Software Design Patterns

Lecture 13

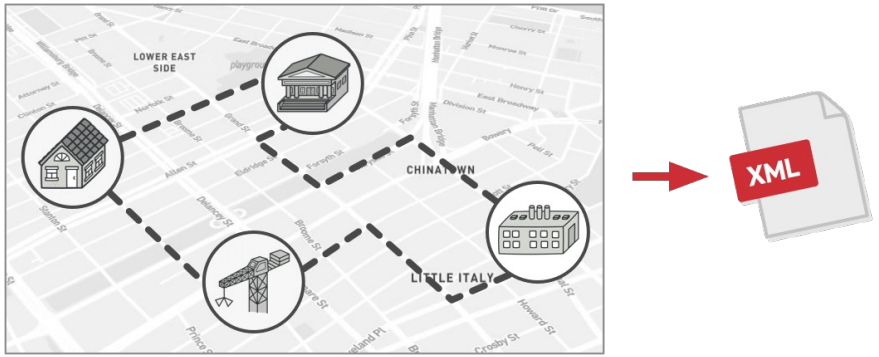
Visitor

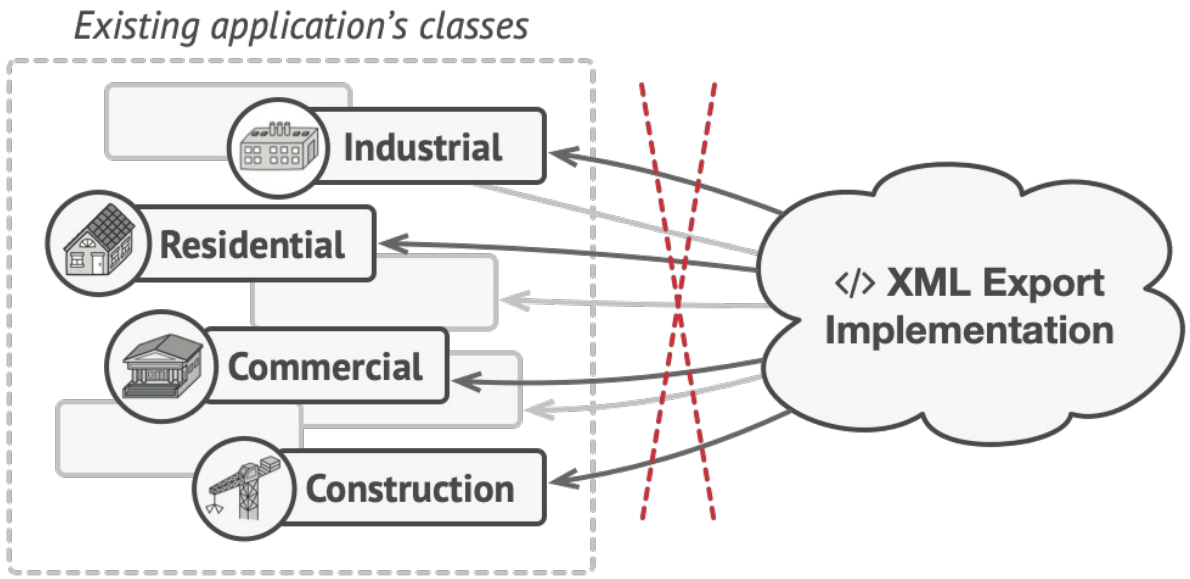
Dr. Fan Hongfei

28 November 2024

Visitor: Problem

- **Example: exporting the graph into XML format**





[Figures in this slide are extracted from https://refactoring.guru/design-patterns/visitor]

Visitor: Solution

- Place the new behavior into a separate class called **visitor**
- The original object passed to the visitor's method as an argument

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```

- To call the methods

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

- Question: why not method overloading?
  - The exact class is unknown in advance
- Visitor pattern utilizes **double dispatch**
  - Delegating the choice to the original objects

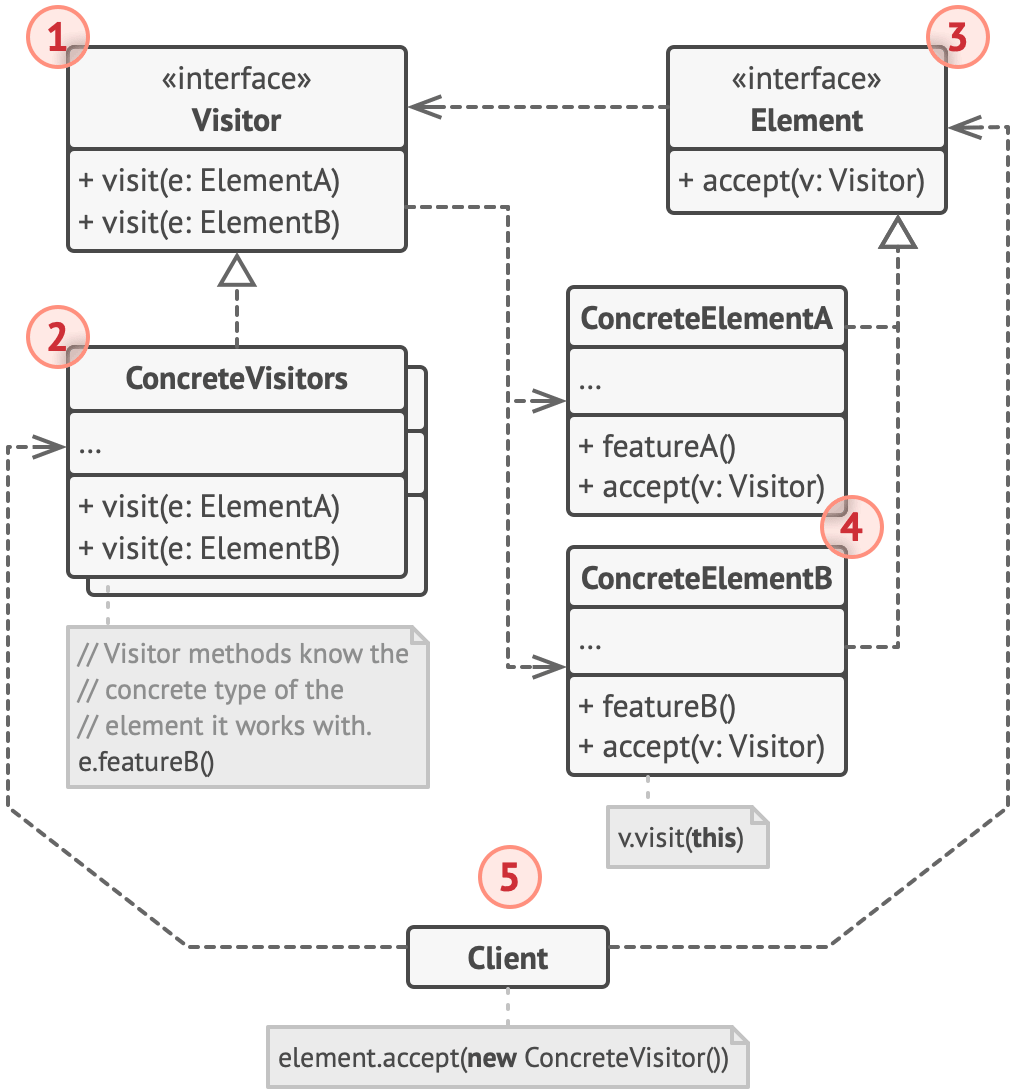
```
// Client code
foreach (Node node in graph)
    node.accept(exportVisitor)

// City
class City is
    method accept(Visitor v) is
        v.doForCity(this)
    // ...

// Industry
class Industry is
    method accept(Visitor v) is
        v.doForIndustry(this)
    // ...
```

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/visitor]

Visitor: Structure

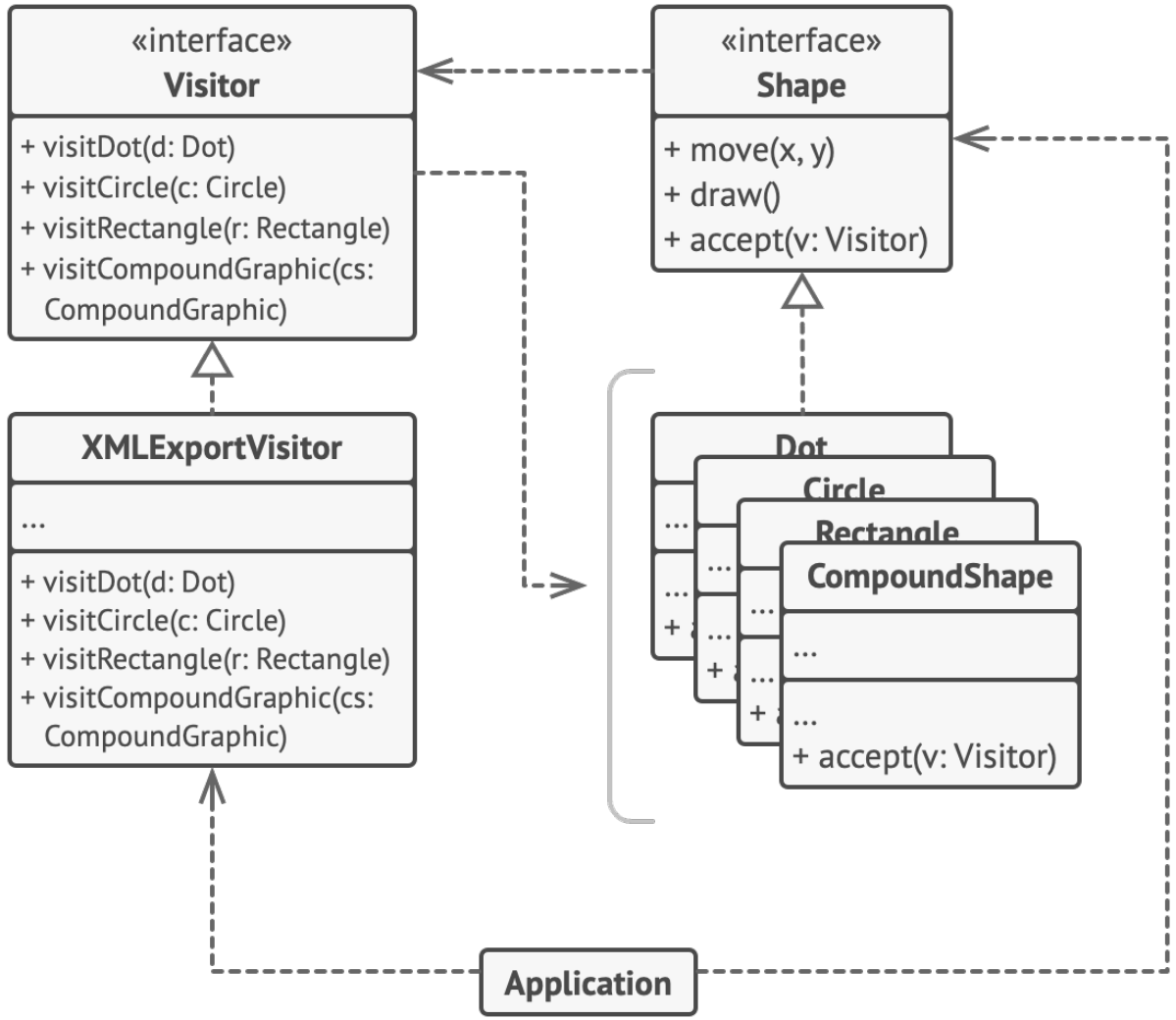


1. **Visitor:** interface, a set of visiting methods taking concrete elements
2. **Concrete Visitor:** several versions of the same behaviors, tailored for concrete element classes
3. **Element:** interface, a method for **accepting** visitors
4. **Concrete Element**: implementing the acceptance method, **redirecting** the call to the proper visitor's method
5. **Client:** usually a collection or other complex object

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/visitor]

Visitor: Example

- **XML export support to a class hierarchy of geometric shapes**



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/visitor]

Visitor: Applicability

- To perform an operation on all elements of a complex object structure
  - Execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation
- To clean up the business logic of auxiliary behaviors
  - Make the primary classes more focused on their main jobs, by extracting all other behaviors into a set of visitor classes
- When a behavior makes sense only in some classes of a class hierarchy, but not in others
  - Extract this behavior into a separate visitor class, and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty

Visitor: Implementation

1. Declare the **visitor interface**, with a set of **"visiting" methods**
2. Declare the **element interface** (or modify an existing base class), with an **abstract "acceptance" method** accepting a visitor object
3. Implement the acceptance methods in all concrete element classes, which simply **redirect the call** to a visiting method on the incoming visitor object
4. The element classes only work with visitors via the interface
5. For each behavior that cannot be implemented inside the element hierarchy, create a new concrete visitor class and implement all visiting methods
   - To access private members of the element class, either make these fields or methods public or nest the visitor class in the element class
6. The client must create visitor objects and pass them into elements via "acceptance" methods

Visitor: Pros and Cons

- **Pros**
  - Open/Closed Principle: introducing new behaviors to work with objects of different classes, without changing existing classes
  - Single Responsibility Principle: moving multiple versions of the same behavior into the same class
  - A visitor object can accumulate some useful information while working with various objects
- **Cons**
  - Need to update all visitors each time a class gets added to or removed from the element hierarchy
  - Might lack the necessary access to the private fields and methods of the elements

Combinations

- **Visitor and Command**
  - Treat Visitor as a powerful version of the Command pattern
- **Visitor and Composite**
  - Use Visitor to execute an operation over an entire Composite tree
- **Visitor and Iterator**
  - Use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements