6.24 Mẫu thiết kế VISITOR

6.24.1 Mở đầu

Để hiểu mẫu thiết kế *Visitor,* chúng ta xem lại mẫu thiết kế *Composite*. Mẫu *Composite* cho phép bạn tích hợp các đối tượng vào một cấu trúc cây để thể hiện phân cấp một phần – tổng thể.

Trong ví dụ mẫu *Composite*, chúng ta đã tạo cấu trúc *html* hợp thành từ các kiểu đối tượng khác nhau. Bây giờ giả sử chúng ta cần bổ sung một lớp *css* vào *tag html*. Một cách làm điều này là bằng cách bổ sung một lớp khi thêm *start tag* sử dụng phương thức *setStartTag*. Nhưng đặt code cứng này sẽ tạo nên tính không mềm dẻo trong code của chúng ta.

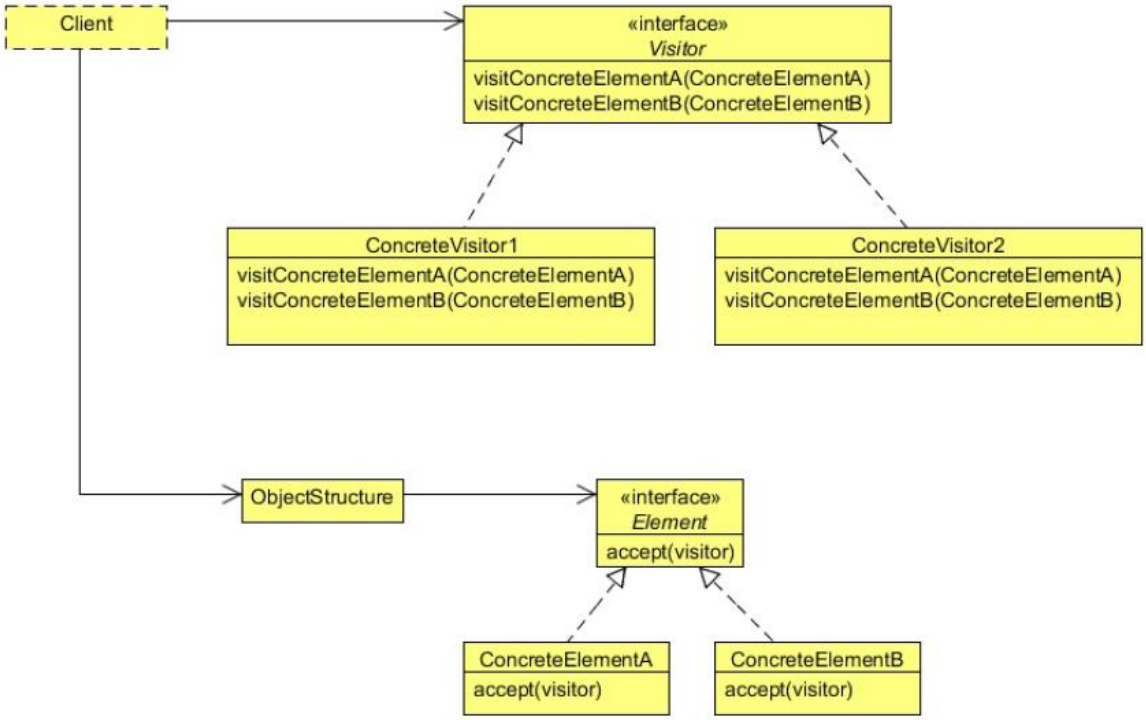
Một cách khác làm điều này là bổ sung phương thức mới kiểu *addClass* trong lớp trừu tượng *HtmlTag*. Mọi con của lớp đó sẽ ghi đè phương thức này và sẽ cung cấp lớp *css*. Một nhược điểm chính của cách tiếp cận này là, nếu có nhiều lớp con (sẽ ở trong trang html lớn), nó sẽ trở nên rất đắt và buộc phải cài đặt phương thức này trong mọi lớp con của nó. Và giả sử, sau này chúng ta cần bổ sung phần tử kiểu khác trong tag, chúng ta cũng cần phải làm như vậy.

Mẫu *Visitor* cung cấp cho bạn cách bổ sung các thao tác mới trên các đối tượng mà không cần thay đổi các lớp của các phần tử, đặc biệt khi các thao tác thay đổi khá thường xuyên.

6.24.2 Mẫu thiết kế Visitor là gì

Mục đích của mẫu thiết kế *Visitor* là biểu diễn một thao tác được thực hiện trên các phần tử của một cấu trúc đối tượng. *Visitor* cho phép bạn định nghĩa một thao tác mới mà không thay đổi các lớp của các phần tử mà trên đó nó thao tác.

Mẫu Visitor là hữu ích khi thiết kế một thao tác dọc theo họ không đồng nhất các đối tượng của một phân cấp lớp. Mẫu *Visitor* cho phép thao tác đó được định nghĩa mà không thay đổi lớp của bất cứ đối tượng nào trong họ đó. Để thực hiện điều đó, mẫu *Visitor* đề xuất định nghĩa thao tác trong một lớp riêng được tham chiếu là lớp visitor. Nó tách bạch thao tác đó khỏi họ đối tượng mà nó thao tác trên đó. Với mỗi thao tác được định nghĩa mới, một lớp *visitor* mới được tạo. Vì thao tác này được thực hiện dọc theo tập các đối tượng, *visitor* đó cần có cách truy cập các thành viên *public* của các đối tượng đó. Yêu cầu này có thể được đáp ứng bằng cách cài đặt hai ý tưởng thiết kế sau.



Visitor

- Khai báo một thao tác *Visit* cho mỗi lớp *ConcreteElement* trong cấu trúc đối tượng. Tên và đối số của thao tác này định danh lớp mà gửi yêu cầu *visit* cho *visitor*. Điều này cho phép *visitor* xác định lớp *concrete* của phần tử cần được viếng thăm. Sau đó *visitor* này có thể truy cập trực tiếp đến phần tử thông qua một giao diện cụ thể của nó.

ConcreteVisitor

- Cài đặt mỗi thao tác được định nghĩa trong *Visitor*. Mỗi thao tác cài đặt một đoạn của một thuật toán được định nghĩa cho lớp tương ứng của đối tượng trong cấu trúc đó. *ConcreteVisitor* cung cấp ngữ cảnh cho thuật toán đó và lưu giữ trạng thái cục bộ của nó. Trạng thái này thường tích góp các kết quả dọc theo quá trình viếng thăm cấu trúc đó.

Element

- Định nghĩa thao tác *Accept* mà nhận đối số là *visitor*

ConcreteElement

Cài đặt thao tác *Accept* mà nhận đối số là *visitor*

ObjectStructure

- Có thể liệt kê các phần tử của nó
- Có thể cung cấp giao diện mức độ cao để cho phép *visitor* viếng thăm các phần tử của nó.
- Có thể hoặc là một hợp thành hoặc một họ như một danh sách hoặc một tập hợp.
- **6.24.2 Mẫu thiết kế Visitor là gì**

6.24.3 Cài đặt mẫu thiết kế Visitor

Để cài đặt mẫu thiết kế *Visitor*, chúng ta sử dụng cùng *Composite Pattern code* và sẽ đưa vào một số giao diện, lớp và phương thức mới.

Cài đặt mẫu *Visitor* yêu cầu hai giao diện quan trọng, giao diện *Element* mà chứa một phương thức *accept* với đối số kiểu *Visitor*. Giao diện này sẽ được cài đặt bởi tất cả các lớp mà cần để cho phép các *visitor* viếng thăm chúng. Trong trường hợp của chúng ta, *Htmltag* sẽ cài đặt giao diện *Element*, như *HtmlTag* là lớp cha trừu tượng của mọi lớp *html* cụ thể, các lớp cụ thể này sẽ kế thừa và sẽ ghi đè phương thức *accept* của giao diện *Element.*

Giao diện khác là giao diện *Visitor*, giao diện này sẽ chứa các phương thức *visit* với các đối số của một lớp mà cài đặt giao diện *Element*. Lưu ý rằng, chúng ta sẽ yêu cầu hai phương thức mới trong lớp *HtmlTag* của chúng ta, *getStartTag* và *getEndTag*, ngược lại với ví dụ chỉ ra trong Bài mẫu thiết kế *Composite.*

```
package com.javacodegeeks.patterns.visitorpattern;

public interface Element {

        public void accept(Visitor visitor);
}

package com.javacodegeeks.patterns.visitorpattern;

public interface Visitor {
        public void visit(HtmlElement element);
        public void visit(HtmlParentElement parentElement);
}
```

Code dưới đây lấy từ Ví dụ Bài mẫu thiết kế Composite với một ít thay đổi:

```
package com.javacodegeeks.patterns.visitorpattern;

import java.util.List;


public abstract class HtmlTag implements Element{

        public abstract String getTagName();
        public abstract void setStartTag(String tag);
        public abstract String getStartTag();
        public abstract String getEndTag();
        public abstract void setEndTag(String tag);
        public void setTagBody(String tagBody){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public void addChildTag(HtmlTag htmlTag){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public void removeChildTag(HtmlTag htmlTag){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public List<HtmlTag>getChildren(){
                throw new UnsupportedOperationException("Current operation is not support  ↩
                    for this object");
        }
        public abstract void generateHtml();

}
```

Lớp trừu tượng *HtmlTag* cài đặt giao diện *Element*. Các lớp cụ thể dưới đây sẽ ghi đè phương thức *accept* của giao diện *Element* và sẽ gọi phương thức *visit* và sẽ truyền thao tác này như một đối số. Điều này cho phép phương thức *visitor* nhận mọi thành viên *public* của đối tượng này, bổ sung các thao tác mới trên nó.

```
package com.javacodegeeks.patterns.visitorpattern;

import java.util.ArrayList;
import java.util.List;

public class HtmlParentElement extends HtmlTag {

	private String tagName;
	private String startTag;
	private String endTag;
	private List<HtmlTag>childrenTag;

	public HtmlParentElement(String tagName){
		this.tagName = tagName;
		this.startTag = "";
		this.endTag = "";
		this.childrenTag = new ArrayList<>();
	}

	@Override
	public String getTagName() {
		return tagName;
	}

	@Override
	public void setStartTag(String tag) {
		this.startTag = tag;
	}

	@Override
	public void setEndTag(String tag) {
		this.endTag = tag;
	}

	@Override
	public String getStartTag() {
		return startTag;
	}

	@Override
	public String getEndTag() {
		return endTag;
	}

	@Override
	public void addChildTag(HtmlTag htmlTag){
		childrenTag.add(htmlTag);
	}

	@Override
	public void removeChildTag(HtmlTag htmlTag){
		childrenTag.remove(htmlTag);
	}

	@Override
	public List<HtmlTag>getChildren(){
		return childrenTag;
	}
```

```
        @Override
        public void generateHtml() {
                System.out.println(startTag);
                for(HtmlTag tag : childrenTag){
                        tag.generateHtml();
                }
                System.out.println(endTag);

        }

        @Override
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }

}
```

```
package com.javacodegeeks.patterns.visitorpattern;

public class HtmlElement extends HtmlTag{

        private String tagName;
        private String startTag;
        private String endTag;
        private String tagBody;

        public HtmlElement(String tagName){
                this.tagName = tagName;
                this.tagBody = "";
                this.startTag = "";
                this.endTag = "";
        }

        @Override
        public String getTagName() {
                return tagName;
        }

        @Override
        public void setStartTag(String tag) {
                this.startTag = tag;
        }

        @Override
        public void setEndTag(String tag) {
                this.endTag = tag;
        }

        @Override
        public String getStartTag() {
                return startTag;
        }

        @Override
        public String getEndTag() {
                return endTag;
        }

        @Override
        public void setTagBody(String tagBody){
                this.tagBody = tagBody;
```

```
        }

        @Override
        public void generateHtml() {
                System.out.println(startTag+""+tagBody+""+endTag);
        }

        @Override
        public void accept(Visitor visitor) {
                visitor.visit(this);
        }

}
```

Bây giờ, các lớp *concrete visitor*: chúng ta tạo hai lớp *concrete*, một sẽ bổ sung lớp *visitor* là *css* cho mọi html tag và cái khác là thay đổi độ rộng *tag* sử dụng thuộc tính *style* của *html tag*.

```
package com.javacodegeeks.patterns.visitorpattern;

public class CssClassVisitor implements Visitor{

        @Override
        public void visit(HtmlElement element) {
                element.setStartTag(element.getStartTag().replace(">", " class='visitor'>") ↩
                    );

        }

        @Override
        public void visit(HtmlParentElement parentElement) {
                parentElement.setStartTag(parentElement.getStartTag().replace(">", " class ↩
                    ='visitor'>"));
        }

}
```

```
package com.javacodegeeks.patterns.visitorpattern;

public class StyleVisitor implements Visitor {

        @Override
        public void visit(HtmlElement element) {
                element.setStartTag(element.getStartTag().replace(">", " style='width:46px ↩
                    ;'>"));

        }

        @Override
        public void visit(HtmlParentElement parentElement) {
                parentElement.setStartTag(parentElement.getStartTag().replace(">", " style ↩
                    ='width:58px;'>"));
        }

}
```

Bây giờ chúng ta kiểm tra ví dụ trên.

```
package com.javacodegeeks.patterns.visitorpattern;

public class TestVisitorPattern {
```

```
    public static void main(String[] args) {

            System.out.println("Before visitor..... \\n");

            HtmlTag parentTag = new HtmlParentElement("<html>");
            parentTag.setStartTag("<html>");
            parentTag.setEndTag("</html>");

            HtmlTag p1 = new HtmlParentElement("<body>");
            p1.setStartTag("<body>");
            p1.setEndTag("</body>");

            parentTag.addChildTag(p1);

            HtmlTag child1 = new HtmlElement("<P>");
            child1.setStartTag("<P>");
            child1.setEndTag("</P>");
            child1.setTagBody("Testing html tag library");
            p1.addChildTag(child1);

            child1 = new HtmlElement("<P>");
            child1.setStartTag("<P>");
            child1.setEndTag("</P>");
            child1.setTagBody("Paragraph 2");
            p1.addChildTag(child1);

            parentTag.generateHtml();

            System.out.println("\\nAfter visitor..... \\n");

            Visitor cssClass = new CssClassVisitor();
            Visitor style = new StyleVisitor();

            parentTag = new HtmlParentElement("<html>");
            parentTag.setStartTag("<html>");
            parentTag.setEndTag("</html>");
            parentTag.accept(style);
            parentTag.accept(cssClass);

            p1 = new HtmlParentElement("<body>");
            p1.setStartTag("<body>");
            p1.setEndTag("</body>");
            p1.accept(style);
            p1.accept(cssClass);

            parentTag.addChildTag(p1);

            child1 = new HtmlElement("<P>");
            child1.setStartTag("<P>");
            child1.setEndTag("</P>");
            child1.setTagBody("Testing html tag library");
            child1.accept(style);
            child1.accept(cssClass);

            p1.addChildTag(child1);

            child1 = new HtmlElement("<P>");
            child1.setStartTag("<P>");
            child1.setEndTag("</P>");
            child1.setTagBody("Paragraph 2");
            child1.accept(style);
            child1.accept(cssClass);
```

```
                p1.addChildTag(child1);

                parentTag.generateHtml();
        }

}
```

Code trên sẽ in đầu ra như sau:

```
Before visitor.........

<html>
<body>
<P>Testing html tag library</P>
<P>Paragraph 2</P>
</body>
</html>

After visitor.......

<html style='width:58px;' class='visitor'>
<body style='width:58px;' class='visitor'>
<p style='width:46px;' class='visitor'>Testing html tag library</P>
<p style='width:46px;' class='visitor'>Paragraph 2</P>
</body>
</html>
```

Đầu ra sau "*Before Visitor ...*" là giống với kết quả trong Ví dụ ở Bài mẫu *Composite*. Sau đó, chúng ta tạo hai *concrete visitor* và bổ sung chúng cho các đối tượng *concrete html* sử dụng phương thức accept. Đầu ra "*After visitor ...*" chỉ ra cho bạn rằng ở đó lớp *css* và các phần tử *style* đã được bổ sung cho các *html tag.*

Lưu ý rằng, ưu điểm của mẫu *Visitor* là ở chỗ chúng ta có thể thêm các thao tác mới cho các đối tượng mà không thay đổi các lớp này. Chẳng hạn, chúng ta có thể bổ sung một số chức năng của *Javascript* như *onclick* hoặc một số *angular js ng tags* mà không thay đổi các lớp đó.

6.24.4 Khi nào sử dụng mẫu thiết kế Visitor

Sử dụng mẫu thiết kế *Visitor* khi:

- Một cấu trúc đối tượng chứa nhiều lớp các đối tượng có sự khác nhau về giao diện và bạn muốn thực hiện các thao tác trên các đối tượng này mà phụ thuộc các lớp cụ thể của chúng.
- Nhiều thao tác không liên quan và khác nhau cần được thực hiện trên các đối tượng trong cấu trúc các đối tượng, và chúng ta muốn tránh làm ảnh hưởng các lớp của họ với các thao tác này. *Visitor* cho phép bạn giữ các thao tác liên quan cùng nhau bởi định nghĩa chúng trong một lớp. Khi cấu trúc các đối tượng được chia sẻ bởi nhiều ứng dụng, sử dụng *Visitor* để đặt các thao tác trong các ứng dụng mà cần đến chúng.
- Các lớp định nghĩa cấu trúc đối tượng rất ít khi thay đổi, nhưng bạn thường xuyên muốn định nghĩa các thao tác mới trên cấu trúc đó. Thay đổi các lớp cấu trúc đối tượng đòi hỏi định nghĩa lại giao diện giao diện cho mọi *visitor*, mà là phải trả giá. Nếu các lớp cấu trúc đối tượng thay đổi thường xuyên, thì có thể tốt hơn là định nghĩa các thao tác trong các lớp đó.