Visitor Pattern

Chapter Content

InterBit
Training & Consulting Ltd.

- Visitor Pattern Overview
- Example: composite text
- Possible visits
- Problem #1: node class
- Problem #2: iteration order
- Code Example

Motivation

InterBit
Training & Consulting Ltd.

- Collections are data types widely used in object oriented programming
- Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type

Motivation

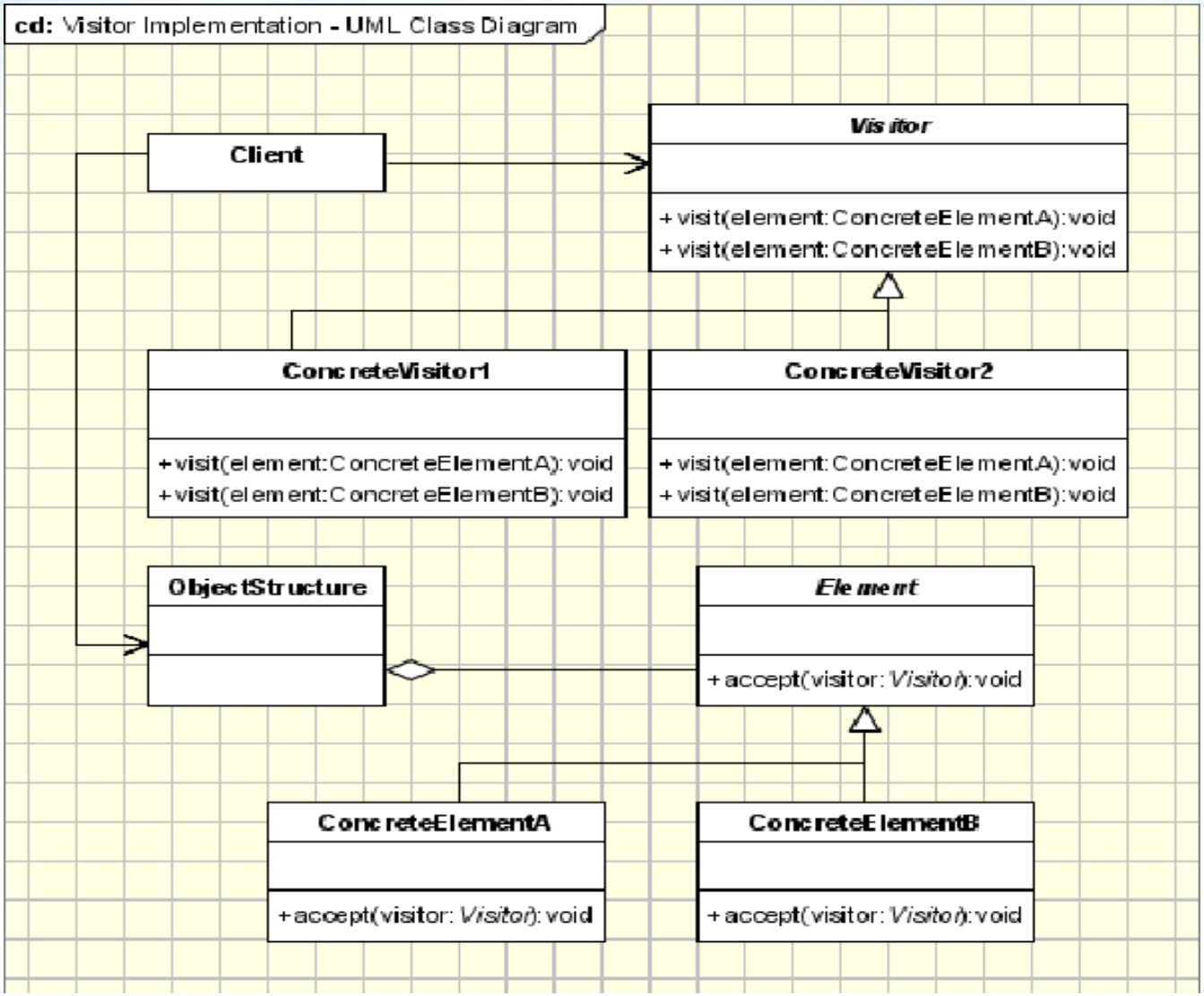
InterBit
Training & Consulting Ltd.

- A possible approach to apply a specific operation on objects of different types in a collection would be the use if blocks in conjunction with '**instanceof**' for each element
- This approach is not a nice one, not flexible and not object oriented at all
- Based on the Open/Close principle - we can replace 'if' blocks with an abstract class and each concrete class will implement its own operation

Intent

- Represents 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

Visitor UML Class Diagram



UML Diagram Explained

InterBit Training & Consulting Ltd.

- **Visitor** - This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes
  - Usually the name of the operation is the same and the operations are differentiated by the method signature: The input object type decides which of the method is called
- **ConcreteVisitor** - For each type of visitor all the visit methods, declared in abstract visitor, must be implemented.
  - Each Visitor will be responsible for different operations. When a new visitor is defined it has to be passed to the object structure.

UML Diagram Explained

InterBit Training & Consulting Ltd.

- **Visitable** - An abstraction declaring the 'accept' operation. The entry point enabling an object to be "visited" by the visitor object.
  - Each object from a collection should implement this abstraction in order to be able to be visited
- **ConcreteVisitable** - Classes implementing the *Visitable* interface
  - The visitor object is passed to this object using the 'accept' operation
- **ObjectStructure** - Contains all the objects that can be visited. Offers a mechanism to iterate through all the elements
  - This structure is not necessarily a collection. Can be a complex structure, such as a composite object

Applicability

InterBit Training & Consulting Ltd.

The visitor pattern is used when:

- Similar operations have to be performed on objects of different types grouped in a structure (a collection or a more complex structure).
- There are many distinct and unrelated operations needed to be performed. Visitor pattern allows us to create a separate visitor concrete class for each type of operation and to separate this operation implementation from the objects structure.
- The object structure is not likely to be changed but is very probable to have new operations which have to be added. Since the pattern separates the visitor (representing operations, algorithms, behaviors) from the object structure it's very easy to add new visitors as long as the structure remains unchanged.

Example

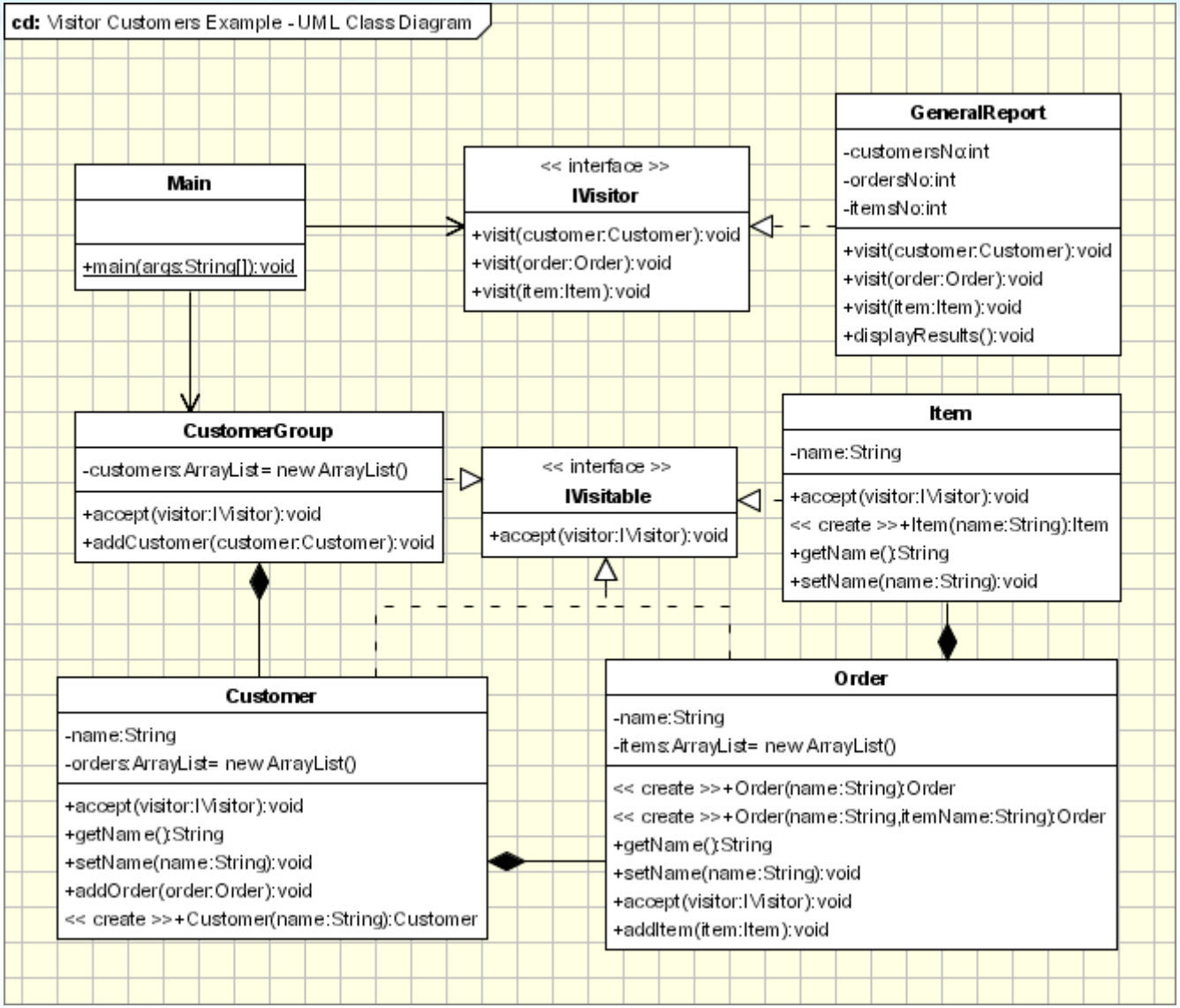
InterBit Training & Consulting Ltd.

Example 1 - Customers Application

- We want to create a reporting module in our application to get statistics about a group of customers
- The statistics should made very detailed so all the data related to the customer must be parsed
- All the entities involved in this hierarchy must accept a visitor so the **CustomerGroup**, **Customer**, **Order** and **Item** are visitable objects:
- ***IVisitor*** and ***IVisitable*** interfaces
- **CustomerGroup**, **Customer**, **Order** and **Item** are all visitable classes. A CustomerGroup represents a group of customers, each Customer can have one or more orders and each order can have one ore more Items
- **GeneralReport** is a visitor class and implements the *IVisitor* interface.

Example UML Diagram

InterBit
Training & Consulting Ltd.



Example – Source Code

InterBit Training & Consulting Ltd.

- IVisitor and IVisitable interfaces

```
public interface IVisitable {
    public void accept(IVisitor visitor);
}

public interface IVisitor {
    public void visit(Customer customer);
    public void visit(Order order);
    public void visit(Item item);
}
```

Example – Source Code

InterBit Training & Consulting Ltd.

- **GeneralReport** is a visitor that implements the *IVisitor* interface.

```
public class GeneralReport implements IVisitor{
	private int customersNo,ordersNo,itemsNo;
	public void visit(Customer customer) {
		System.out.println(customer.getName());
		customersNo ++;
	}
	public void visit(Order order) {
		System.out.println(order.getName());
		ordersNo++;
	}
	public void visit(Item item){
		System.out.println(item.getName());
		itemsNo++;
	}
	public void displayResults() {
		System.out.println("Nr of customers:" + customersNo);
		System.out.println("Nr of orders:   " + ordersNo);
		System.out.println("Nr of itemss:   " + itemsNo);
	}
}
```

Example – Source Code

InterBit Training & Consulting Ltd.

- CustomerGroup, Customer, Order and Item are all visitable classes. A CustomerGroup represents a group of customers, each Customer can have one or more orders and each order can have one ore more Items

```
import java.util.ArrayList;
import java.util.Iterator;

public class CustomerGroup {

	private ArrayList customers = new ArrayList();

	public void accept(IVisitor visitor) {
		for (Iterator it=customers.iterator(); it.hasNext();) {
			((Customer)it.next()).accept(visitor);
		}
	}

	public void addCustomer(Customer customer) {
		customers.add(customer);
	}
}
```

Example – Source Code

InterBit Training & Consulting Ltd.

- The Customer Class

```
import java.util.ArrayList;
import java.util.Iterator;

public class Customer implements IVisitable{
      private String name;
      private ArrayList orders = new ArrayList();
      public void accept(IVisitor visitor) {
            visitor.visit(this);
            for (Iterator it=orders.iterator(); it.hasNext();)
                  ((IVisitable)it.next()).accept(visitor);
      }
     // getters and setters omitted for brevity
      public void addOrder(Order order) {
            orders.add(order);
      }
      public Customer(String name){
            this.name = name;
      }
}
```

Example – Source Code

InterBit Training & Consulting Ltd.

- The Order Class

```
public class Order implements IVisitable {
      private String name;
      private ArrayList items = new ArrayList();
      public Order(String name) {
            this.name = name;
      }
      public Order(String name, String itemName) {
            this.name = name;
            this.addItem(new Item(itemName));
      }
      public void accept(IVisitor visitor) {
            visitor.visit(this);
            for (Iterator it=items.iterator(); it.hasNext();)
                  ((Item)it.next()).accept(visitor);
      }
      public void addItem(Item item) {
            items.add(item);
      }
}
```

Example – Source Code

InterBit Training & Consulting Ltd.

- The Item Class

```
public class Item implements IVisitable{
     private String name;
    // getters and setters omitted for brevity
     public void accept(IVisitor visitor) {
          visitor.visit(this);
     }
     public Item(String name){
          this.name = name;
     }
}
```

Example – Test Source Code

InterBit Training & Consulting Ltd.

```
public class Main {
      public static void main(String[] args) {
            Customer c = new Customer("customer1");
            c.addOrder(new Order(".order1", "..item1"));
            c.addOrder(new Order(".order2", "..item1"));
            c.addOrder(new Order(".order3", "..item1"));
            Customer c2 = new Customer("customer2");
            Order o = new Order(".order_a");
            o.addItem(new Item("..item_a1"));
            o.addItem(new Item("..item_a2"));
            o.addItem(new Item("..item_a3"));
            c2.addOrder(o);
            c2.addOrder(new Order(".order_b", "..item_b1"));
            CustomerGroup customers = new CustomerGroup();
            customers.addCustomer(c);
            customers.addCustomer(c2);
            GeneralReport visitor = new GeneralReport();
            customers.accept(visitor);
            visitor.displayResults();
      }
}
```

problems and implementation

InterBit Training & Consulting Ltd.

Tight Coupled Visitable objects

- The classic implementation of the Visitor pattern have a major drawback because the type of visitor methods has to be known in advance.
- The Visitor interface can be defined using polymorphic methods or methods with different names:

```
public interface IVisitor {
    public void visit(Customer customer);
     public void visit(Order order);
     public void visit(Item item);
}
public interface IVisitor {
     public void visitCustomer(Customer customer);
     public void visitOrder(Order order);
     public void visitItem(Item item);
}
```

problems and implementation

InterBit Training & Consulting Ltd.

Tight Coupled Visitable objects (continued)

- However this type should be known in advance
- When a new type is added to the structure a new method should be added to this interface and all existing visitors have to be changed accordingly
- Then, A pair method is written in the concrete Visitable objects:

```
public class Customer implements IVisitable{
     public void accept(IVisitor visitor) {
          visitor.visit(this);
     }
}
```

problems and implementation

InterBit Training & Consulting Ltd.

Tight Coupled Visitable objects (continued)

- It doesn't really matters if the polymorphic methods with the same name but different signatures are used or not
- Either way the type is known at compile time so for each new visitable object this method must be implemented accordingly
- The main advantage of the fact that new visitors can be easily added is tainted by the fact that the addition of new visitable objects is really hard.

Problems and implementation

InterBit Training & Consulting Ltd.

Solution: Visitor Pattern using Reflection

- Reflection can be used to overcome the main drawback of the visitor pattern
- When the standard implementation of visitor pattern is used the method to invoke is determined at runtime
- Reflection is the mechanism used to determine the method to be called at compile-time
- This way the visitable object will use the same code in the accept method
- This code can be moved in an abstraction so the *IVisitable* interface will be transformed to an advanced class.

Problems and implementation

InterBit Training & Consulting Ltd.

Solution: Visitor Pattern using Reflection

- Let's take our example: We need to add a new visitable class in our structure, called **Product**.
- We should modify the *IVisitor* interface to add a visitProduct() method
  - But changing an interface is one of the worst things we can do.
  - Usually, interfaces are extended by lots of classes changing the interface means changing the classes
  - Maybe we have lots of visitors but we don't want to change all of them, we only need another report

Problems and implementation

InterBit Training & Consulting Ltd.

Solution: Visitor Pattern using Reflection

- In this case we start from the idea that we should keep the interface unchanged
- The solution is to replace the interface with an abstract class and to add an abstract method called *defaultVisit()*
  - The *defaultVisit()* will be implemented by each new concrete visitor, but the interface and old concrete visitors will remain unchanged.

Problems and implementation

InterBit Training & Consulting Ltd.

- The visit(Object object) method check if there is visit method for the specific object. If there is not an available visit the call is delegated to the *defaultVisit()* method:

```
public abstract class Visitor implements IVisitor {
      public abstract void visit(Customer customer);
      public abstract void visit(Order order);
      public abstract void visit(Item item);
      public abstract void defaultVisit(Object object);
      public void visit(Object obj) {
          try {
              Method downPolymorphic = obj.getClass().getMethod("visit",new Class[]{obj.getClass()});
               if (downPolymorphic == null)
                   defaultVisit(object);
               else
                   downPolymorphic.invoke(this, new Object[] {object});
            } catch (Exception e) {
                 this.defaultVisit(object);
            }
      }
}
```

Problems and implementation

InterBit Training & Consulting Ltd.

- The new Product class is now compatible and it's accept method now casts to a concrete visitor heirarchy:

```
public class Product implements IVisitable {
    private double price;
    public Product(double price) {
        this.price = price;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void accept(IVisitor visitor) {
        ((Visitor) visitor).visit(this);
    }
}
```

Visitor Pattern Summary

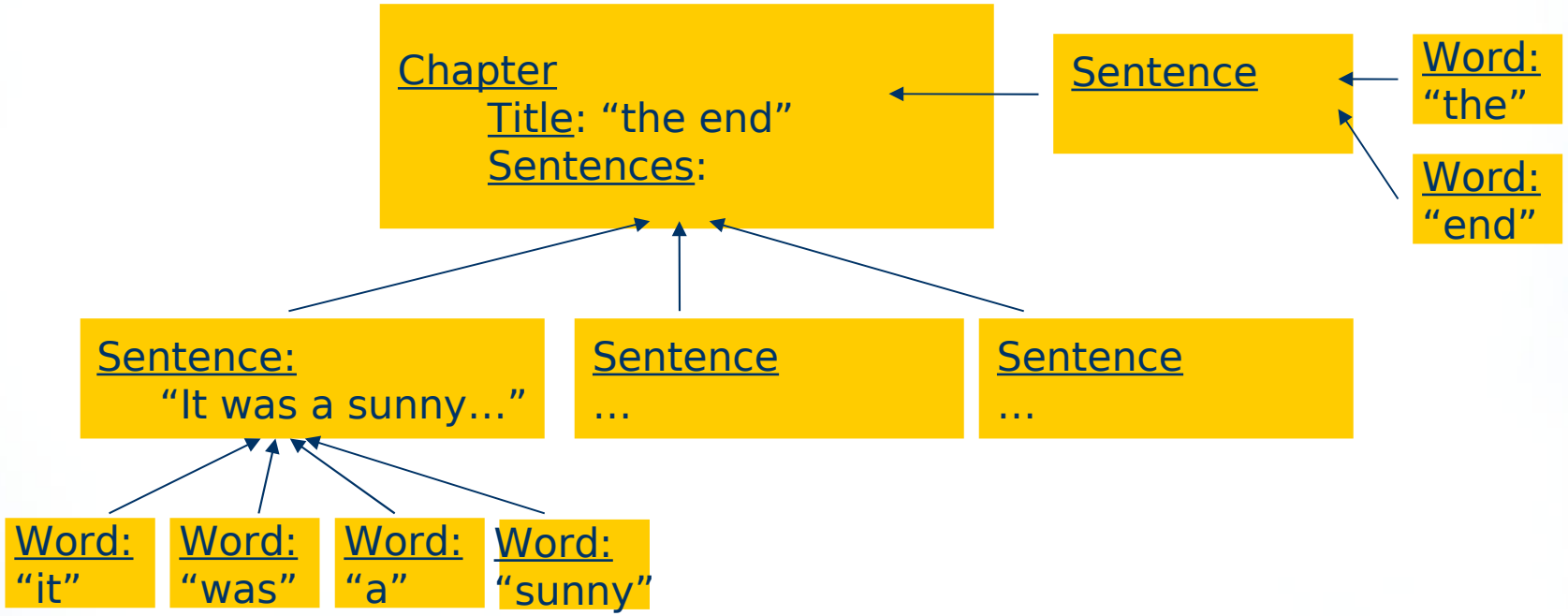
InterBit Training & Consulting Ltd.

- The visitor pattern is a great way to provide a flexible design for adding new visitors to extend existing functionality without changing existing code
- The Visitor pattern comes with a drawback: If a new visitable object is added to the framework structure all the implemented visitors need to be modified
- The separation of visitors and visitable is only in one sense: visitors depend of visitable objects while visitable are not dependent of visitors
- Part of the dependency problems can be solved by using reflection with a performance cost.

Example: composite text

InterBit Training & Consulting Ltd.

Classes to be visited:



Visited Class Overview (cont.)

InterBit Training & Consulting Ltd.

```
abstract class Node { ...
}
class Chapter extends Node {
    private Sentence title;
    private ArrayList sentences;
    public Iterator sentenceIterator(){ return sentences.iterator(); }
}
class Sentence extends Node{
     private ArrayList words;
     public Iterator wordIterator(){ return words.iterator(); }
}
class Word extends Node {
     private String str;
}
// TODO: setters, getters, constructors ...
```

Possible visits (functionality)

InterBit Training & Consulting Ltd.

- Spell check.
- Count words.
- Count sentences.
- …

- Instead of adding methods to Chapter/ Sentence/Word, we'd rather define such operations in an external visitor.

Problem #1: Node class

InterBit Training & Consulting Ltd.

- Who is responsible for identifying the concrete type of a Node (Chapter/Word..).
- This problem arises due to lack of multi-polymorphism, combined with the fact that operations now move outside the class.

```
public static void f(Chapter chap)  { System.out.println("CHAPTER");}
public static void f(Word word)     { System.out.println("WORD");}
Node n = (Math.random() > 0.5 ? new Chapter() : new Word());

f(n);   // ERROR !
```

Problem #1: Node class (cont.)

InterBit Training & Consulting Ltd.

- Solution (A): visitor uses *instanceof*

```
class SpellCheckVisitor {
    public void visit(Node node) {
     if (node instanceof Chapter)
         ...   // check title, and all sentences in chapter
     else if (node instanceof Word)
         ...   // check word according to dictionary
    }
}
```

Not GoF classical recommendation, but may be required when it's too late to change the visited class (e.g: XML Dom object).

Problem #1: Node class (cont.)

InterBit Training & Consulting Ltd.

- Solution (B) – the classical pattern:
  - Visitor will have a different operator implementation for each Node type:

```
abstract class Visitor {
    public abstract void visitChapter(Chapter chap);
    public abstract void visitSentence(Sentence sent);
    public abstract void visitWord(Word word);
}
```

Visited object has an *accept(Visitor)* method which tells the visitor which type to use:

Problem #1: Node class (cont.)

InterBit Training & Consulting Ltd.

```
abstract class Node {
     public abstract void accept(Visitor v);
}
class Chapter extends Node{
    ...
    public void accept(Visitor v) { v.visitChapter(this); }
}
class Sentence extends Node{
    ...
    public void accept(Visitor v) { v.visitSentence(this);}
}
class Word extends Node {
    ...
    public void accept(Visitor v) { v.visitWord(this); }
}
```

Problem #2: iteration order

InterBit Training & Consulting Ltd.

- Who decides how/when to iterate on complex visited objects?
  - Will all visitors be using the same order of iteration?

Problem #2: iteration order (cont.)

- Solution (A): Visited object holds all iteration responsibility:

```
class Sentence extends Node{
    ...
    public void accept(Visitor v) {
        v.visitSentence(this);
        for(Iterator iter = wordIterator(); iter.hasNext();)
            v.visitWord((Word)iter.next());
    }
}
```

Visited class (e.g. Sentence) tells visitor how to progress

Saves code duplication for similar visitors, but is not flexible (forcing uniform traversal).

Problem #2: iteration order (cont.)

- Solution (B): visitor is responsible for iteration, possibly aided by iterators.

```
class SpellCheckVisitor {
    public void visitSentence(Sentence sent) {
        for(Iterator iter=sent.wordIterator(); iter.hasNext();)
            v.visitWord((Word)iter.next());
    }
}
```

Visitor decides how to progress

- Now different algorithms may use different traversals: in-order, post-order,backwards…
- Visitor may decide to skip some components.

Code Example

InterBit Training & Consulting Ltd.

- For the following code example, we've made the following choices:
  - Type recognition:
    visited class helps visitor know which class is visited (avoiding *instanceof*).
  - Iteration responsibility:
    visitor decides on iteration
    (relying on iterators provided by the visited classes).

Code Example (cont.)

```
// Base class for chapter, sentence, word...
abstract class Node {
    public abstract void accept(Visitor v);
}

class Chapter extends Node{
    private Sentence title;
    private ArrayList sentences;
    public void accept(Visitor v){
        v.visitChapter(this);
    }
    public Iterator sentenceIterator(){
     return sentences.iterator();
    }
    ... // constructors, setters, getters
}
```

The visited classes:

Chapter, Sentence, Word…

Code Example (cont.)

InterBit Training & Consulting Ltd.

```
class Sentence extends Node{
     private ArrayList words;
     public void accept(Visitor v){
          v.visitSentence(this);
     }
     public Iterator wordIterator(){
          return words.iterator();
     }
}
class Word extends Node {
     private String str;
     public void accept(Visitor v){
          v.visitWord(this);
     }
}
... // consturctors, setters, getters
```

Code Example (cont.)

InterBit Training & Consulting Ltd.

```
abstract class Visitor {
    public abstract void visitChapter(Chapter chap);
    public abstract void visitSentence(Sentence sent);
    public abstract void visitWord(Word word);
}
class SpellCheckVisitor extends Visitor {
    public void visitChapter(Chapter chap){
        visitSentence(chap.getTitle());
        for(Iterator it= chap.sentenceIterator(); it.hasNext(); )
            visitSentence((Sentence)it.next());
    }
    public void visitSentence(Sentence sent){
         for(Iterator it= sent.wordIterator(); it.hasNext(); )
             visitWord((Word)it.next());
    }
    public void visitWord(Word word){
        ...  // Check word against dictionary DB. If misspelled, log it
    }
```

General interface for all visitors

Concrete visitor for spell-checking