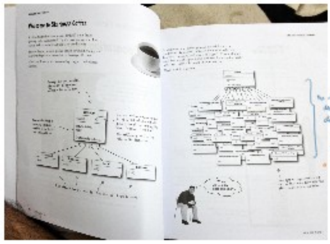
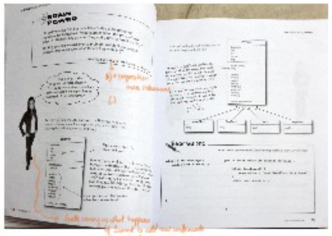
Decorator pattern

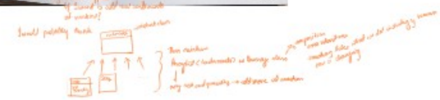
19 May 2025 06:26

- give objects new responsibilities (yours or someone else's) without making any code changes to underlying classes (extension at runtime).
- Problem of class explosion



The idea of adding only description and defining class name and cost
DarkRoast was ok
DarkRoastwithSteamedMilk and AlmondDarkRoast with SteamedMilk and Caramel → things like these cause class explosion, also everyone extends Beverage.
let's say price of milk goes up, many cost() will be affected, this becomes maintenance nightmare



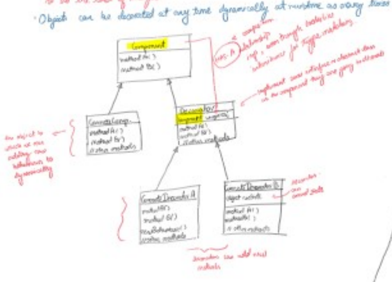


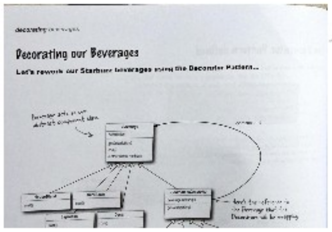
Open Closed Principle

Classes should be open for extension but closed for modification.
Be careful when choosing areas of code that need to be extended, applying open closed principle everywhere is wasteful and unnecessary → can lead to hard to understand and complex code.

- Decorators have the same super type as objects they decorate.
- You can use one or more decorators to wrap an object
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object
- Decorator adds its own behaviour before and/or after delegating to the object it decorates to do the rest of the job
- Objects can be decorated at any time dynamically at runtime as many times as we want

→ Maybe because subclass should do as superclass can point to subclass in java.





implementation

```
public abstract class Beverage {
    String description = "Unknown Beverage";
    public String getDescription(){
        return description;
    }
    public abstract double cost();
}

public abstract class CondimentDecorator extends Beverage {
    Beverage beverage;        → composition → HAS-A-relationship
    public abstract String getDescription();
}
```

we need it to be interchangeable with Beverage class.

Beverage supertype so decorator can add/wrap any Beverage

coding Beverages

```
public class Espresso extends Beverage {
    public Espresso() {
        description = Espresso
    }
    public double cost() {
        return 0.99;
    }
}
```

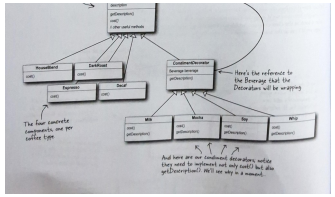
same for HouseBlend, DarkRoast, Decaf

coding condiments

```
public class Mocha extends CondimentDecorator {
    public Mocha (Beverage beverage) {
        this.beverage = beverage;
    }
```

driver

```
Beverage beverage = new Espresso()   → no condiments nothing
Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha ( beverage2);   → Mocha + DarkRoast
    beverage2 = new Mocha (beverage2);    → Mocha + Mocha + DarkRoast
    beverage2 = new Whip
```




```
public Mocha (Beverage beverage) {
    this.beverage = beverage;
}
public String getDescription() {
    return beverage.getDescription() + ", Mocha";
}
public double cost {
    return beverage.cost() + 0.2;
}
}
```