デザインパターン
12.Decorator

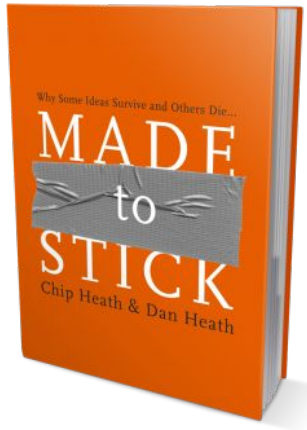
どんなもの？　使い道は？

・機能追加したい！

→そんな時に、そのクラスをいじるのではなく、そのクラスへの **装備品を作るイメージ**

・包まれるもの（アイス）を **変更することなく、機能の追加**（トッピング）を行うことができる！

・それを可能にするのがDecorator！



登場人物

共通の　アイスインタフェース

```
public interface Icecream{
    public String getName();
    public String howSweet();
}
```

バニラアイス

```
public class VanillaIcecream implements Icecream{
    public String getName(){
        return "バニラアイスクリーム";
    }
    public String howSweet(){
        return "バニラ味";
    }
}
```

抹茶アイス

```
public class GreenTeaIcecream implements Icecream{
    public String getName(){
        return "抹茶アイスクリーム";
    }
    public String howSweet(){
        return "抹茶味";
    }
}
```

バニラ＋カシューナッツをトッピングしたものを作りたい！

普通にやると..

```
public class CashewNutsVanillaIcecream extends VanillaIcecream{
    public String getName(){
        return "カシューナッツバニラアイスクリーム";
    }
}
```

こういうクラスが1個ならいいけど…
・抹茶アイスとか、チョコアイスにトッピングしたい時、毎回継承するの？
アイスがn個ある場合、トッピング１種類増やすだけでn個新しいクラス増やすの？

→これをDecoratorで解決！！

❌ダメな例

```
public class CashewNutsVanillaIcecream extends VanillaIcecream{
    public String getName(){
        return "カシューナッツバニラアイスクリーム";
    }
}
```

cashewNutsVanilla = new CashewNutsVanillaIceCream()

○Decoratorを導入すると..

```
public class CashewNutsToppingIcecream implements Icecream{
    private Icecream ice = null;
    public CashewNutsToppingIcecream(Icecream ice){
        this.ice = ice;
    }
    public String getName(){
        String name = "カシューナッツ";
        name += ice.getName();
        return name;
    }
    public String howSweet(){
        return ice.howSweet();
    }
}
```

cashewNutsVanilla = new CashewNutsToppingIcecream (VanillaIceCream())

メリットは？

- 変更に強くなる！！
- 書くコードが減る！！

どう嬉しいのか具体的に

❌ダメな例

```
public class CashewNutsVanillaIcecream extends VanillaIcecream{
    public String getName(){
        return "カシューナッツバニラアイスクリーム";
    }
}
```

○Decoratorを導入すると..

```
public class CashewNutsToppingIcecream implements Icecream{
    private Icecream ice = null;
    public CashewNutsToppingIcecream(Icecream ice){
        this.ice = ice;
    }
    public String getName(){
        String name = "カシューナッツ";
        name += ice.getName();
        return name;
    }
    public String howSweet(){
        return ice.howSweet();
    }
}
```

仕様変更への対応

レモンアイスと、イチゴアイス増やして！

あと、全部に黒蜜と、チョコソースをトッピングできるように！

チョコアイス（クラス）を作る

チョコアイス継承した黒蜜チョコアイスを作る…

チョコアイス継承した、チョコソースがけチョコアイスを作る…

イチゴアイスクラスを作る

イチゴアイスを継承した黒蜜イチゴアイスを作る..

イチゴアイス継承したチョコソースがけイチゴアイス作る..

チョコアイス（クラス）を作る

黒蜜トッピングクラスを作る

イチゴアイスクラスを作る

黒蜜トッピングクラスを作る

○Decoratorの自分的大事ポイント

```
public class CashewNutsToppingIcecream implements Icecream{
    private Icecream ice = null;
    public CashewNutsToppingIcecream(Icecream ice){
        this.ice = ice;
    }
    public String getName(){
        String name = "カシューナッツ";
        name += ice.getName();
        return name;
    }
    public String howSweet(){
        return ice.howSweet();
    }
}
```

this.ice = new VanillaIce()

としていたら,バニラアイスにしか対応できなくなる。
オブジェクト指向で大事なSOLID原則のうち

・抽象(インタフェース)に依存させる
→アイスインタフェースに依存したトッピングクラス

・拡張に対して開かれて、修正に対して閉じられていなければならない
→新しいトッピングも、アイスインタフェースに依存さえしてれば、なんでもOK

の2点が特に関係していて
これらを意識した結果発生したデザインパターンなのかな、という印象を受けた。
クラスの密結合を避けるためにnewを隠蔽せよって法則見かけたことあるけど、実例に出会った感。