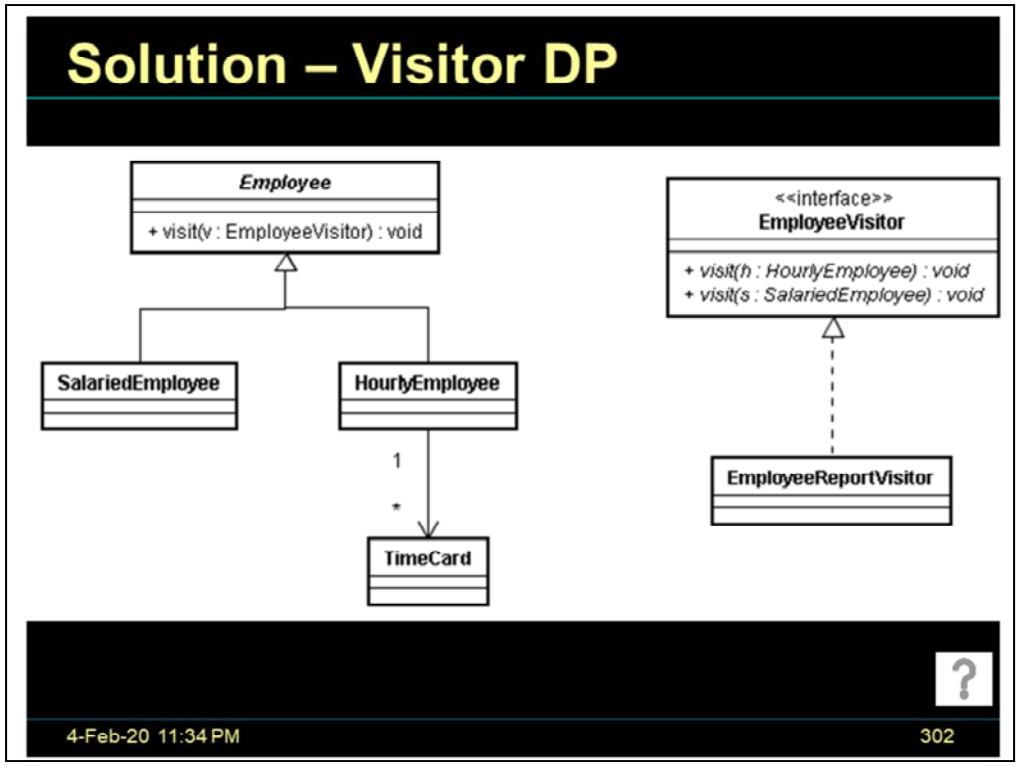

Visitor DP
4-Feb-20 11:34 PM
300

Assignment

- We have an Employee base class that holds the name of the employee.
  - We also have two derivatives named HourlyEmployee and SalariedEmployee.
  - HourlyEmployee holds a list of TimeCard objects, whereas SalariedEmployee holds just the salary.
- Now let's say we want to generate a report, one line per employee.
  - For every hourly employee we want that line to read 'Hourly Bob worked 8 hours.', whereas for every salaried employee we'd like it to read 'Salaried Bill earns $350.'



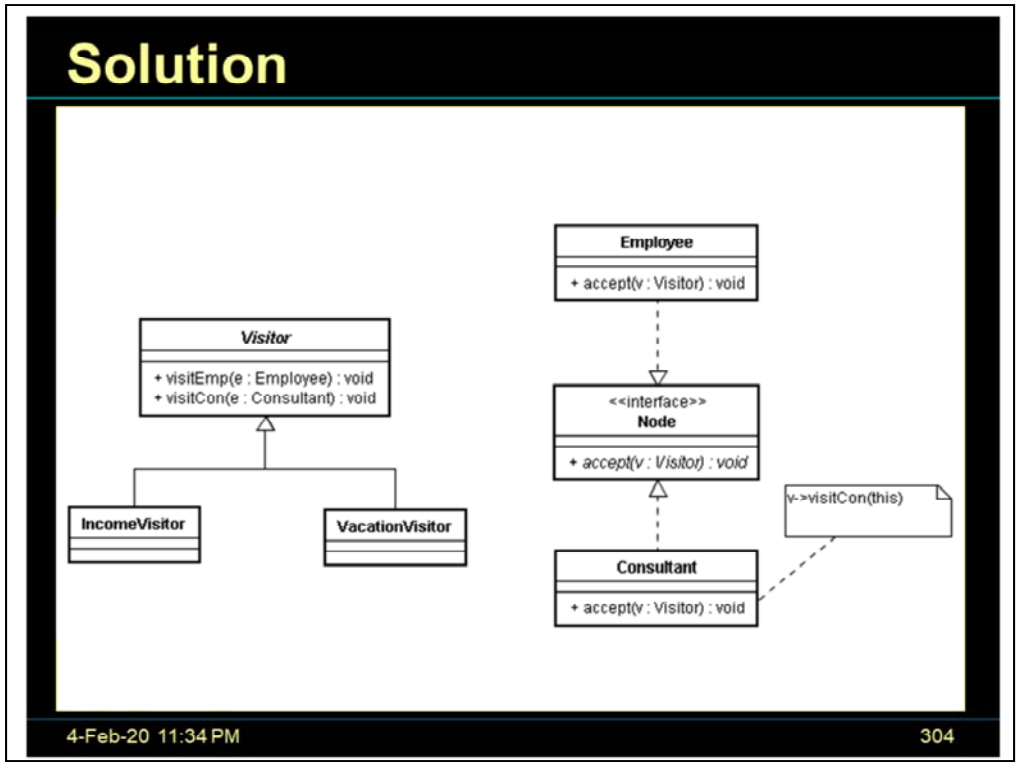
Code examples present.

```
public abstract class Employee { protected String name;
                    public Employee(String name) {this.name = name;}
                    public abstract void accept(EmployeeVisitor v);
                    public String getName() {return name;}
}
public class SalariedEmployee extends Employee {private int salary;
                    public SalariedEmployee(String name, int salary) {          super(name); this.salary = salary;}
                    public void accept(EmployeeVisitor v) {v.visit(this);}
                    public int getSalary() {return salary;}
}
public class HourlyEmployee extends Employee { private List<TimeCard> timeCards;
                    public HourlyEmployee(String name) {super(name);timeCards = new ArrayList<TimeCard>();}
                    public void addTimeCard(TimeCard timeCard) {timeCards.add(timeCard);}
                    public void accept(EmployeeVisitor v) {v.visit(this);}
                    public int getHours() {int hours = 0;for (TimeCard tc : timeCards)hours += tc.getHours();return hours;}
}
public class TimeCard {private int hours;
                    public TimeCard(int hours) {this.hours = hours;}
                    public int getHours() {return hours;}
}
public interface EmployeeVisitor { void visit(HourlyEmployee hourlyEmployee);
                    void visit(SalariedEmployee salariedEmployee);
}
public class EmployeeReportVisitor implements EmployeeVisitor {
                    private String reportLine;
                    public String getReportLine() {
                    return reportLine;
}
public void visit(HourlyEmployee hourlyEmployee) {
```

```
                int hours = hourlyEmployee.getHours(); String name = hourlyEmployee.getName();
                reportLine = String.format("Hourly %s worked %d hours.", name, hours);
}
public void visit(SalariedEmployee salariedEmployee) {
                String name = salariedEmployee.getName();
                int salary = salariedEmployee.getSalary();
                reportLine = String.format("Salaried %s earns $%d.", name, salary);
}
}
```

Problem

- Give a generalized solution:
  - We have an Employee class and Consultant class. We may store employees in other forms also later on.
  - We want to increment the income of everyone. Also sometimes, we want to calculate the vacation details of everyone.
    - The manner in which the increment and vacation information are calculated are very complex algorithm. We may need to do other operations on employees also.



Represents an operation to be performed on different classes

Visitor lets us define a new operation without changing the classes on which it operates.

Adding new operations is easy.

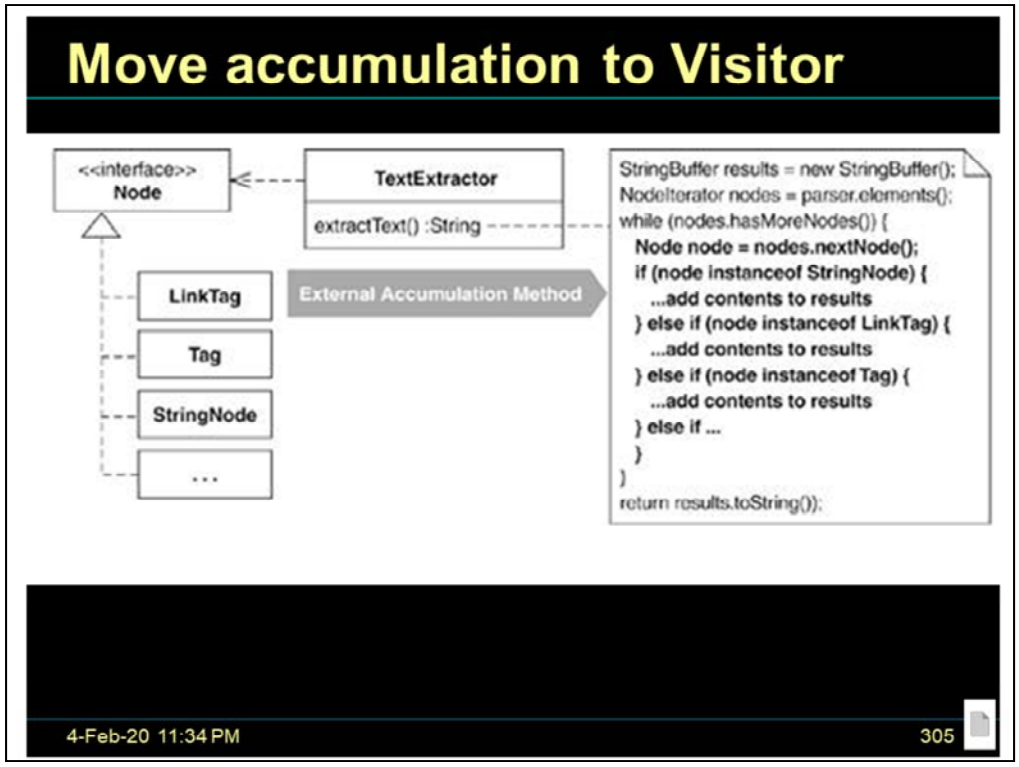
The Visitor pattern is like a more powerful Command pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.

==========

We want to perform many distinct and unrelated operations on different classes, but we do now want to pollute the classes with these operations.

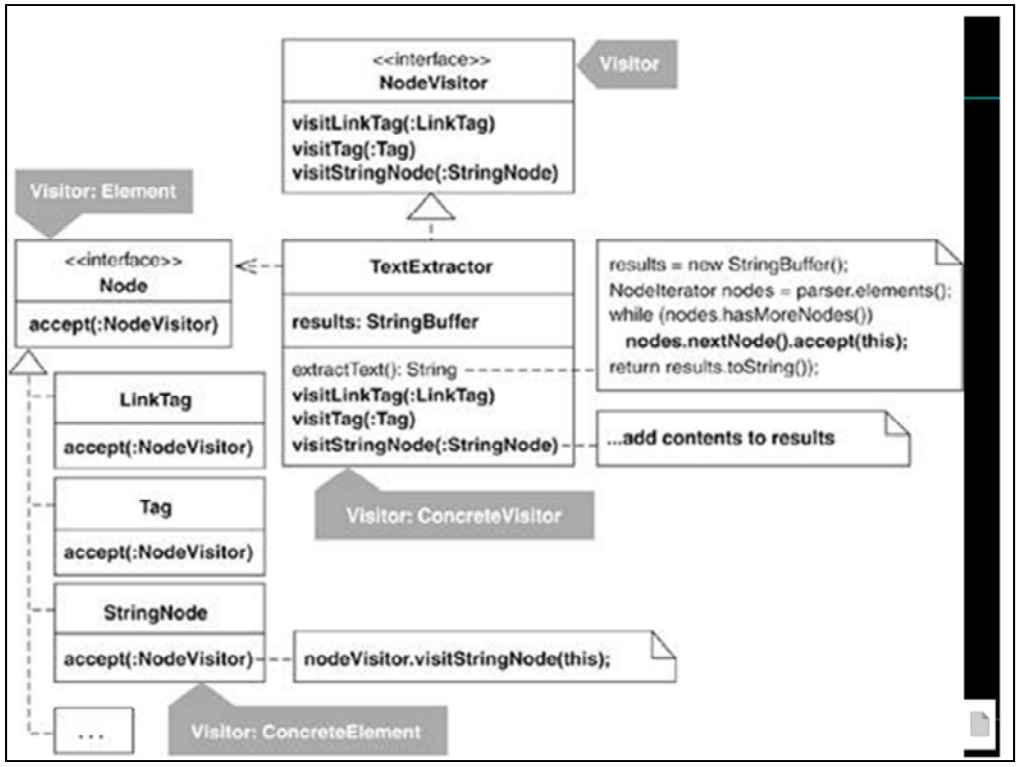
The existing classes rarely change. But the operations on them keep changing. So it makes sense to separate these operations.

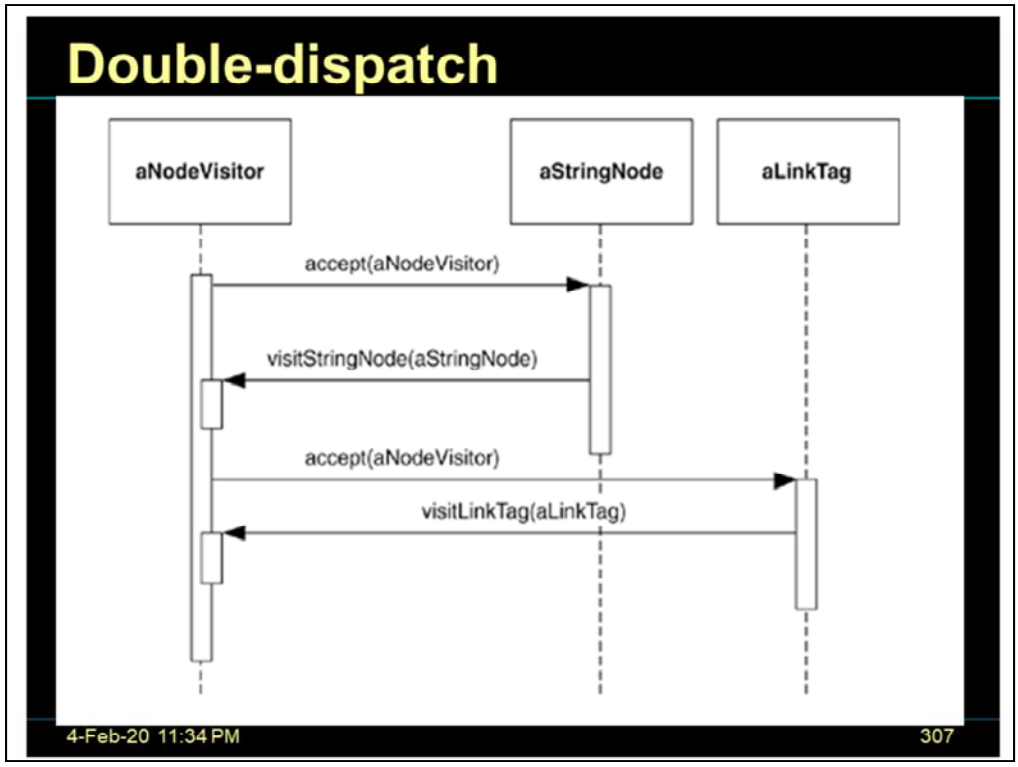
This pattern is generally used during maintenance of exiting classes, when we don't want to change the exiting classes much. It's use is unlikely during new development.



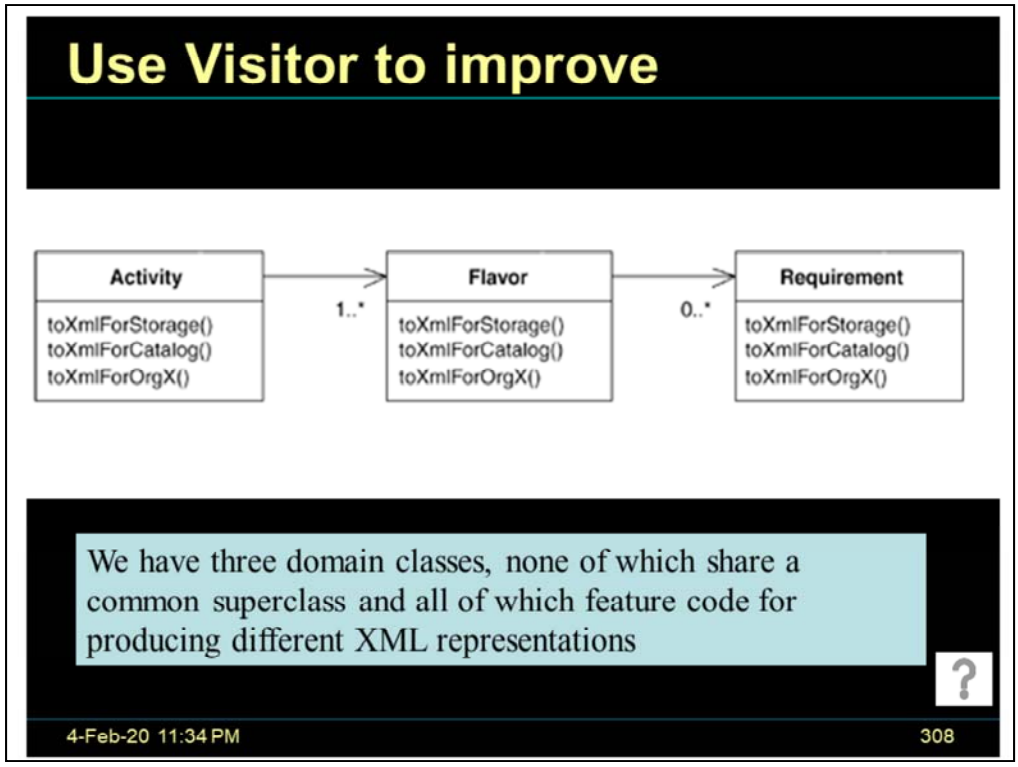
A method accumulates information from heterogeneous classes.

Move the accumulation task to a Visitor that can visit each class to accumulate the information.

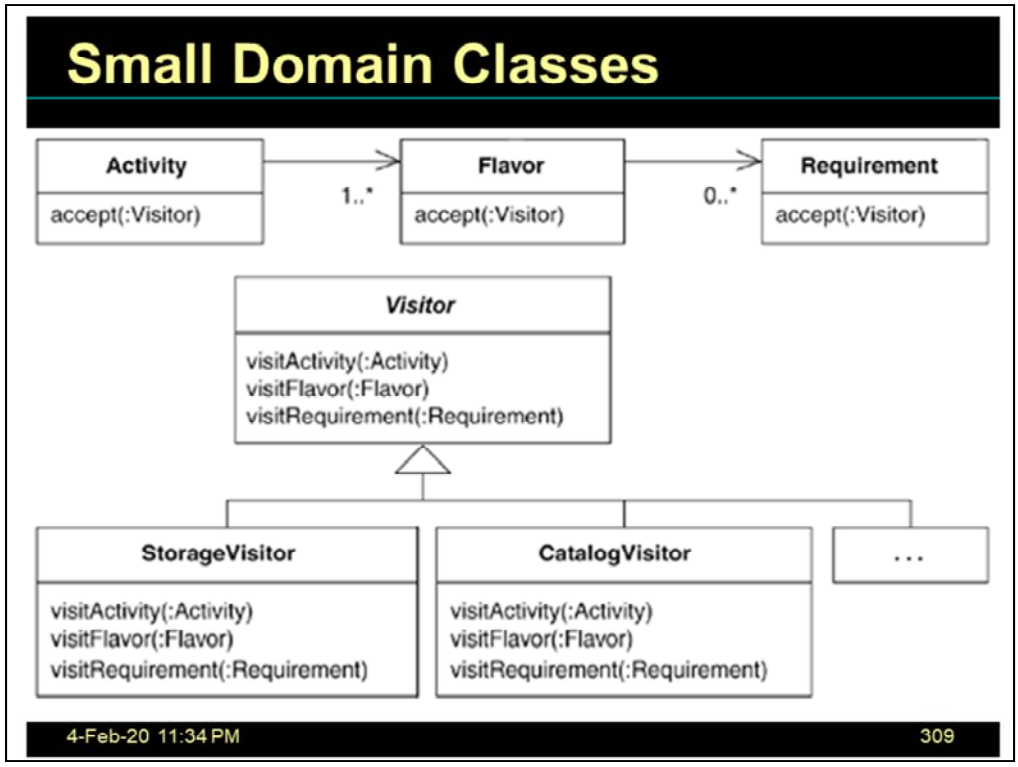

<<interface>>
NodeVisitor
visitLinkTag(:LinkTag)
visitTag(:Tag)
visitStringNode(:StringNode)
Visitor
Visitor: Element
<<interface>>
Node
accept(:NodeVisitor)
TextExtractor
results: StringBuffer
extractText(): String
visitLinkTag(:LinkTag)
visitTag(:Tag)
visitStringNode(:StringNode)
results = new StringBuffer();
NodeIterator nodes = parser.elements();
while (nodes.hasMoreNodes())
nodes.nextNode().accept(this);
return results.toString());
...add contents to results
Visitor: ConcreteVisitor
LinkTag
accept(:NodeVisitor)
Tag
accept(:NodeVisitor)
StringNode
accept(:NodeVisitor)
nodeVisitor.visitStringNode(this);
...
Visitor: ConcreteElement



A Visitor is a class that performs an operation on an object structure. The classes that a Visitor visits are heterogeneous, which means they hold unique information and provide a specific interface to that information. Visitors can easily interact with heterogeneous classes by means of double-dispatch. This means that each of a set of classes accepts a Visitor instance as a parameter (via an "accept" method: accept(Visitor visitor)) and then calls back on the Visitor, passing itself to its corresponding visit method, as shown in the following diagram.



What's wrong with this design? The main problem is that you have to add a new toXml method to each of these domain classes every time you have a new XML representation. In addition, the toXml methods bloat the domain classes with representation code, which is better kept separate from the domain logic, particularly when you have a lot of it. In the Mechanics section, I refer to the toXml methods as internal accumulation methods because they are internal to the classes used in the accumulation.



With this new design, the domain classes may be represented using whatever Visitor is appropriate. Furthermore, the copious representation logic that once crowded the domain classes is now encapsulated in the appropriate Visitor.

Benefits and Liabilities

+ Accommodates numerous algorithms for the same heterogeneous object structure.

+Visits classes in the same or different hierarchies.

+ Calls type-specific methods on heterogeneous classes without type-casting.

–Complicates a design when a common interface can make heterogeneous classes homogeneous.

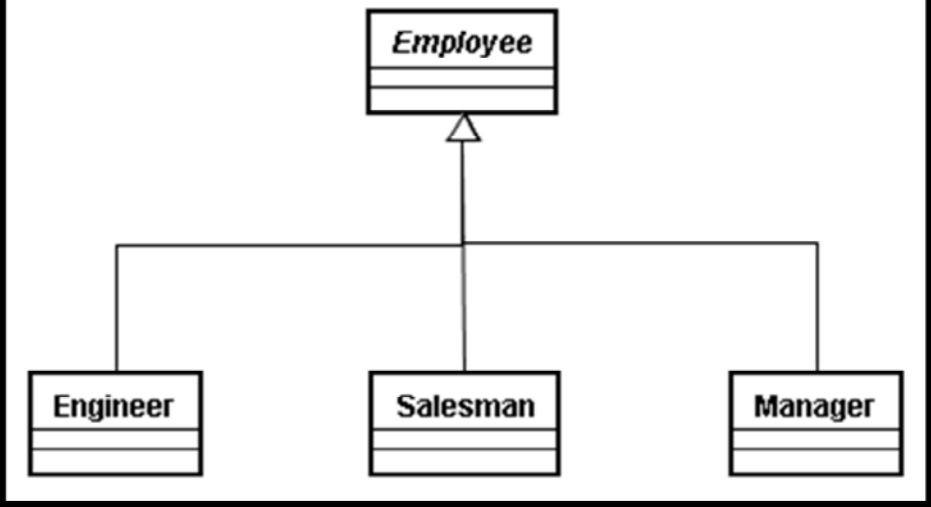
– A new visitable class requires a new accept method along with a new visit method on each Visitor.

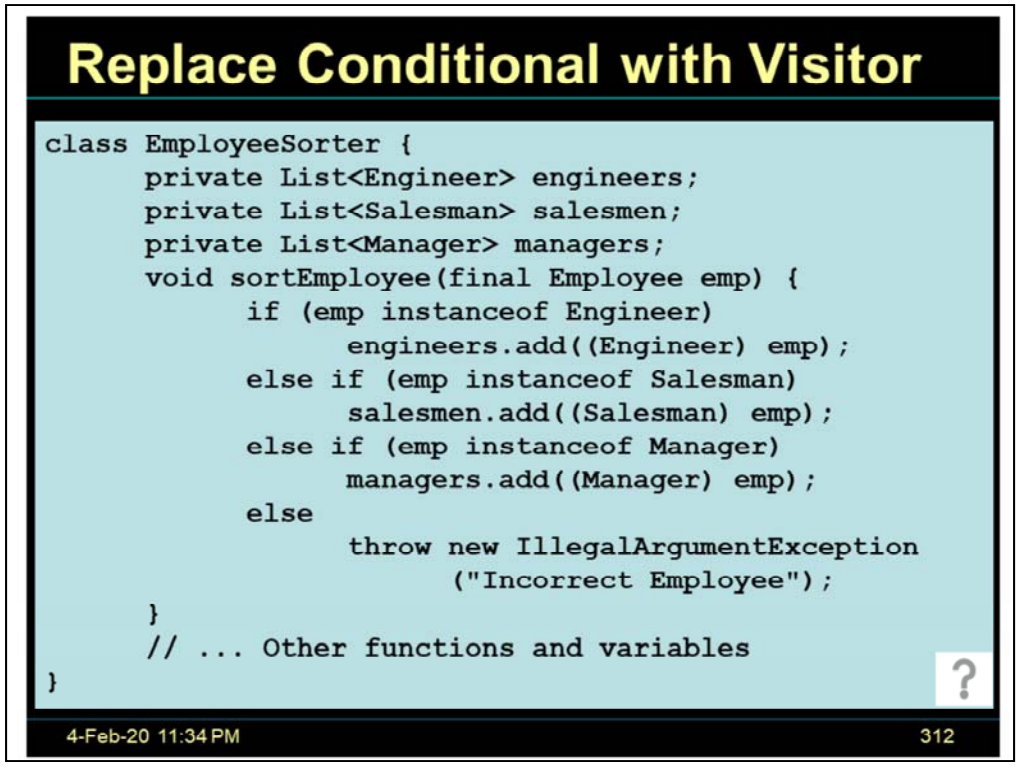
– May break encapsulation of visited classes.

Assignment

- In a rich GUI editor, we want a spell-checker and an hyphenator.
  - These functions are separate and hence must not be stored along with others.

For this Hierarchy…





Problem in DesignPatternsParticipants.visitor package for java, DesignPartternsParticipants.visitor namespace for C#, EmployeeSorter.cpp in C++

Solution not present yet.