# デザインパターン
# 19.state

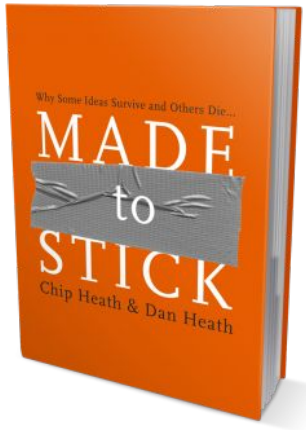
# どんなもの？　使い道は？

オブジェクト指向設計では、モノをクラスとして表現することが多くある。

State パターンとは、モノではなく、「状態」をクラスとして表現する パターンです。

**→状態の変化に応じて振る舞いが変わる場合に強い！**

具体例として、機嫌によって行動が変わる人を実装します！



# 由美ちゃんという、
# 機嫌によって行動が変わる人を実装！

```
public class Yumichan {
    private static final int STATE_ORDINARY = 0;    /** 通常の状態を表す */
    private static final int STATE_IN_BAD_MOOD = 1;     /** 機嫌の悪い状態を表す */
    private int state = -1;

    public void changeState(int state) {
        this.state = state;
    }

     public String morningGreet() {
        if (state == STATE_ORDINARY) {
            return "おっす!";
        } else if (state == STATE_IN_BAD_MOOD) {
            return "何か用？";
        } else {
            return "・・・";
        }
    }

    public String getProtectionForCold() {
        if (state == STATE_ORDINARY) {
            return "寒いから走るか～";
        } else if (state == STATE_IN_BAD_MOOD) {
            return "寒いけど防寒もめんどくさい..";
        } else {
            return "・・・";
        }
    }
}
```

```
yumichan = new Yumichan()
yumichan.changeState(0)

yumichan.morningGreet()
//おっす！

yumichan.getProtectionForCold()
//寒いから走るか～

yumichan.changeState(1)
yumichan.morningGreet()
//何か用？
```

# 問題はいつ発生する？→追加するとき、追加コードが膨大

```
public class Yumichan {
    private static final int STATE_ORDINARY = 0;    /** 通常の状態を表す */
    private static final int STATE_IN_BAD_MOOD = 1;    /** 機嫌の悪い状態を表す */
    private static final int STATE_IN_GOOD_MOOD = 2 /**機嫌の良い状態を追加**/
    private int state = -1;

    public void changeState(int state) {
        this.state = state;
    }

     public String morningGreet() {
        if (state == STATE_ORDINARY) {
            return "おっす!";
        } else if (state == STATE_IN_BAD_MOOD) {
            return "何か用？";
        } else if (state ==  STATE_IN_GOOD_MOOD){
             return "おはよー！超元気！！！！！！！！！！ "
        } else {
            return "・・・";
        }
    }

    public String getProtectionForCold() {
        if (state == STATE_ORDINARY) {
            return "寒いから走るか～";
        } else if (state == STATE_IN_BAD_MOOD) {
            return "寒いけど防寒めんどくさいから何もしないわ";
        } else if (state == STATE_IN_GOOD_MOOD){
            return "寒いけど元気！！！！防寒いらない！！！！！！ "
        } else {
            return "・・・";
        }
    }
}
```

追加コード量　＝　追加する状態数　X　実装してる挨拶の数
実装してる挨拶の数が多いほど、地獄を見る ..
else if 地獄..　**これをシンプルに解決する！**

## ○インターフェースに依存させる！！

```
interface State {
    public String morningGreet();
    public String getProtectionForCold();
}

public class StatePatternYumichan {
    private State state = null;

    private void changeState(State state) {
        this.state = state;
    }

    public String morningGreet() {
        return this.state.morningGreet();
    }

    public String getProtectionForCold() {
        return this.state.getProtectionForCold();
    }
}
```

**メリットは？**

- 変更に強くなる！！
- 書くコードが減る！！

機嫌が増えるごとにこれを追加するだけでOK！

```
class BadMoodState implements State {
    public String morningGreet() {
        return "おお";
    }
    public String getProtectionForCold() {
        return "寒いけど防寒めんどいから何もしないわ";
    }
}

class OrdinaryState implements State {
    public String morningGreet() {
        return "おっす！";
    }
    public String getProtectionForCold() {
        return "寒いから走るか～";
    }
}

//追加箇所はここだけでOK!
class GoodState implements State{
    public String morningGreet() {
        return "おはよー！超元気！！！！！！！！！!";
    }
    public String getProtectionForCold() {
        return "寒いけど元気！！防寒いらない！!";
    }
}
```

## ○Decoratorのまとめ

・抽象(インタフェース)に依存させてる
　前回のDecoratorとの共通だと感じた。

・Decoratorとの違いは？

前回はアイス＋トッピング（ Decorator)　という感じで、アイスが割とメインだった感。
今回はstateを追加するごとに、メイン処理・具体処理を記述してるイメージ。
あらかじめ抽象を実装しておいて、具体の実装は各 stateクラスに任せている。

→処理のすることや順番がある程度決まっていて、具体処理が細かく異なる場合に使うのかな？と思いました！