

# Behavioral Patterns

Iterator

Visitor

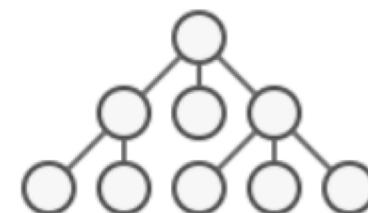
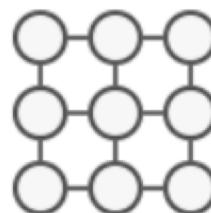
```
class ProductCatalog<T> {  
    private Node<T> productRoot; // This is a BST  
    public Node<T> getElements() {  
        return this.productRoot; // DESIGN FLAW  
    } //... methods to manage catalog item like add in BST  
}  
class Client {  
    public <T> void printCatalog (ProductCatalog<T> catalog) {  
        Node<T> root = catalog.getElements();  
        this.printInOrder(root);  
        //Client is forced to write its own logic, go left/right etc  
    }  
    private <T> void printInOrder(Node<T> node) { /* ... */ }  
}
```

# Iterator Pattern

---

## ■ Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- An *aggregate object* is an object that contains other objects for the purpose of grouping those objects as a unit.



# Why talk about iterator?

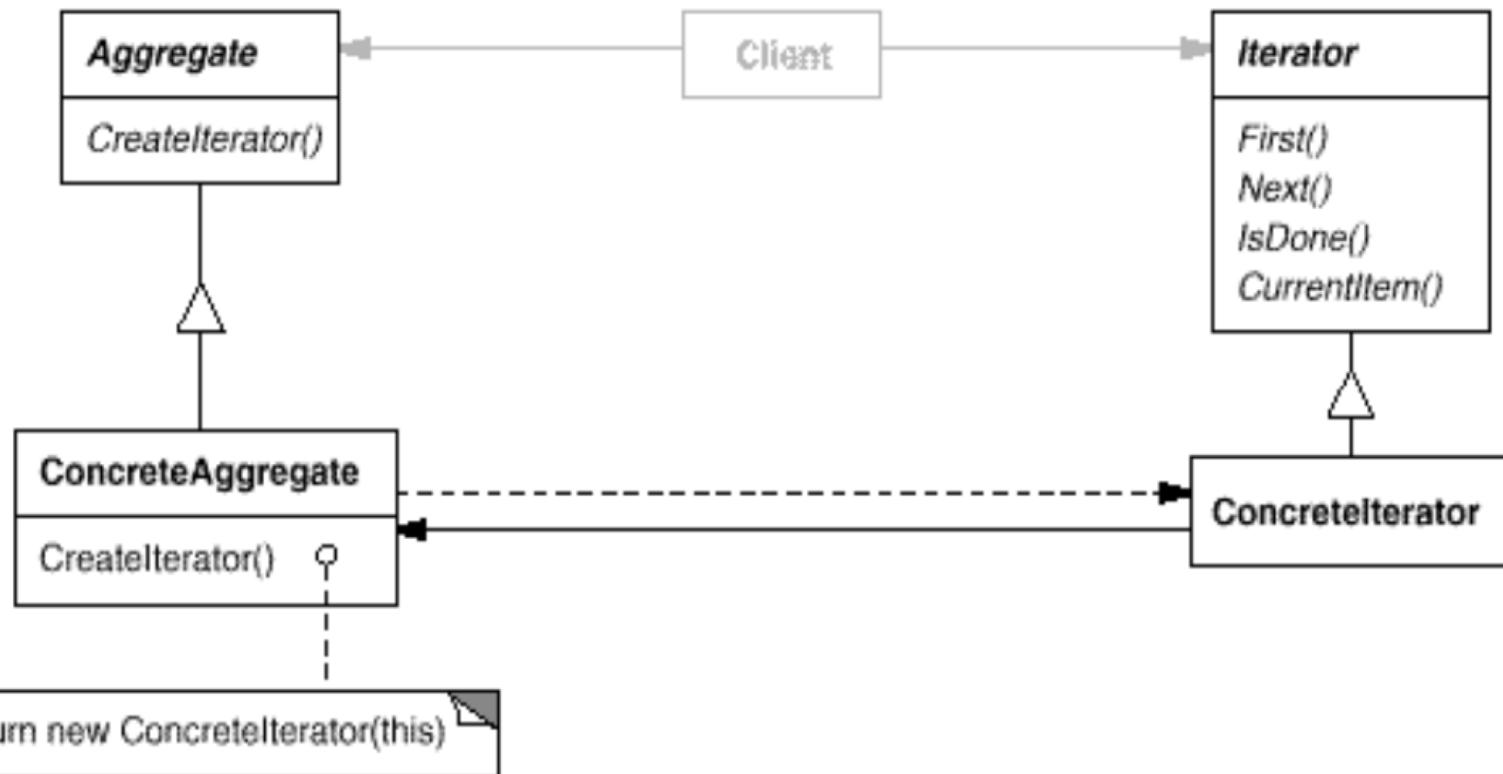
---

- We can write our own iterator!
  - Customized iterator for our own classes
  - Customized traversal algorithms for our own classes
- When should we use an iterator?
- Why using iterators make sense?

You know *how to use it*. Today, *why we must have it*

- Suggestion: use an existing iterator if possible.

# Structure –GoF



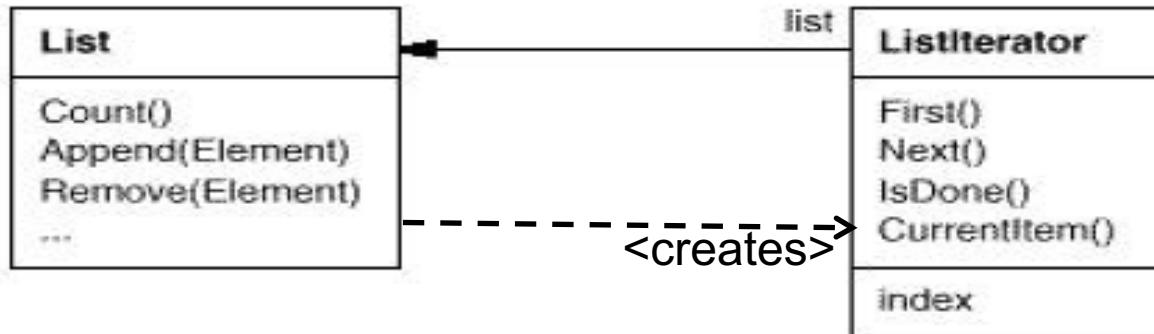
iterator(); next(); hasNext(); Java alternative  
begin(); it++; end(); C++ alternative  
Foreach loops are using iterators behind the scene

# Why iterator anyway?

---

- Iterator **hides internal representation**
  - Iterator of a any collection has the same interface
  - List, Set, PriorityQueue ...all create iterator with the same interface
- Encapsulate traversal of a collection
  - Clients do not have to implement them
- Simpler aggregates: when the collection **itself provides traversal**, it will be a **mess**
  - Many traversal happen at the same time
  - Many object will call next(), how to save the position of the traversal for each of them?

# Iterator



- Traveling task is on the iterator, not on the aggregate
- **Simplifying** the aggregate interface
- Relieves the aggregate from the traversing responsibility (more cohesive)
- **Multiple** traversal

```
List<Object> lst=new ArrayList<Object>();
```

```
Iterator it1=lst.iterator(); it1.next(); //traversal one
```

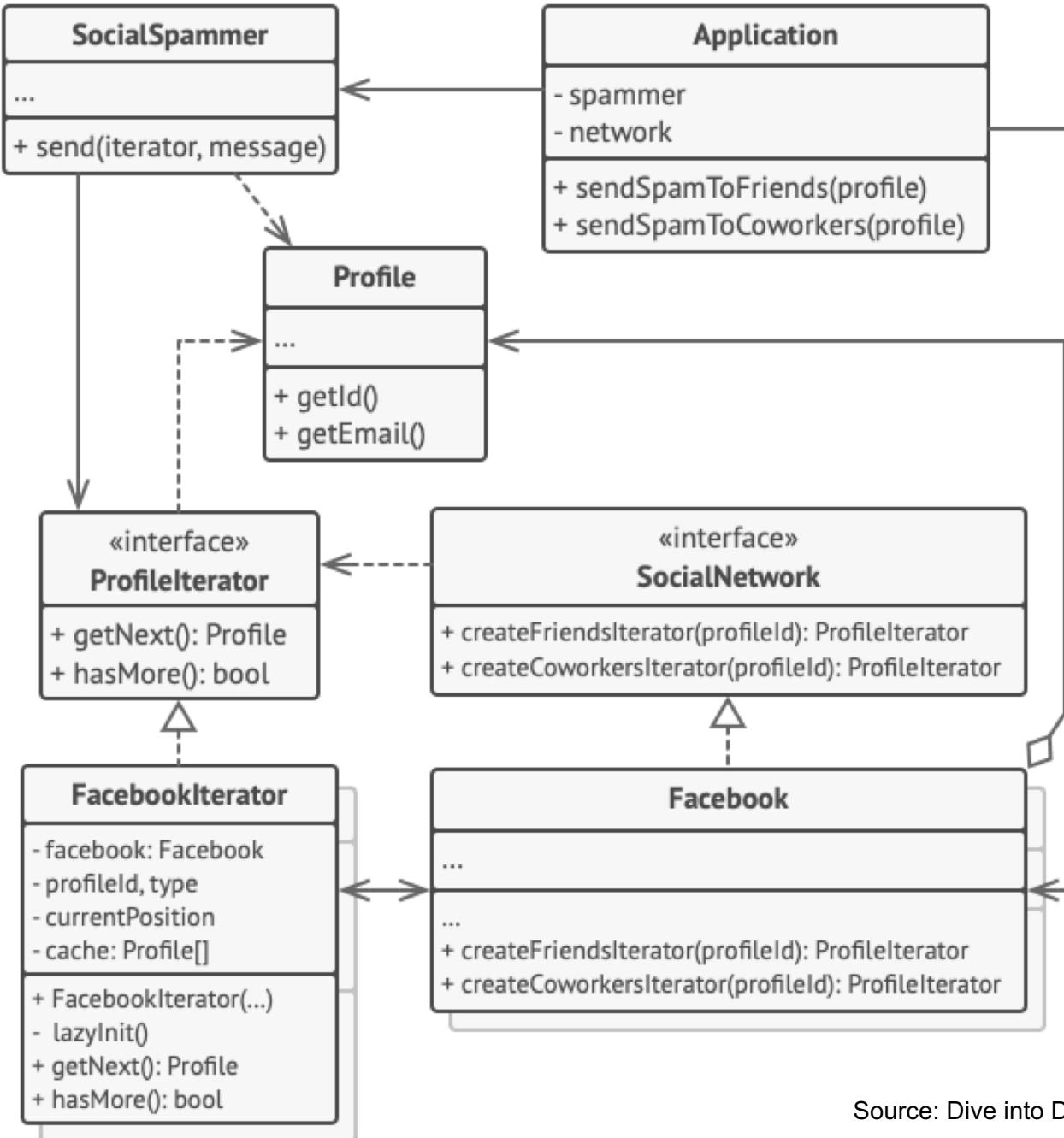
```
Iterator it2=lst.iterator(); it2.next(); it2.next(); //traversal two
```

# Different traversal

---

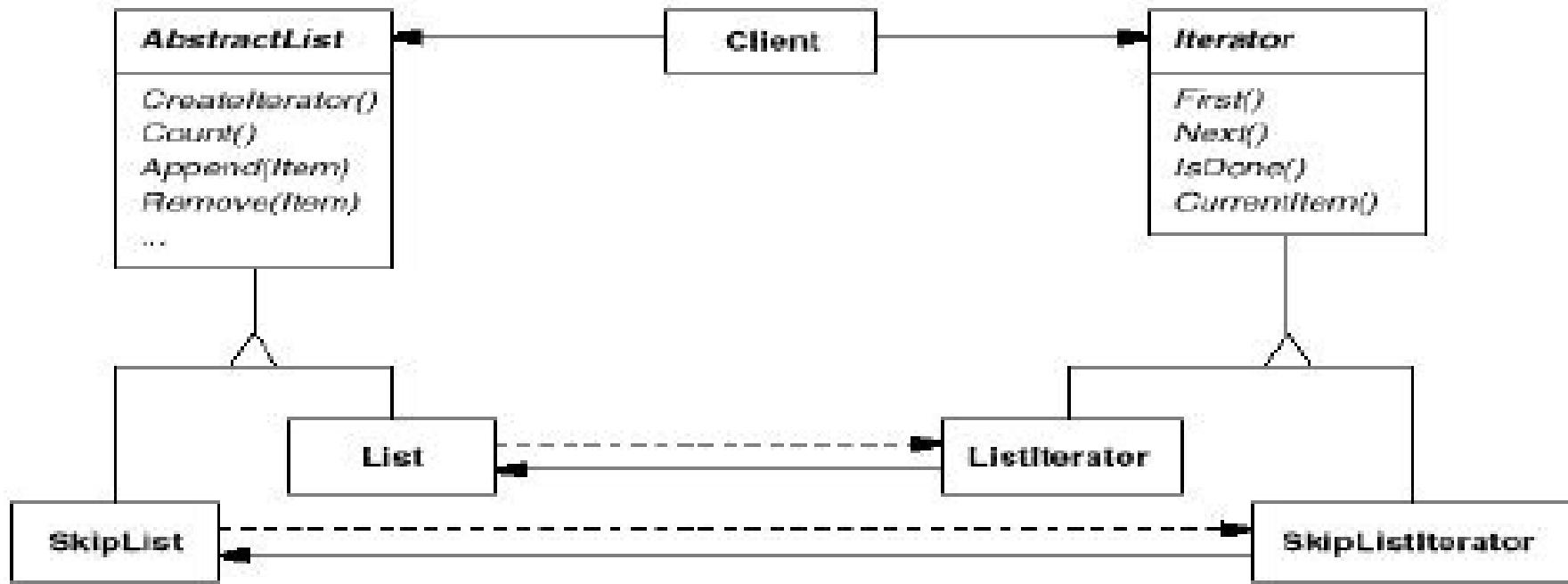
- Complex aggregates may be traversed many ways
- Example: Tree
  - in-order, pre-order, post-order
- Other traversals
  - Forward, backward
  - Skip even, skip odd , .....
- For each algorithm, have one concrete iterator
- `java.util.TreeSet` has 2 iterators
  - `iterator()`: returns an Iterator, for ascending order
  - `descendingIterator()`: returns an Iterator, for descending order

# Example: Writing your own iterator



If the language provides iterator that fits, **no** need to reinvent the wheel.

# Structure (polymorphic iterator)

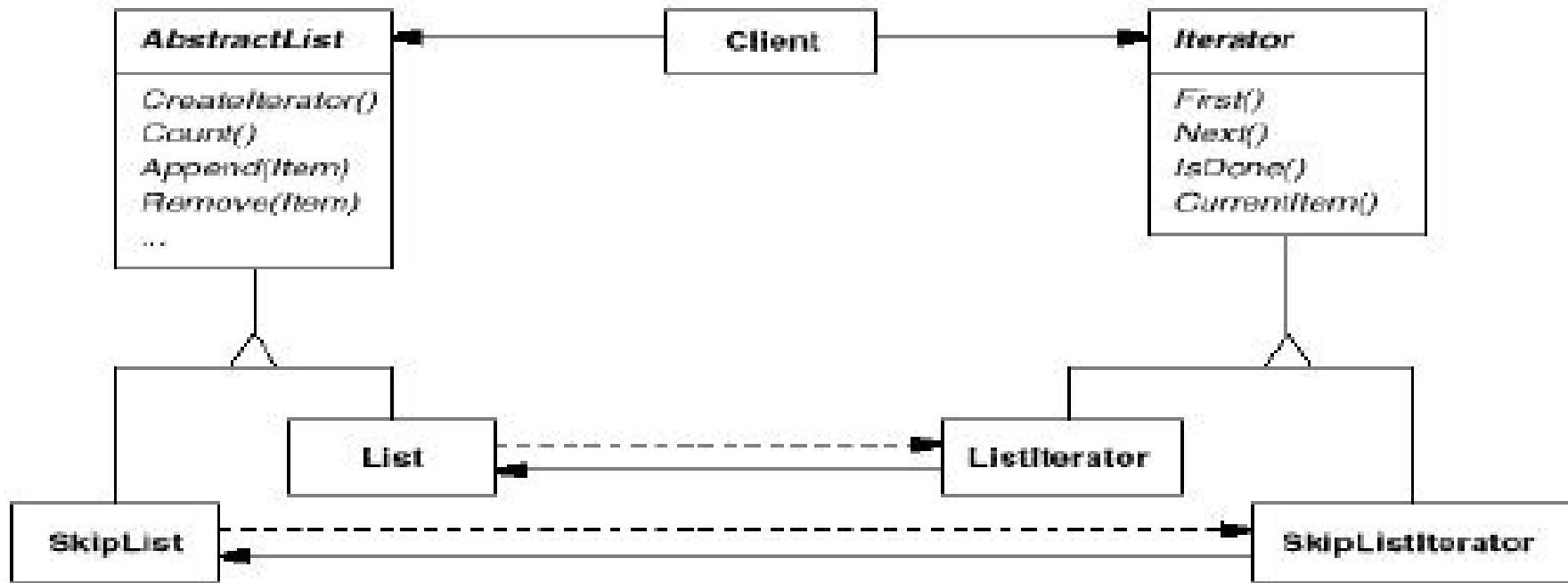


Each concrete aggregate is responsible to create the appropriate concrete iterator

The iterator implementation w.r.t. underlying data structure.

- e.g. cursor : index of an array
- e.g. cursor: pointer to a node

# Structure (polymorphic iterator)



Each concrete aggregate is responsible to create the appropriate concrete iterator

`CreateIterator()` method lets the subclasses to decide which concrete iterator to create. **Is this familiar?**

# Iterator -Consequences

---

- Simplifies the interface of the Aggregate by not polluting it with traversal methods
- Supports multiple, concurrent traversals
- Supports variant traversal techniques

# Language support

---

- C++ STL library has 6 iterators
  - `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, `random_access_iterator`, `contiguous_iterator`
- Java has basic 3 iterators
  - `Collection<t>` extends `Iterable<T>` interface, all collections return an iterator
  - There is indirect support to create iterators for arrays
- STL weakens the iterator intent “sequential” access, also support bidirection
- Java `ListIterator` is bidirectional

# Related patterns

---

- Iterator can traverse a **Composite**.
- Polymorphic Iterators rely on **Factory Methods** to instantiate the appropriate Iterator subclass.
- **Memento** is often used in conjunction with Iterator.
  - An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.
- **Visitor** (next) and Iterator both traverse a structure

# implementation issues

---

- Can the aggregate be modified while a traversal is ongoing?
  - An iterator that allows insertion and deletions without affecting the traversal and without making a copy of the aggregate is called a **robust** iterator.
  - C++ STL provides such iterators
  - Java throws ConcurrentModificationException
- Use library iterators to avoid pitfalls
  - Might want to implement iterator for your own complex data structure
    - When there are traversal code in many places
  - Might use library iterators inside your own iterator

# implementation issues

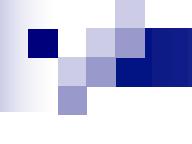
---

- Who defines the traversal algorithm?
  - The iterator => more common; easier to have variant traversal techniques
    - E.g ListIterator::next() {return list.get(current);}
  - The aggregate => iterator only keeps state of the iteration.
    - A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.
- Iterator may need Collection to expose its internals --friend

# Iterator and multithreading

---

- Iterator does not raise special problems when the collection used from different threads as long the collection is not changed.
- Collection changes and iterator:
  1. A new element is added/removed to the collection (at the end). The iterator should be aware of the new size of the collection and to iterate till the end.
  2. A new element is added/removed to the collection before the current element. In this case all the iterators of the collection should be aware of this.
- Multithreading iterator should be a robust iterator and have locking mechanism (synchronization)
- Languages support iterators for multi-threaded. Use them.



**VISITOR**

# General Problem

---

- There are many objects of different types
  - Some share the same interface, some not.
- We want to apply operation to several or all of them
  - Cannot foresee all the operations we will need
- Additionally
  - These operations are **not primary** responsibilities of the objects
  - We want to add new operations without modifying the object classes

# Example Scenarios

---

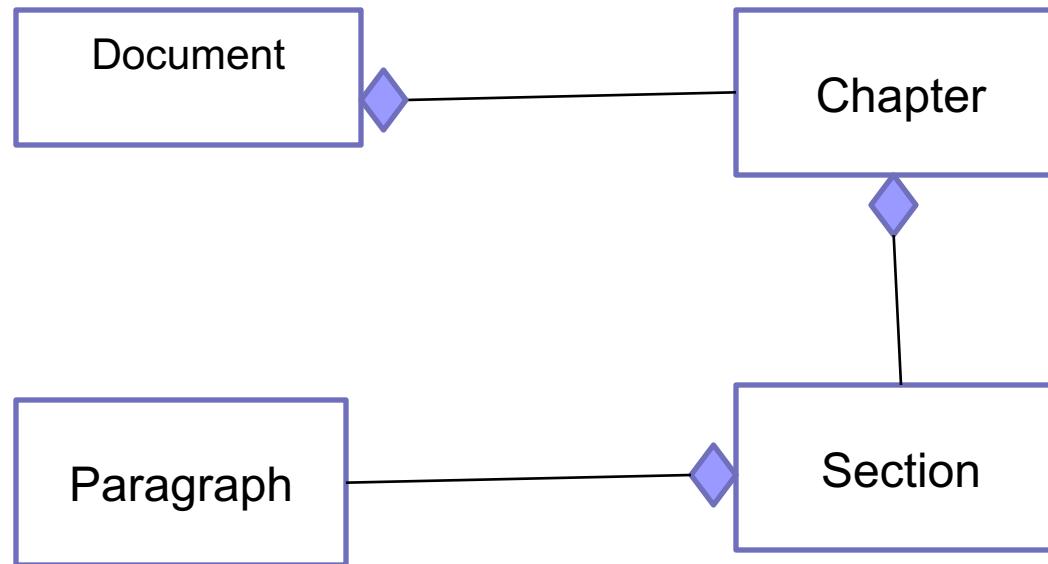
## Scenario 1:

- There are a collection of chapter, section, paragraph classes
- Adding secondary operations: Table of Contents, word count operation

## Scenario 2:

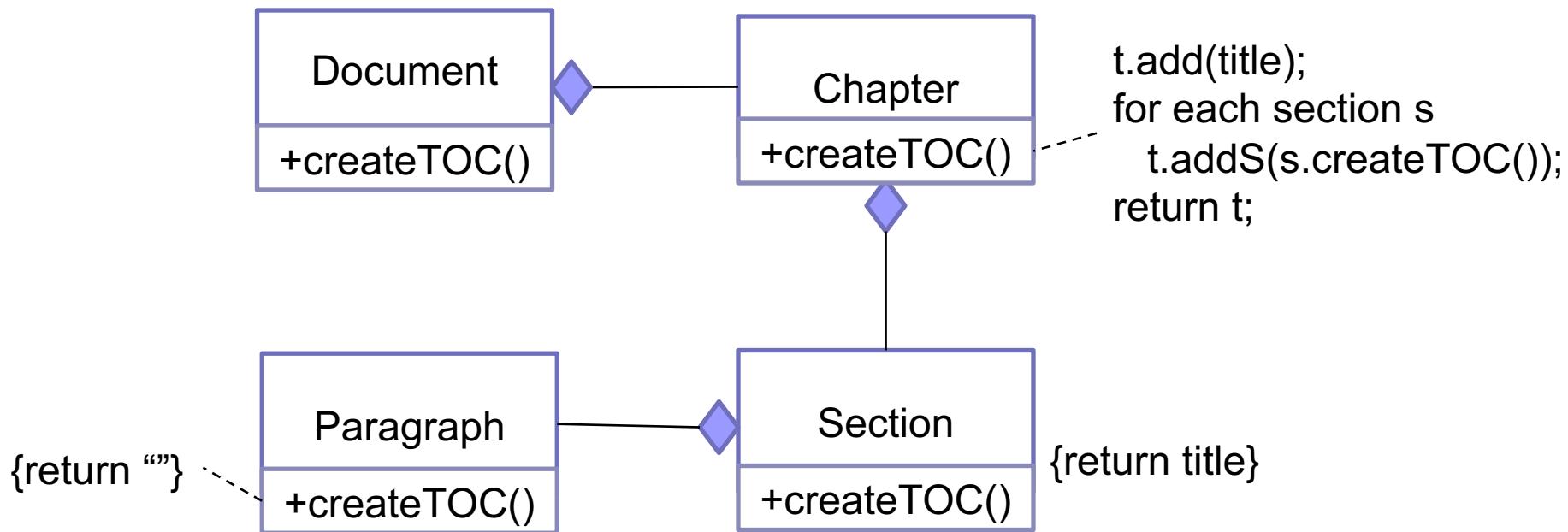
- There are a different types of products in an e-commerce site
- Adding discount, exporting info to different formats.

# Creating Table of Contents

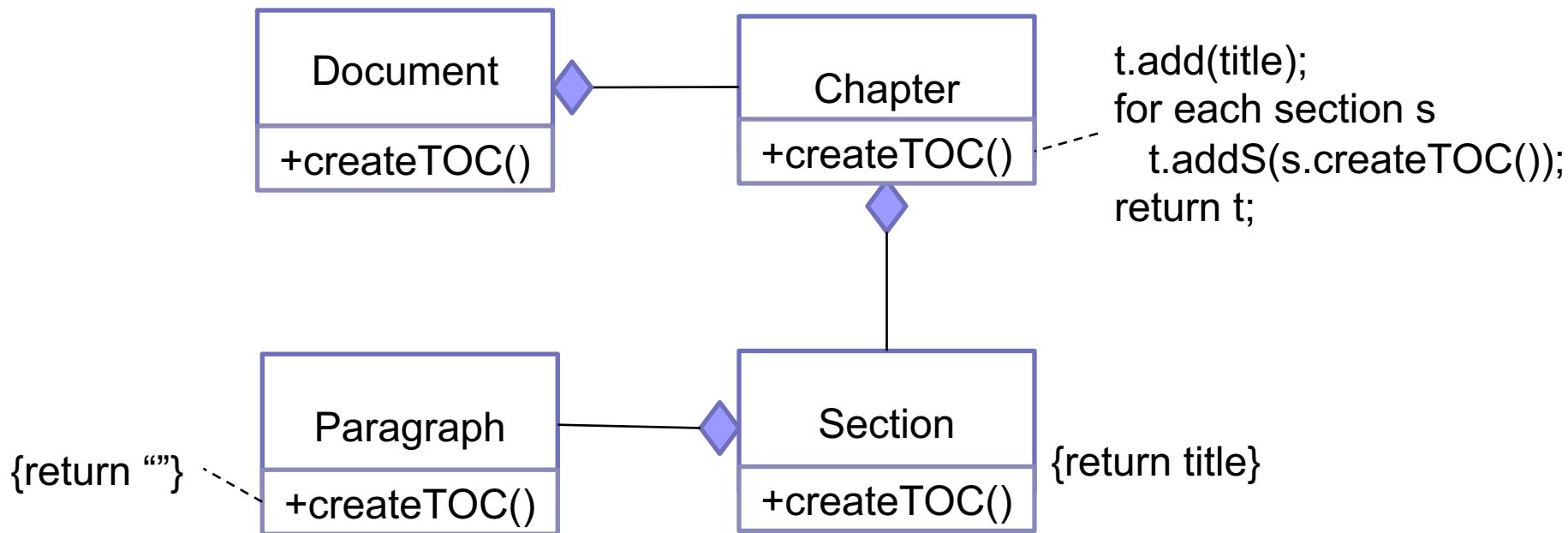


- Classes contain methods related to edit, format and display texts
- I want to add the functionality “create table of Contents”

# Attempt 1:



# Attempt 1:



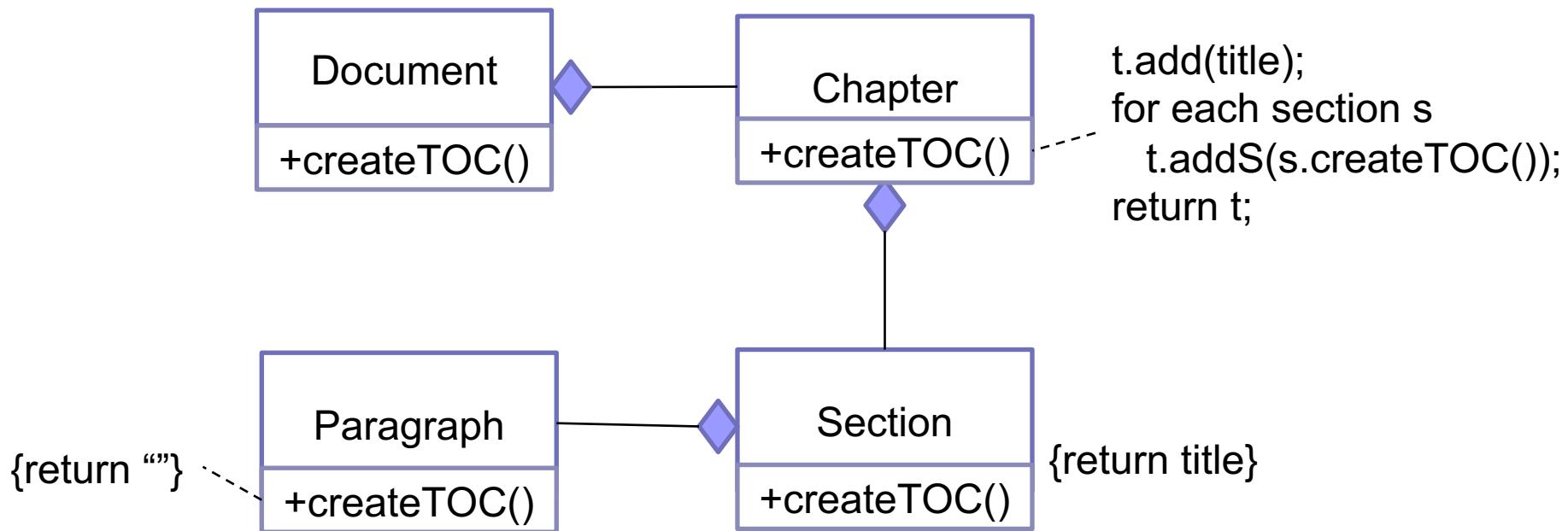
- Need to change the code of all classes.
  - What if I am not permitted to change these code?
- createToC logic is spread all over
- This is not the primary job of these classes (SRP?)
- There will be word count operation later. (modify all)

# Constraints

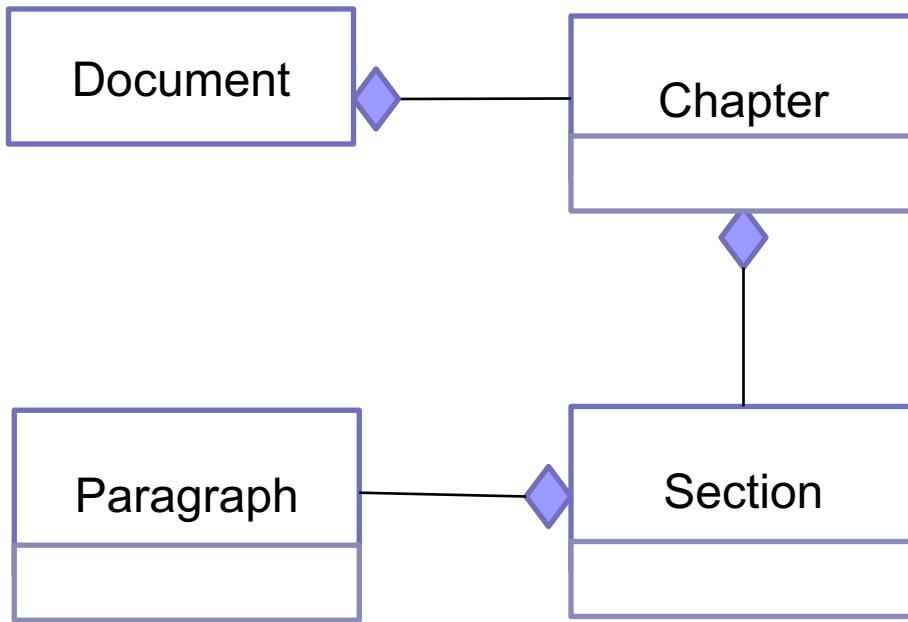
---

- Keep the classes cohesive
  - Focus on primary job and provide a point to add secondary responsibilities
    - I will add count words later
- Put the gathering ToC info job in its dedicated class
  - Bring the distributed logic in one place
- Preemptive
  - mechanism for adding new operations later
  - Might not have permission to change the classes

# Attempt 1:

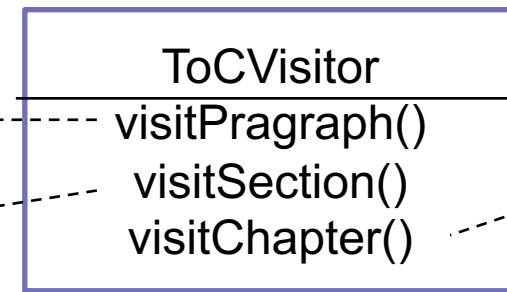


# Towards Solution 1:



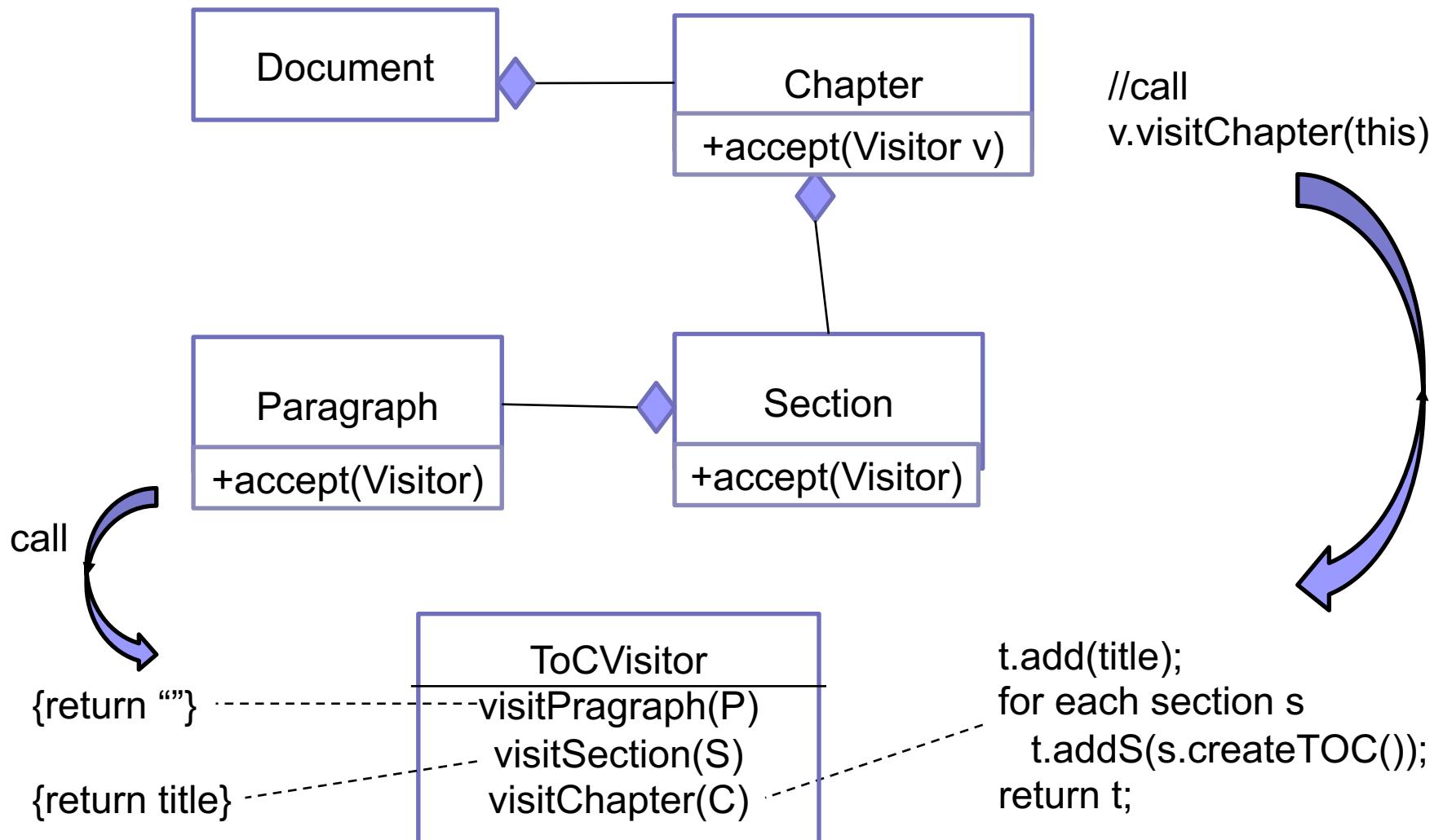
{return “”}

{return title}

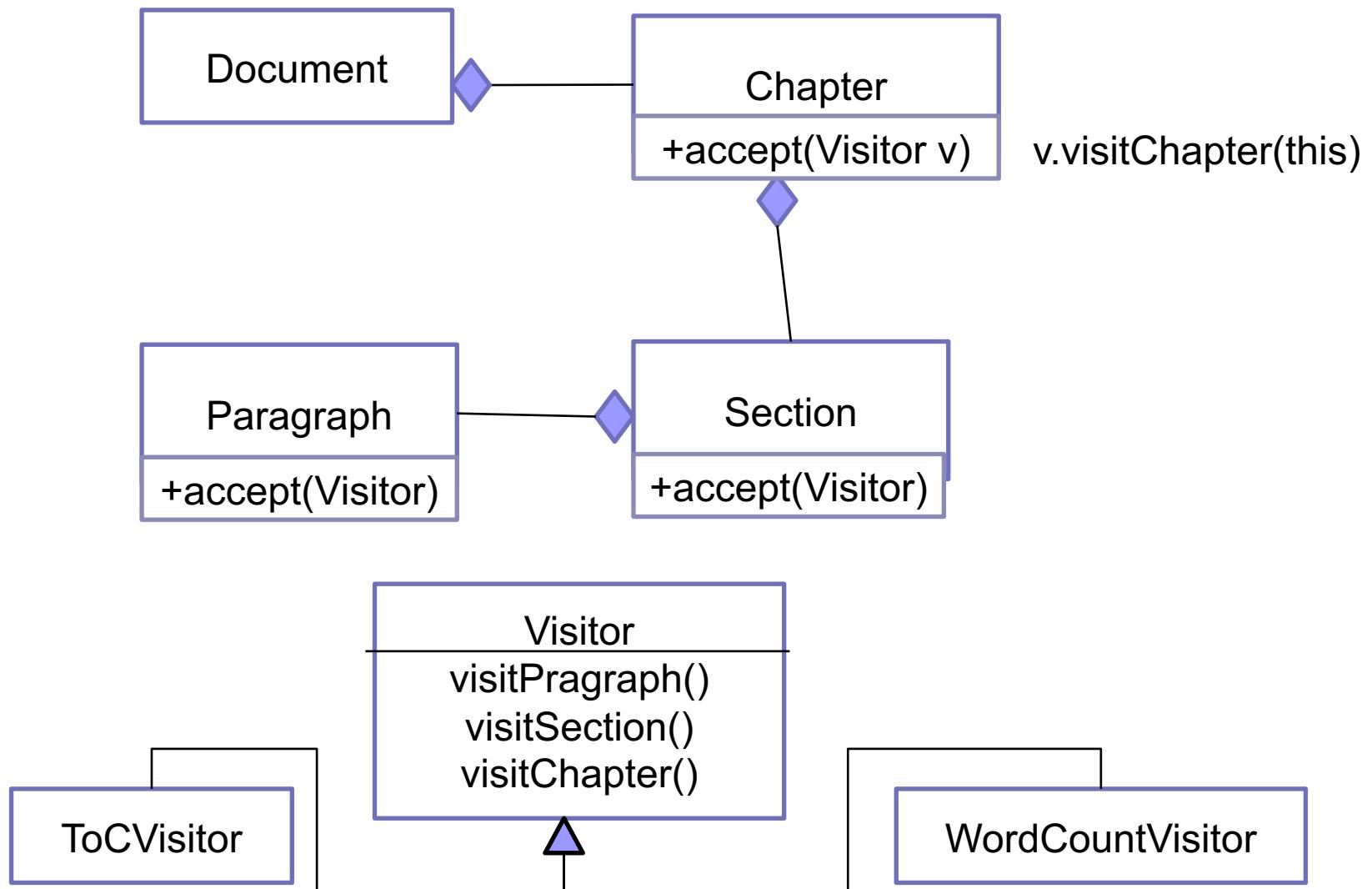


```
t.add(title);
for each section s
    t.addS(s.createTOC());
return t;
```

# Towards Solution 2:



# Extensible Solution:



# Visitor

---

## ■ Intent:

- Represent an operation to be performed on the elements of an object structure.
  - Visitor lets you define new operation without changing the classes the elements.
- 
- Allows adding further operations to objects without modifying them.

# Visitor

---

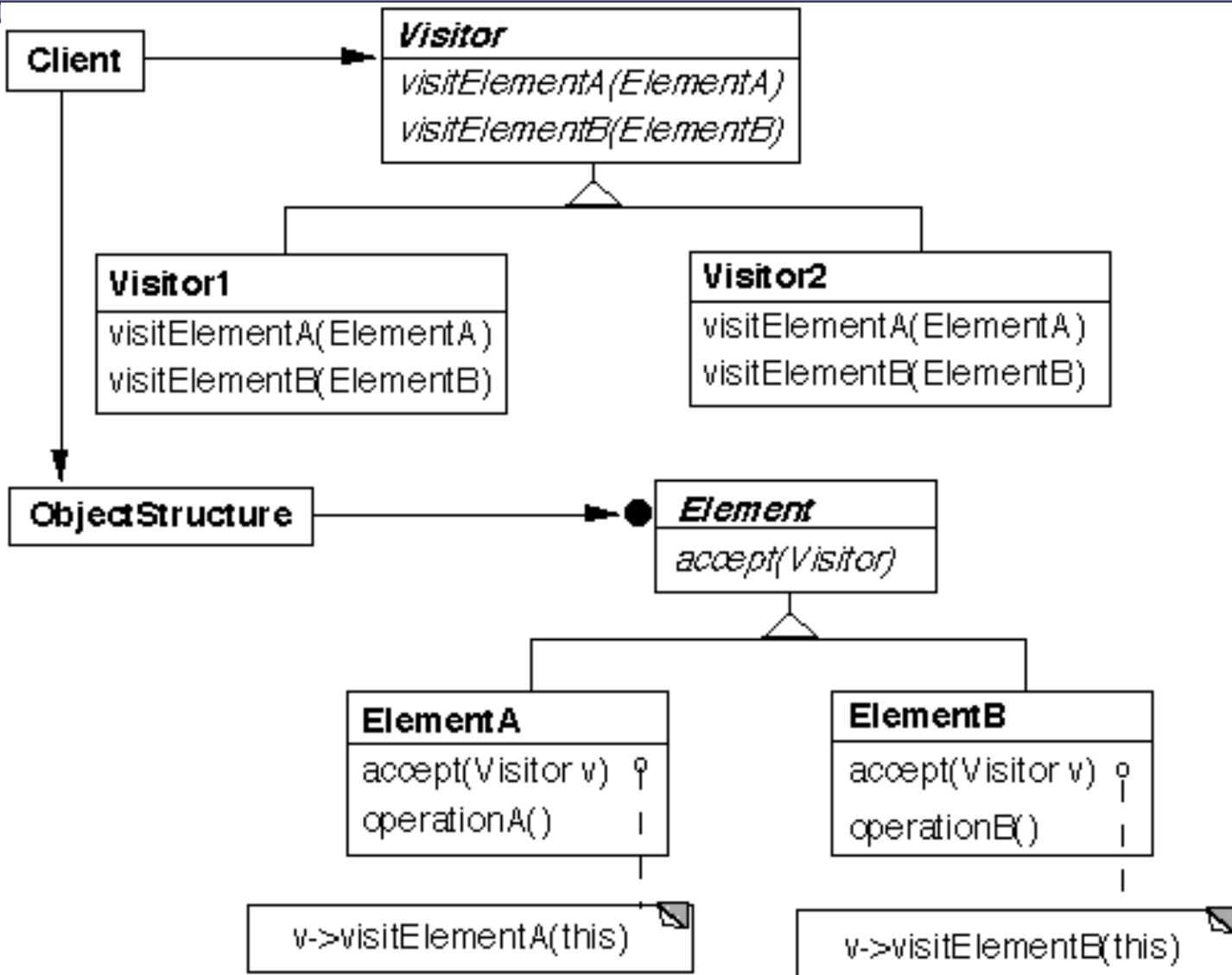
## ■ Intent:

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define new operation without changing the classes the elements.

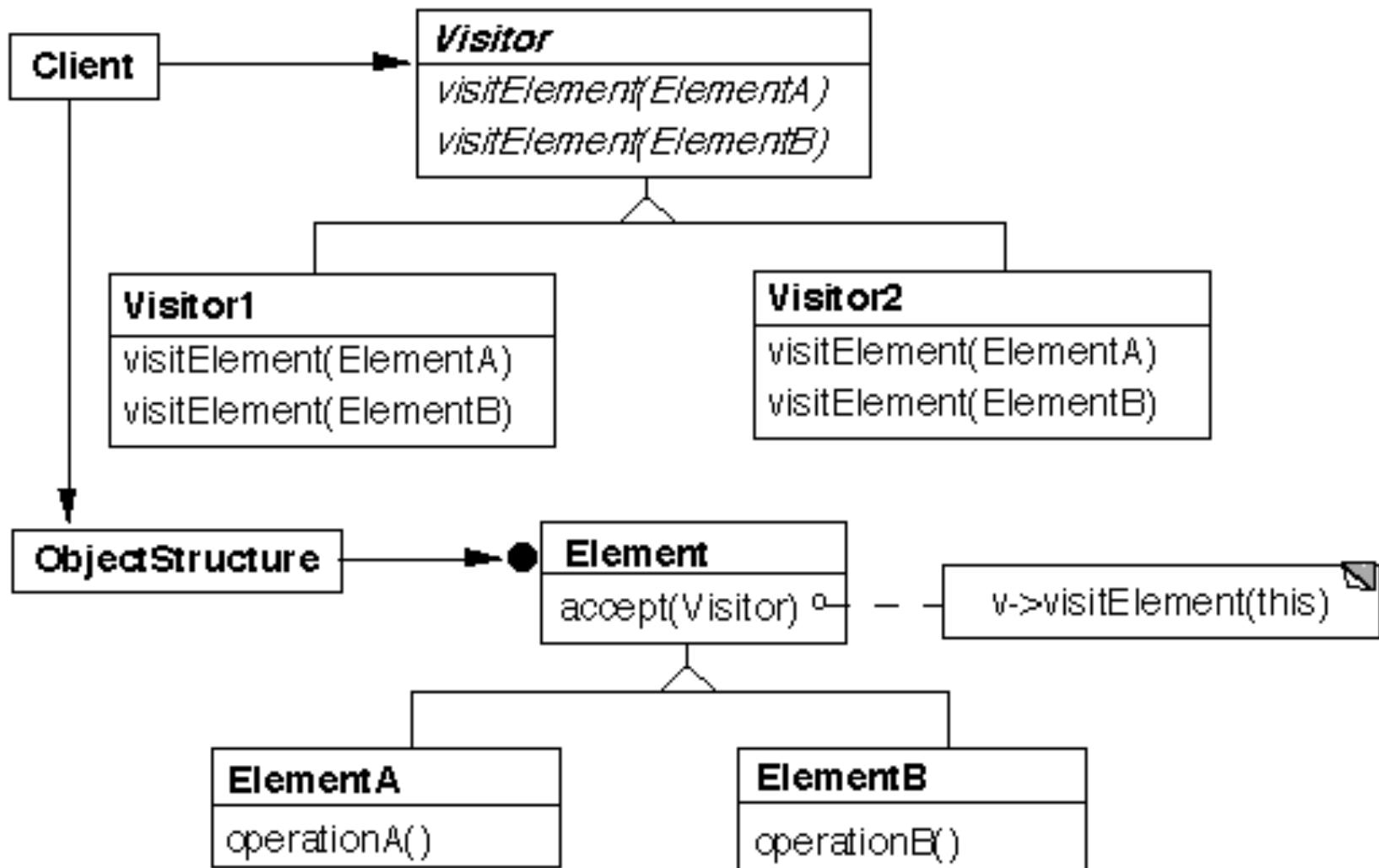
## ■ Applicability: When..

- Object structure has many classes with different interfaces
- Don't want to pollute with many unrelated operations – ISP and SRP
- Classes defining the object structure rarely change but you want to add new operations.
  - You may not be allowed to change

# Visitor – not using overloading



# Visitor structure (overload visit)



Using overloading

# Visitor - Collaborations

---

- We have visitor that collects ToC information from chapter, section, paragraph
  - `visit(Chapter c), visit(Section), visit(Paragraph)`
  - They access object's data `c->getTitle();`
- We have object structure that chooses the right visit method when receiving a visitor
  - `accept(Visitor& v){ v.visit(this);}`
- Need a mechanism for visitor objects to **traverse** the objects – not yet defined
  - 3 options

# Visitor -Traversals

---

## 1. Traversing different types of objects

- Some are data structure or collection

```
for (Node node : collection) {  
    node.accept(visitor);
```

```
}
```

- `Iterator` is very helpful

- Some not

```
obj1.accept(visitor);
```

```
obj2.accept(visitor);
```

- Neither visit nor accept method makes the traversal

# Exercise 1:

---

- Asteroid, Spaceship, Alien classes
- Print names of all

```
int main(){
```

```
    list<Asteroid*>as=Factory.getInstance().makeAsteroid(10);  
    Spaceship ship;  
    list<Alien*>aliens=Factory.getInstance().makeAlien(5);  
    Visitor* visitor=new PrintVisitor();  
    for(auto asteroid: as) asteroid->accept(*visitor);  
    ship.accept(*visitor);  
    for(auto alien: aliens) alien->accept(*visitor);  
}
```

Implement the visitor interface, visit alien, and Alien::accept

# Visitor -Traversals

---

## 1. Traversing different types of objects and collections

```
obj1.accept(visitor);  
for (Node node : collection) {  
    node.accept(visitor);}
```

- Iterator is very helpful
- Neither visit() nor accept() does traversal

## 2. Traversing a recursive structure or a composite pattern

- Structure propagates accept() using existing links
- Leveraging the composite
  - **root.accept(visitor)**

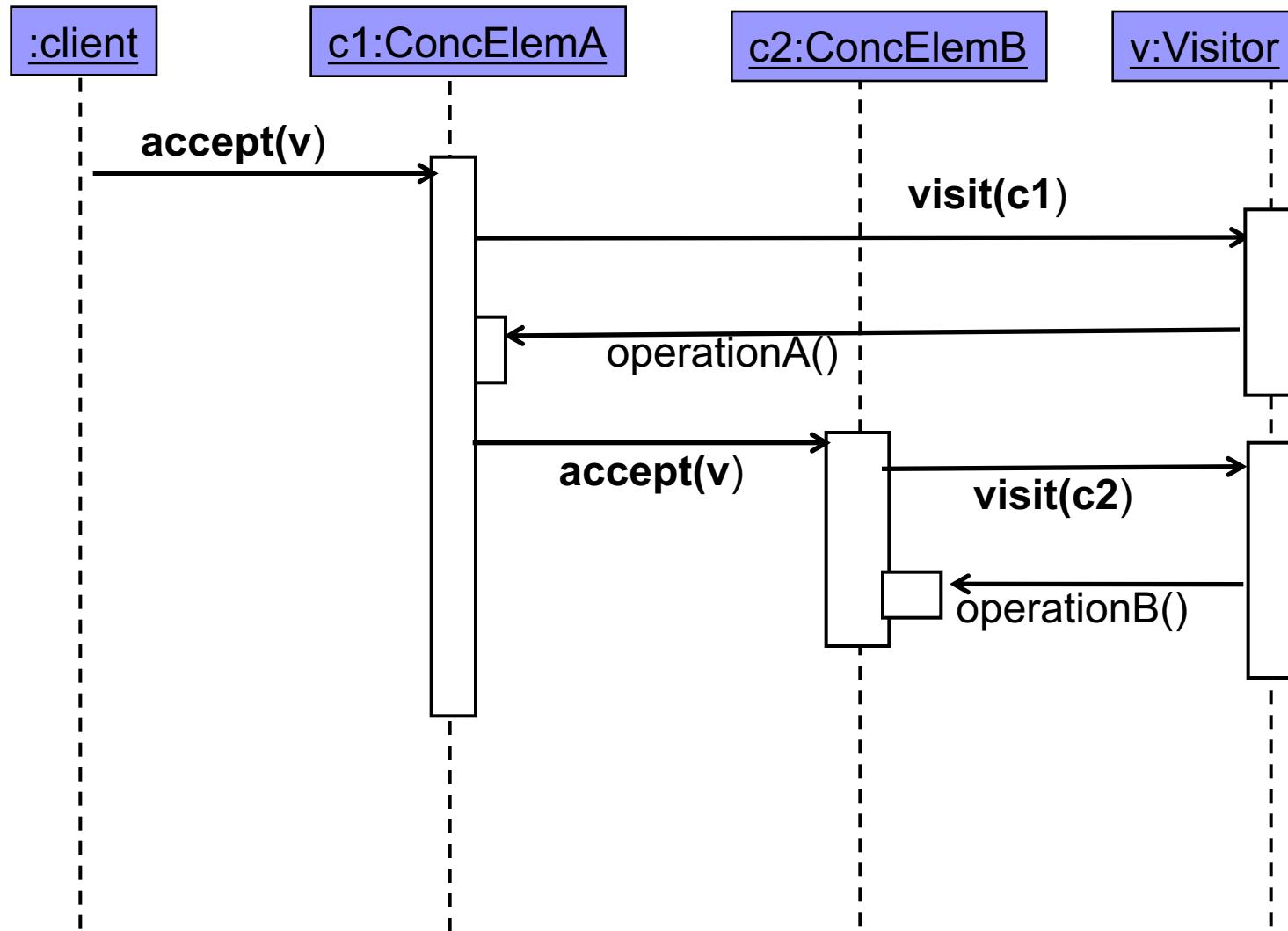
# Visitable Composite

---

```
public class Composite{...  
    public void accept(Visitor v){  
        v.visit(this); //get the info or do what you need  
        for(Component child : children)  
            child.accept(v);  
    }  
}  
  
void Composite::accept(Visitor& v){  
    for(Component* child : children)  
        child->accept(v);  
    v.visit(this);  
}
```

---

# Visitor- structure leads the traversal



# Exercise 2: make ToC

---

```
public class Document{ //...
    private List<Chapter> chapters;
    private List<Page> frontmatter;
    public void createToC(){
        Visitor v=new ToCVisitor();
        for(Chapter c:chapters) c.accept(v);
        frontmatter.add(v.getToC());
    }
    public void makeListofFigures(){
        Visitor v=new ListFigureVisitor();
        for(Chapter c:chapters) c.accept(v);
        frontmatter.add(v.getFigureList());
    }
}
```

After class, refactor and remove this repetition

# Exercise 2.1: make ToC

```
public interface Visitor{
    void visit(Chapter c);           Visitor can traverse
    void visit(Section s);           through hierarchies
    void visit(Paragraph p);
}

public class Chapter {           Chapter and Section do not
    private String title; //...
    private List<Section> parts; //Chapter has Sections
    public void accept(Visitor v){
        v.visit(this);
        for(var section: parts) section.accept(v);
    }
    public String getTitle(){ return title; }
}
```

# Exercise 2.2: make ToC

```
public interface Visitor{  
    void visit(Chapter c);  
    void visit(Section s);  
    void visit(Paragraph p);  
} // Let's use the Composite pattern  
public class Chapter extends CompositeNode{  
    private String title; //...  
    private List<Node> parts;//chapter has paragraphs and sections  
    public void accept(Visitor v){ //an operation for composite  
        v.visit(this);  
        for(var node: parts) node.accept(v);  
    }  
    public String getTitle(){ return title; }  
}
```

# Exercise 2: make ToC

---

```
public class ToCVisitor{  
    private Page page=new Page();    int ccnt=0; int scnt=0;  
    public void visit(Chapter c){ page.add(++ccnt,c.getTitle());}  
    public void visit(Section s){ page.add(ccnt,++scnt,  
s.getTitle());}  
    public void visit(Paragraph p){}  
    public Page getToC(){ return page;}  
} //no traversal logic in the visitor
```

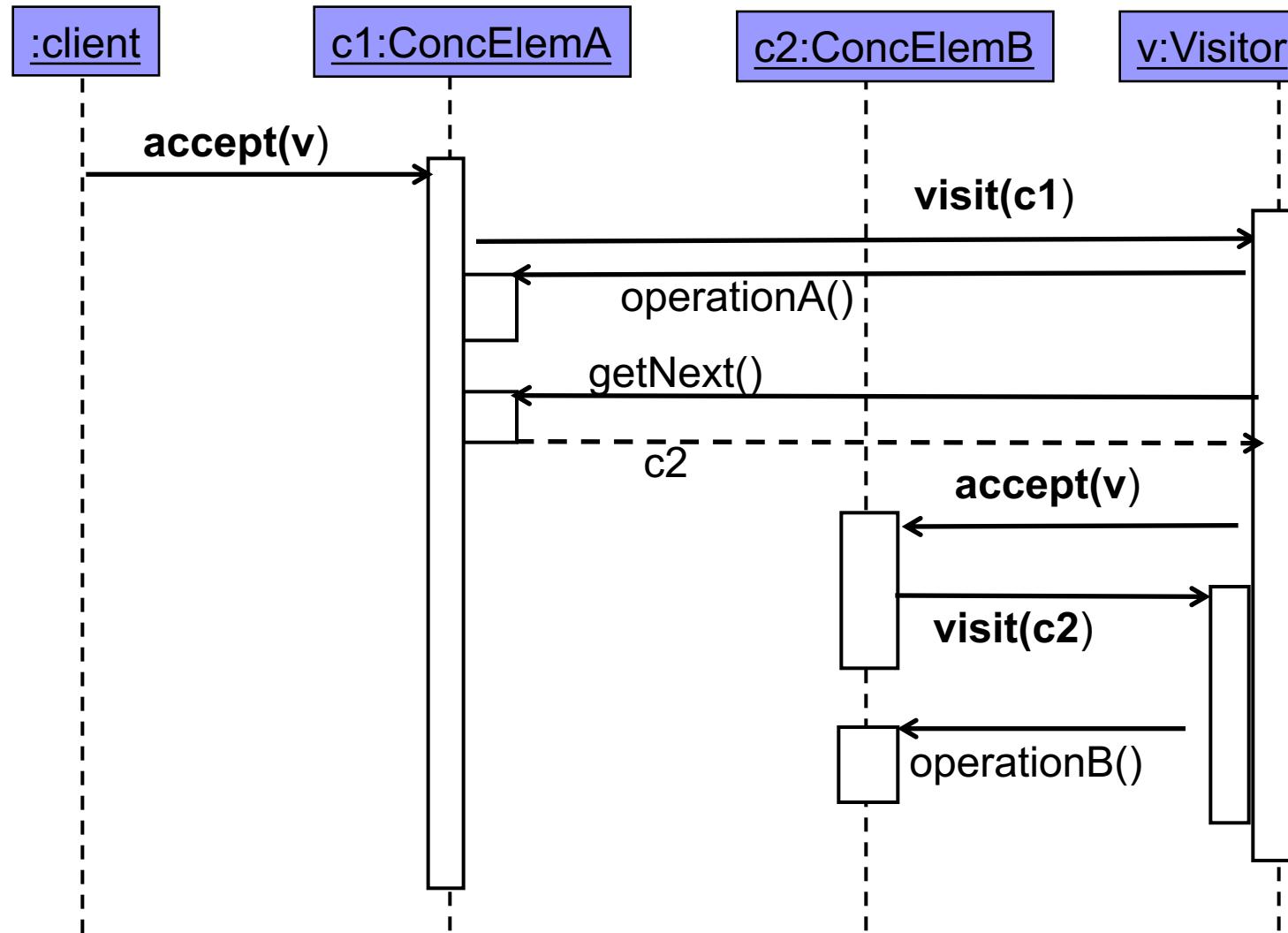
```
public class Chapter extends CompositeNode{  
    public void accept(Visitor v){ //an operation for composite  
        v.visit(this);  
        for(var node: parts) node.accept(v);  
    } //...
```

# Visitor -Traversals

---

1. Traversing nodes of a data structure using iterator
  - Chapter::accept(Visitor& v){v.visit(this);}
  - Visit methods collect data or perform operation
2. Traversing a composite pattern
  - Composite::accept(v) → all children accept(v) and v.visit(this)
  - Visit methods collect data or perform operation
3. Visitor decides what to visit next
  - ```
visit(Chapter* c){  
    accumulator=c->getTitle();//actual purpose  
    for(auto s:c->sections) s->accept(v);  
}
```
  - Chapter::accept(Visitor& v){v.visit(this);}

# Visitor - visitor leads the traversal



# Example 3: visitors for AST

---

## ■ Abstract Syntax Tree

- Print visitor
- Type checker visitor
- Visitors for several static analyses
- Code generating visitors

## ■ Visitor interface

- `void visit(ClassDeclarationNode node);`
- `void visit(MethodDeclarationNode node);`
- `void visit(VariableDeclarationNode node);`
- `void visit(AssignmentNode node);`
- `void visit(IfElseNode node);`
- `void visit(WhileNode node);`
- `// ...visit methods for other types of nodes.`

# Exercise 3: visitor of AST

---

## ■ Example: PrintVisitor

```
void PrintVisitor::visit(IfElseNode* node) {  
    cout<<"If: " + node->getCondition()<<endl;  
    node->getThen()->accept(this);  
    cout<<"Else:"<<endl;  
    node.getElse().accept(this);  
}  
  
class IfElseNode:public Node{  
private:  
    Node* condition; Node* thenStmt; Node* elseStmt;  
public: //constructors, getters, etc  
    void accept(Visitor& v){v.visit(this);}  
}; //no traversal logic in the Visitable
```

# When to Use Visitor

---

- When an object **structure** contains many classes of objects with **differing interfaces**, and you want to **perform operations** on these objects that depend on their concrete classes
- When many distinct and **unrelated operations** need to be performed on objects in an object **structure** and you want to avoid cluttering the classes with these operations
- When the classes defining the structure rarely change, but you often want to define **new operations** over the structure
  - If the object structure classes change often, then it's probably better to define the operations in those classes

# Exercise:

---

- Our app is used for human resources
- It reflects the organization hierarchy of the customer company.
- You are not permitted to change the code frequently. We are shipping first version soon.
- Customer want additional modules
  - Report vacation days left for the for selected departments or all departments
  - Report total budget left at each department including their subdivisions
- Note: if you were allowed to change, these reports may be an operation of the object structure

# Exercise 2:

---

- In the application there are a collection of Customers.
- Customers make orders and each order consists of items.
- We want to create a reporting module in our application to make statistics about a group of customers.
- The first statistics is which customer (no personal info) has made which purchases
  - Print orders and items into
  - Print how much the customer spent

# Visitor-Consequences -1

---

- Visitors makes adding new operations easier
  - If the structure involves many different classes, then adding a new operation to the structure requires changing all those classes
  - Coding a new Visitor does not affect the structure
- Visitors gathers related operations, separates unrelated ones
  - Related behavior is not spread over the classes defining the object structure; it's localized in a visitor.
  - Unrelated sets of behavior are partitioned in their own visitor subclasses.
  - A print visitor could have all the different ways to print

# Visitor-Consequences -2

---

## ■ Accumulating state

- A visitor can accumulate information as it traverses the object structure.
- Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.

## ■ Visiting across class hierarchies

- It can visit objects that do not have a common parent class

# Consequence

---

- Adding new `ConcreteElement` classes is hard
  - To add a new `ConcreteElement` we need to change all existing visitors
  - i.e. Add a new abstract operation on `Visitor` and a corresponding implementation in every `ConcreteVisitor` class.
- Breaking encapsulation
  - The `ConcreteElement` interface must be powerful enough to let visitors do their job.
  - We may be forced to provide public operations that access an element's internal state, which may compromise its encapsulation.
  - C++ friends are useful here

# Known uses

---

- `java.nio.file.FileVisitor` and `SimpleFileVisitor`
- `javax.faces.component.visit.VisitContext` and `VisitCallback`
- `javax.lang.model.element.AnnotationValue` and `AnnotationValueVisitor`
- `javax.lang.model.element.Element` and `ElementVisitor`
- `javax.lang.model.type.TypeMirror` and `TypeVisitor`

# Related Patterns

---

## ■ Visitors and Composites

- The visitor pattern can be used together with the composite pattern.
- The object structure would be a composite structure.
- Leveraging that an operation of the composite tells all the children do the operation
  - implementation of the accept method of the composite object invokes the accept methods of the component object

# Visitors and Iterators

---

- Both are used to traverse object structures.
- Iterator can help visitor, but visitor can manage its own traversal, like on a tree.
- Intents are different: encapsulating operation vs traversal
- Iterator only traverses; performs no operation itself.
  - type-blind: just gives you the next element (could be a base type, like Shape).
  - client "pulls" an element, then decides what to do.
- Visitor: Performs an operation across a structure of different concrete types.
  - type-aware: uses double-dispatch (e.g., visit(Circle), visit(Square)) to run the correct logic.
  - the operation is "pushed" to the element `element.accept(visitor)`