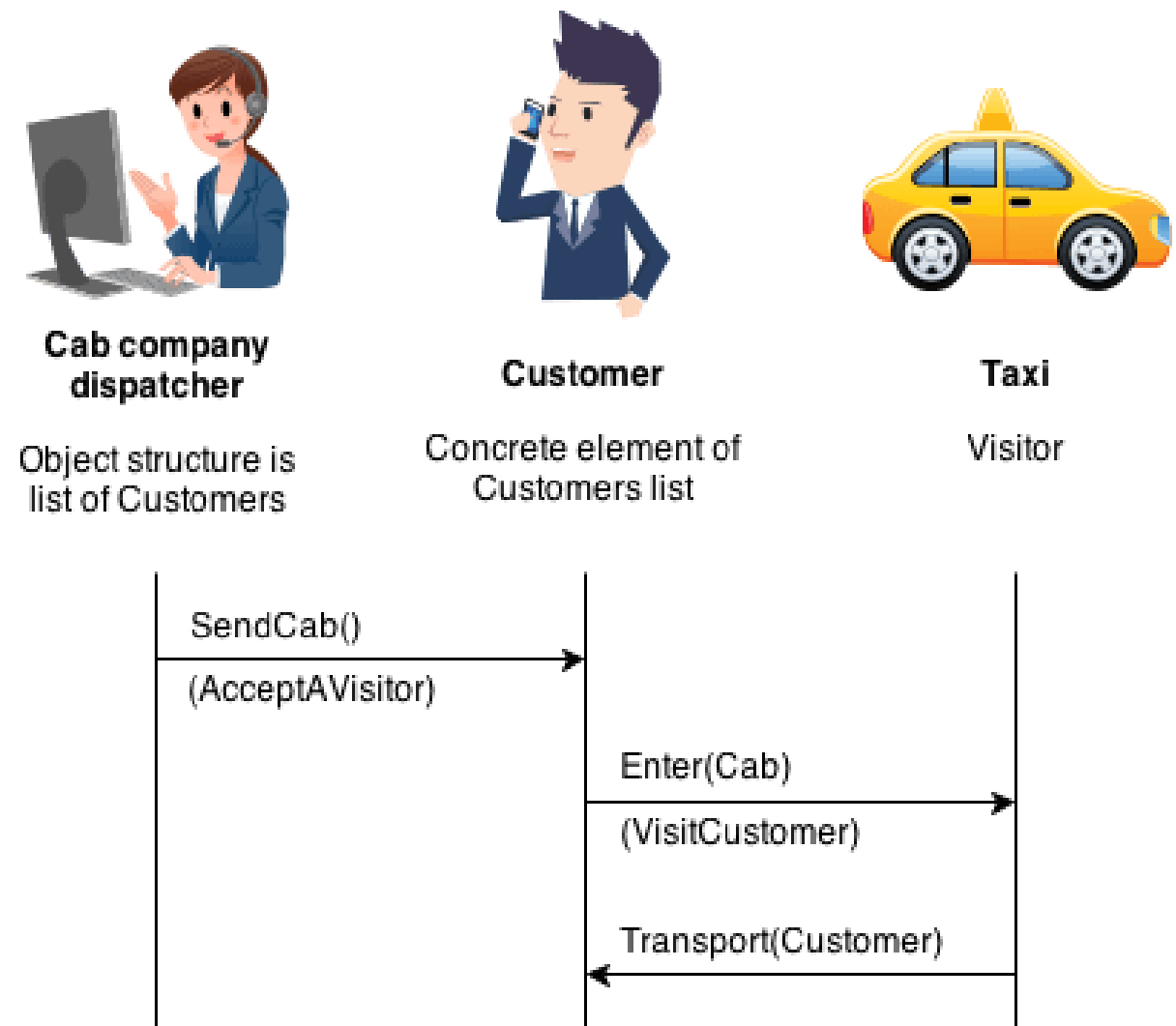


# Visitor Considered

- Visitor implements "double dispatch"
- OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver
- In "double dispatch", the operation executed depends on: the name of the request, and the type of TWO receivers (the type of the Visitor and the type of the element it visits).
- An acknowledged objection to the Visitor pattern is that it represents a regression to functional decomposition - separate the algorithms from the data structures
  - Alternate view: promoting non-traditional behavior to full object status.

# Another Visitor Example

- The Visitor pattern represents an operation to be performed on the elements of an object structure without changing the classes on which it operates
- This pattern can be observed in the operation of a taxi company
- When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer
- Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is



[https://sourcemaking.com/design\\_patterns/visitor](https://sourcemaking.com/design_patterns/visitor)

# Visitor in Java - Elements

interface ItemElement

```
{
    public int accept(ShoppingCartVisitor visitor);
}
```

class Book implements ItemElement

```
{
    private int price;
    private String isbnNumber;
    public Book(int cost, String isbn)
    {
        this.price=cost;
        this.isbnNumber=isbn;
    }
    public int getPrice()
    {
        return price;
    }
    public String getIsbnNumber()
    {
        return isbnNumber;
    }
    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }
}
```

class Fruit implements ItemElement

```
{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm)
    {
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg()
    {
        return pricePerKg;
    }

    public int getWeight()
    {
        return weight;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }
}
```

# Visitor in Java – Visitor, Client

```
interface ShoppingCartVisitor
{
    int visit(Book book);
    int visit(Fruit fruit);
}
class ShoppingCartVisitorImpl implements ShoppingCartVisitor
{
    @Override
    public int visit(Book book)
    {
        int cost=0;
        //apply 5$ discount if book price is greater than 50
        if(book.getPrice() > 50)
        {   cost = book.getPrice()-5;   }
        else cost = book.getPrice();
        System.out.println("Book ISBN::"+
            book.getIsbnNumber() + " cost =" +cost);
        return cost;
    }
    @Override
    public int visit(Fruit fruit)
    {
        int cost = fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName() + " cost = " +cost);
        return cost;
    }
}
```

```
class ShoppingCartClient
{
    public static void main(String[] args)
    {
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"),
            new Book(100, "5678"), new Fruit(10, 2, "Banana"),
            new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost = " +total);
    }

    private static int calculatePrice(ItemElement[] items)
    {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item : items)
        {
            sum = sum + item.accept(visitor);
        }
        return sum;
    }
}
```

Output:

```
Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160
```

# Visitor and other Patterns

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.
- The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.
- [https://sourcemaking.com/design\\_patterns/visitor](https://sourcemaking.com/design_patterns/visitor)

# Visitor – Key Points

- Allows you to add operations to a Composite structure without changing the structure itself
- Adding new operations is relatively easy
- The code for operations performed by the Visitor is centralized
- The Composite classes' encapsulation is broken when the Visitor is used
- Because the traversal function is involved, changes to the Composite structure are more difficult
- The Visitor pattern makes adding new operations (or utilities) easy - simply add a new Visitor derived class, but... if the subclasses in the aggregate node hierarchy are not stable, keeping the Visitor subclasses in sync requires a prohibitive amount of effort
- Visitor downside: its usage **makes it more difficult to maintain the code if we need to add new elements to the object's structure**
- For example, if we add new a new element, then we need to update all existing concrete visitors with the new method desired for processing this element
- When using this pattern, the business logic related to one particular object gets spread over all visitor implementations



# Next steps

- Grading...
- Project 4 due noon Wed 11/4
  - First part of three for the semester project
  - Get started sooner than later – a lot of parts
- Graduate Draft Presentation due noon Wed 11/11
  - Expecting a thorough research effort, not a surface topic review
- Quiz 7 up today, due next Wed 11/4
- Article Reviews are available for extra bonus points...
- New discussion topic is up... Visit Piazza often – it is for your participation grade, so participate!
- Coming up: Finishing up the patterns this week – Prototype, Visitor, more OO concepts, bonus exercises, more...
- If you need help – Office hours, Piazza, e-mail – we are here for you!