6.18 Mẫu thiết kế STATE

6.18.1 Mở đầu

Để minh họa sử dụng mẫu thiết kế trạng thái, giả sử chúng ta giúp một công ty mà đang xây dựng một robot nấu ăn. Công ty muốn một con robot đơn giản mà chỉ là đi lại và nấu ăn. Người sử dụng cần vận hành con robot sử dụng tập các lệnh thông qua điều khiển từ xa. Hiện tại, con robot cần làm ba việc walk, cook hoặc nó có thể được tắt.

Công ty thiết lập các giao thức để định nghĩa chức năng của robot. Nếu robot đang ở trạng thái on, bạn có thể ra lệnh cho nó walk, nếu yêu cầu cook, trạng thái của nó chuyển sang "cook" hoặc nó ở trạng thái "off", nếu nó được tắt đi.

Tương tự, khi đang ở trạng thái "cook", nó có thể walk hoặc cook, nhưng không thể bị tắt. Và cuối cùng, khi đang ở trạng thái "off" nó sẽ tự động trở thành "on" và walk, khi người sử dụng ra lệnh cho nó walk, nhưng không thể cook ở trạng thái "off".

Có thể trông là dễ cài đặt lớp robot với tập các phương thức walk, cook, off và các trạng thái như là on, cook và off. Bạn có thể sử dụng các rẽ nhánh if-else hoặc switch để cài đặt các giao thức thiết lập bởi công ty. Nhưng rất nhiều lệnh if-else hoặc switch sẽ tạo ra một cơn ác mộng bao trì và độ phức tạp sẽ tăng lên trong tương lai.

Bạn sẽ nghĩ là chúng ta có thể cài đặt với các lệnh if-then mà không gặp vấn đề gì, nhưng một sư thay đổi sẽ làm cho code trở nên phức tạp hơn. Yêu cầu này chỉ ra rõ ràng rằng hành vi của đối tượng là hoàn toàn dựa trên trạng thái của đối tượng. Chúng ta có thể sử dụng mẫu thiết kế State mà đóng gói các trạng thái của đối tượng vào lớp riêng khác và giữ lớp context độc lập với bất cứ thay đổi trạng thái nào.

Trước hết tìm hiểu về mẫu thiết kế State và sau đó chúng ta cài đặt nó để giải bài toán bên trên.

6.18.2 Mẫu thiết kế State là gì

Mẫu thiết kế *State* cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của đối tượng thay đổi. Đối tượng sẽ dường như là thay đổi lớp của nó.

Trạng thái của đối tượng có thể được định nghĩa như điều kiện chính xác của nó tại một thời điểm cho trước, phụ thuộc vào giá trị của các tính chất hoặc các thuộc tính. Tập các phương thức được cài đặt bởi một lớp tạo thành hành vi của các khởi tạo của nó. Khi có thay đổi giá trị của các thuộc tính của nó, chúng ta nói trạng thái của đối tượng đã thay đổi.

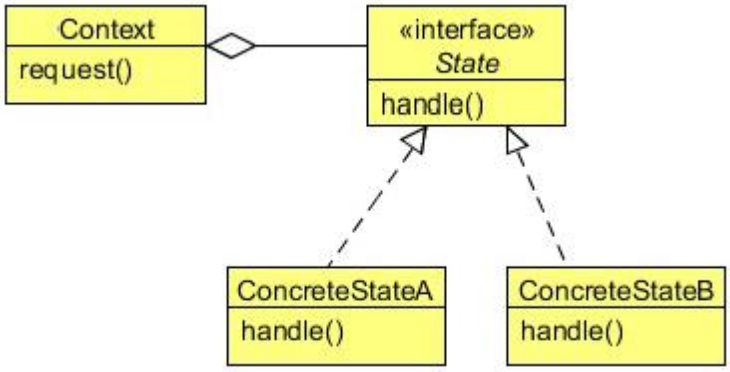
Mẫu *State* là hữu ích trong thiết kế cấu trúc hiệu quả cho một lớp, một khởi tạo tiêu biểu của nó có thể tồn tại trong nhiều trạng thái khác nhau và thể hiện các hành vi khác nhau phụ thuộc vào trạng thái nào nó đang ở trong đó. Nói cách khác, trong trường hợp của một đối tượng của lớp như vậy, một số hoặc tất cả các hành vi là hoàn toàn bị ảnh hưởng bởi trạng thái hiện tại. Trong mẫu thiết kế *State* thuật ngữ, như một lớp được tham chiếu là lớp *Context*. Một đối

tượng *Context* có thể thay đổi hành vi của nó khi có sự thay đổi ở trạng thái bên trong và nó cũng được tham chiếu là đối tượng có trạng thái Stateful.

Mẫu trạng thái đề xuất chuyển hành vi chuyên biệt trạng thái ra khỏi lớp *Context* vào tập các lớp riêng được tham chiếu là các lớp *State*. Mỗi trong số nhiều trạng thái mà đối tượng *Context* có thể ở trong đó có thể được ánh xạ vào một lớp *State* riêng biệt. Việc cài đặt lớp *State* chứa hành vi *context* mà chuyên biệt cho *state* đã cho, không chứa hành vi tổng thể của bản thân *Context*. *Context* hoạt động như *client* đến tập các đối tượng *State* theo nghĩa là nó sử dụng các đối tượng trạng thái khác nhau để thể hiện hành vi chuyên biệt trạng thái cần thiết cho đối tượng ứng dụng mà sử dụng *context*.

Bằng việc đóng gói hành vi chuyên biệt trạng thái vào các lớp riêng, cài đặt *contex*t trở nên đơn giản hơn để đọc: không có quá nhiều các lệnh điều kiện như *if-else* hoặc cấu trúc *switch - case*. Khi một đối tượng *Context* được tạo đầu tiên, nó khởi tạo nó với đối tượng trạng thái *State* ban đầu. Đối tượng *State* này trở thành đối tượng trạng thái hiện tại cho *Context*. Bằng cách thay đối tượng trạng thái hiện tại với đối tượng trạng thái mới, *context* chuyển sang trạng thái mới.

Ứng dụng *client* sử dụng *context* không có trách nhiệm đặc tả đối tượng trạng thái hiện tại cho *context*, nhưng thay vào đó, mỗi lớp trạng thái State mà thể hiện trạng thái chuyên biệt là được mong đợi để cung cấp cài đặt cần thiết để chuyển *context* sang trạng thái khác. Khi đối tượng ứng dụng gọi tới một phương thức của *context* (hành vi), nó chuyển tiếp lời gọi phương thức cho đối tượng trạng thái hiện tại.



Context

- Định nghĩa giao diện quan tâm cho client
- Duy trì một khởi tạo của một lớp con *ConcreteState* mà xác định trạng thái hiện tại.

State

- Định nghĩa giao diện để đóng gói hành vi gắn kết với một trạng thái cụ thể của *Context*

Các lớp con ConcreteState

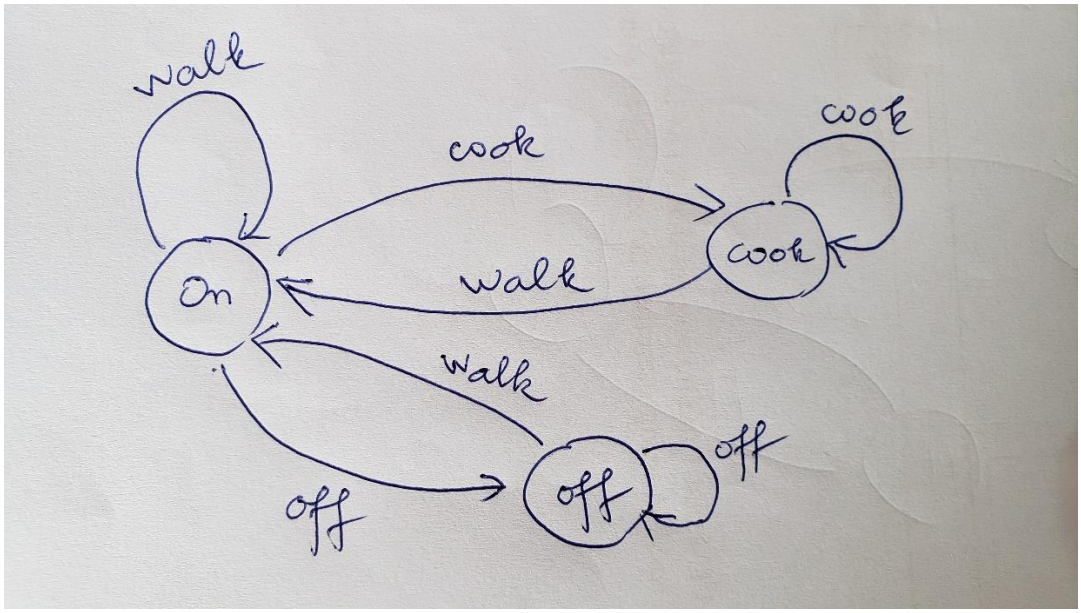
- Mỗi lớp con thể hiện hành vi gắn kết với một trạng thái của *Context*

6.18.3 Cài đặt mẫu thiết kế State

Sau đây là giao diện RoboticState mà chứa hành vi của Robot.

```
package com.javacodegeeks.patterns.statepattern;

public interface RoboticState {

        public void walk();
        public void cook();
        public void off();

}
```

Lớp Robot là lớp cụ thể cài đặt giao diện *RoboticsState*. Lớp này chứa tập các trạng thái mà robot có thể ở trong đó.



```
package com.javacodegeeks.patterns.statepattern;

public class Robot implements RoboticState{

	private RoboticState roboticOn;
	private RoboticState roboticCook;
	private RoboticState roboticOff;

	private RoboticState state;

	public Robot(){
		this.roboticOn = new RoboticOn(this);
		this.roboticCook = new RoboticCook(this);
		this.roboticOff = new RoboticOff(this);

		this.state = roboticOn;
	}

	public void setRoboticState(RoboticState state){
		this.state = state;
	}

	@Override
	public void walk() {
		state.walk();
	}

	@Override
	public void cook() {
		state.cook();

	}

	@Override
	public void off() {
		state.off();
	}

	public RoboticState getRoboticOn() {
		return roboticOn;
	}

	public void setRoboticOn(RoboticState roboticOn) {
		this.roboticOn = roboticOn;
	}

	public RoboticState getRoboticCook() {
		return roboticCook;
	}

	public void setRoboticCook(RoboticState roboticCook) {
		this.roboticCook = roboticCook;
	}

	public RoboticState getRoboticOff() {
		return roboticOff;
	}

	public void setRoboticOff(RoboticState roboticOff) {
```

```
                this.roboticOff = roboticOff;
        }

        public RoboticState getState() {
                return state;
        }

        public void setState(RoboticState state) {
                this.state = state;
        }

}
```

Lớp này khởi tạo mọi trạng thái và đặt trạng thái hiện tại là on.

Bây giờ ta sẽ thấy mọi trạng thái cụ thể của Robot. Một robot sẽ ở một trong số các trạng thái ở thời điểm bất kỳ.

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOn implements RoboticState{

        private final Robot robot;

        public RoboticOn(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("Walking...");
        }

        @Override
        public void cook() {
                System.out.println("Cooking...");
                robot.setRoboticState(robot.getRoboticCook());
        }

        @Override
        public void off() {
                robot.setState(robot.getRoboticOff());
                System.out.println("Robot is switched off");

        }

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticCook implements RoboticState{

        private final Robot robot;

        public RoboticCook(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("Walking...");
                robot.setRoboticState(robot.getRoboticOn());
        }
```

```
        @Override
        public void cook() {
                System.out.println("Cooking...");
        }

        @Override
        public void off() {
                System.out.println("Cannot switched off while cooking...");
        }
}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticOff implements RoboticState{

        private final Robot robot;

        public RoboticOff(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("Walking...");
                robot.setRoboticState(robot.getRoboticOn());
        }

        @Override
        public void cook() {
                System.out.println("Cannot cook at Off state.");
        }

        @Override
        public void off() {
                System.out.println("Already switched off...");
        }
}
```

Bây giờ ta kiểm tra code trên

```
package com.javacodegeeks.patterns.statepattern;

public class TestStatePattern {

        public static void main(String[] args) {
                Robot robot = new Robot();
                robot.walk();
                robot.cook();
                robot.walk();
                robot.off();

                robot.walk();
                robot.off();
                robot.cook();

        }

}
```

Code trên cho đầu ra như sau:

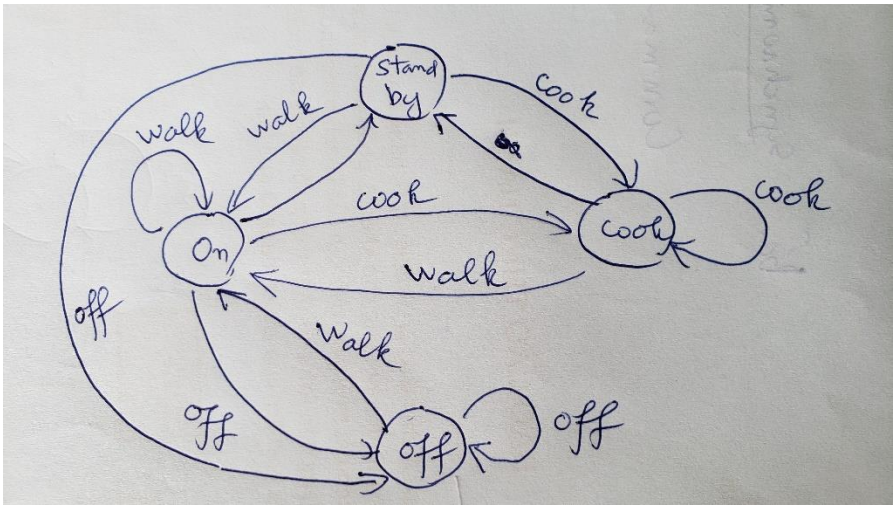
```
Walking...
Cooking...
Walking...
Robot is switched off
Walking...
Robot is switched off
Cannot cook at Off state.
```

Trong ví dụ trên, chúng ta đã nhìn thấy là bằng việc đóng gói các trạng thái của một đối tượng vào các lớp khác nhau làm cho code dễ quản trị và linh hoạt hơn.

Bất cứ thay đổi trong một trạng thái sẽ chỉ tác động đến một lớp cụ thể và chúng ta có thể bổ sung trạng thái mới mà không thay đổi nhiều trong code hiện tại. Giả sử chúng ta thêm trạng thái stand-by. Sau khi walk hoặc cook robot sẽ chuyển sang chế độ stand-by để tiết kiệm năng lượng, và chúng ta có thể ra lệnh walk, cook hoặc tắt từ chế độ stand-by.

Để cài đặt thêm điều này, chúng ta cần đưa ra lớp trạng thái mới và bổ sung trạng thái này vào lớp robot. Sau đây là các thay đổi.

```
package com.javacodegeeks.patterns.statepattern;

public class Robot implements RoboticState{

        private RoboticState roboticOn;
        private RoboticState roboticCook;
        private RoboticState roboticOff;
        private RoboticState roboticStandby;

        private RoboticState state;

        public Robot(){
                this.roboticOn = new RoboticOn(this);
                this.roboticCook = new RoboticCook(this);
                this.roboticOff = new RoboticOff(this);
                this.roboticStandby = new RoboticStandby(this);

                this.state = roboticOn;
        }

        public void setRoboticState(RoboticState state){
                this.state = state;
        }

        @Override
        public void walk() {
                state.walk();
                setState(getRoboticStandby());
        }

        @Override
        public void cook() {
                state.cook();
                setState(getRoboticStandby());
        }

        @Override
        public void off() {
                state.off();
        }

        public RoboticState getRoboticOn() {
                return roboticOn;
        }
```



```
        public void setRoboticOn(RoboticState roboticOn) {
                this.roboticOn = roboticOn;
        }

        public RoboticState getRoboticCook() {
                return roboticCook;
        }

        public void setRoboticCook(RoboticState roboticCook) {
                this.roboticCook = roboticCook;
        }

        public RoboticState getRoboticOff() {
                return roboticOff;
        }

        public void setRoboticOff(RoboticState roboticOff) {
                this.roboticOff = roboticOff;
        }

        public RoboticState getState() {
                return state;
        }

        public void setState(RoboticState state) {
                this.state = state;
        }

        public RoboticState getRoboticStandby() {
                return roboticStandby;
        }

        public void setRoboticStandby(RoboticState roboticStandby) {
                this.roboticStandby = roboticStandby;
        }

}
```

```
package com.javacodegeeks.patterns.statepattern;

public class RoboticStandby implements RoboticState{

private final Robot robot;

        public RoboticStandby(Robot robot){
                this.robot = robot;
        }

        @Override
        public void walk() {
                System.out.println("In standby state...");
                robot.setState(robot.getRoboticOn());
                System.out.println("Walking...");
        }

        @Override
        public void cook() {
                System.out.println("In standby state...");
                robot.setRoboticState(robot.getRoboticCook());
                System.out.println("Cooking...");
        }
```

```
        @Override
        public void off() {
                System.out.println("In standby state...");
                robot.setState(robot.getRoboticOff());
                System.out.println("Robot is switched off");

        }

}
```

Thay đổi code trên cho kết quả đầu ra:

```
Walking...
In standby state...
Cooking...
In standby state...
Walking...
In standby state...
Robot is switched off
Walking...
In standby state...
Robot is switched off
Cannot cook at Off state.
```

6.18.4 Khi nào sử dụng mẫu thiết kế State

Sử dụng mẫu *State* cho một trong các trường hợp sau:

- Một hành vi của một đối tượng phụ thuộc vào trạng thái của nó, và nó cần phải thay đổi hành vi của nó trong thời gian chạy phụ thuộc vào trạng thái đó.
- Các thao tác có các lệnh điều kiện nhiều phần, lớn mà phụ thuộc vào trạng thái đối tượng. Trạng thái này thường được biểu diễn bởi một hoặc nhiều ràng buộc.Thông thường, một số thao tác sẽ chứa các cấu trúc điều kiện. Mẫu *State* đặt mỗi nhánh của điều kiện thành một lớp riêng. Điều này cho phép bạn xử lý trạng thái đối tượng như một đối tượng ở trong chính nó mà có thể đa dạng phụ thuộc vào các đối tượng khác.