The state pattern is a pattern, imagine you have a Robot friend. It can have different moods like Happy and sad based on weather. The state-pattern will give your robot friend a set of rules, so that it knows when to be sad or happy.

```
class Brightness {
    private State state;

    public Brightness() {
        this.state = new LowState();
    }

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

    public void pressButton() {
        state.handleRequest(this);
    }

    public String toString() {
        return state.toString();
    }
}
```

This is the brightness class which will manage the state of brightness and this also has a reference to the current 'State' object.
This is initialized with a default state which in this case is LowState.
The also provides a method 'setState(State state) which will change the state.
This also has the method 'pressButton' which will control the handling of the button press to the current state.
This will override the 'toString()' to provide a string representation of the current state.

```
interface State {
    void handleRequest(Brightness brightness);
}
```

This is the interface State. Here I have declared the method 'handleRequest(Brightness brightness) that concrete states will implement.
This will represent the interface for all concrete states.

```
class LowState implements State {
    public void handleRequest(Brightness brightness) {
        brightness.setState(new MediumState());
    }

    public String toString() {
        return "Brightness is low";
    }
}

class MediumState implements State {
    public void handleRequest(Brightness brightness) {
        brightness.setState(new HighState());
    }

    public String toString() {
        return "Brightness is medium";
    }
}

class HighState implements State {
    public void handleRequest(Brightness brightness) {
        brightness.setState(new LowState());
    }

    public String toString() {
        return "Brightness is High";
    }
}
```

These are the State classes LowState, MediumState,Highstate.
They implement the interface ‘State’
They provide a specific behavior for handling requests in this case , if the button is pressed based on the current state.
This class will also update the state of the ‘brightness’ object when the request is handled.
This will also Override the ‘toString()’ to provide a string representation of the current brightness state.

There are a lot of benefits of using this Pattern in this code or in software in general.
Clean separation of concerns which means that each state (LowState,MediumState,HighState) encapsulates its behavior. This is beneficial because this separation makes it easier to understand this code because each state has a different behavior.If I did not use the pattern then I would end up with a lot of conditional statements in the ‘Brightness’ class to manage the different states. This could lead to the code being more complex, harder to read. Ease of Adding new states, which means that, if i need a new state of the brightness level then I can just create a new class which will implement the interface ‘State’. This all can happen without modifying the existing code. This flexibility made the code extensible. It would be hard to add new states, if i did not use this pattern then i would need to modify the existing code, which could potentially introduce new bugs. This would also violate the Open-Closed principle, which encourages extending behavio without modifying the existing code.