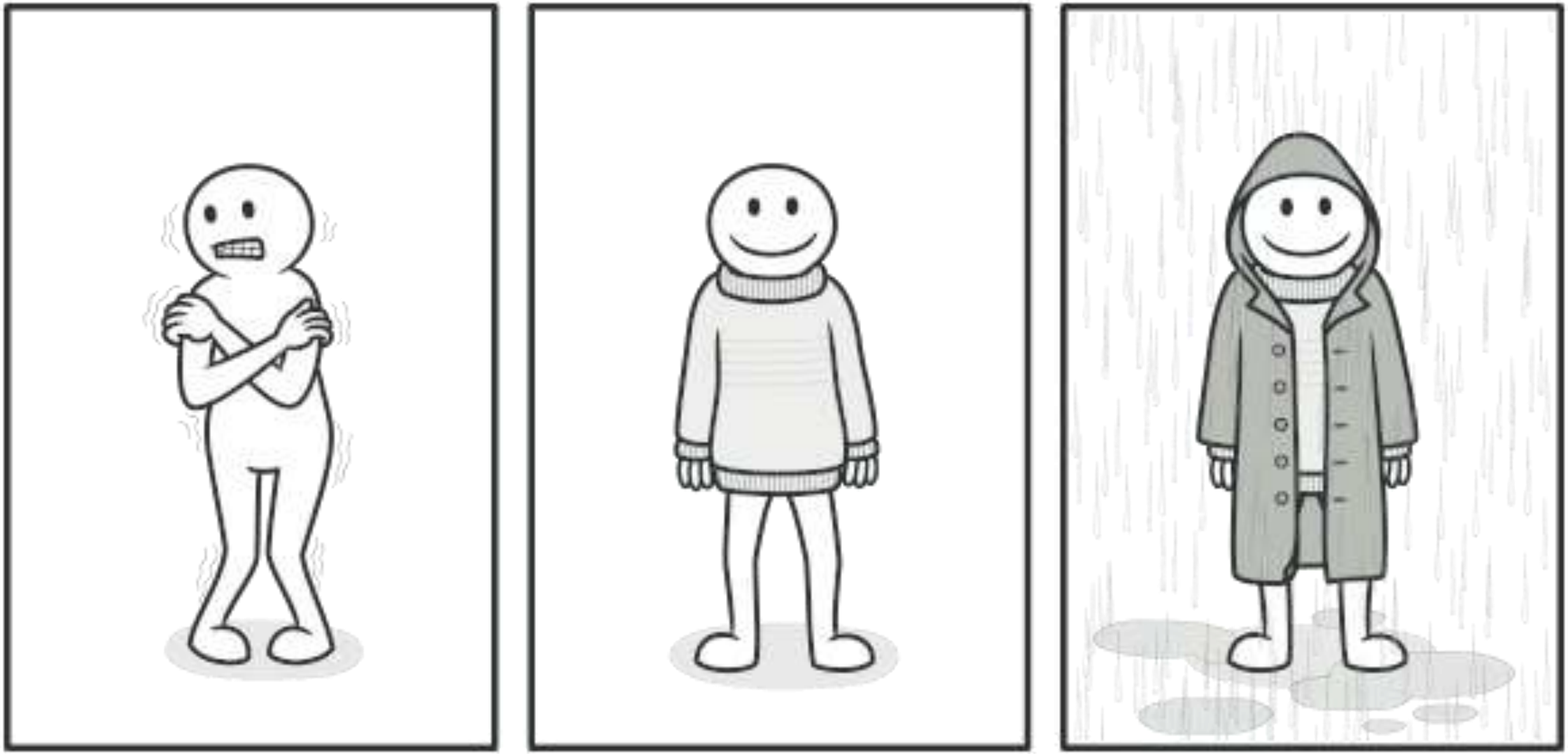
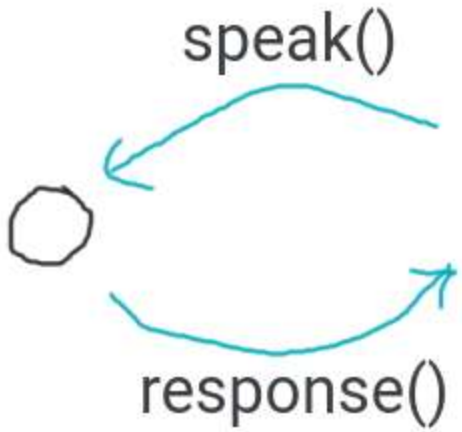
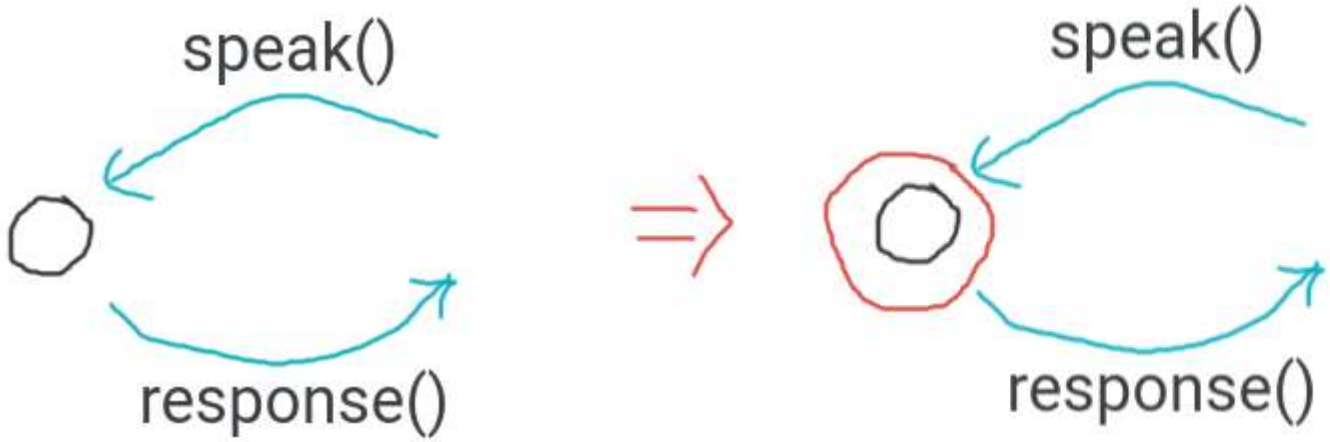
DECORATOR



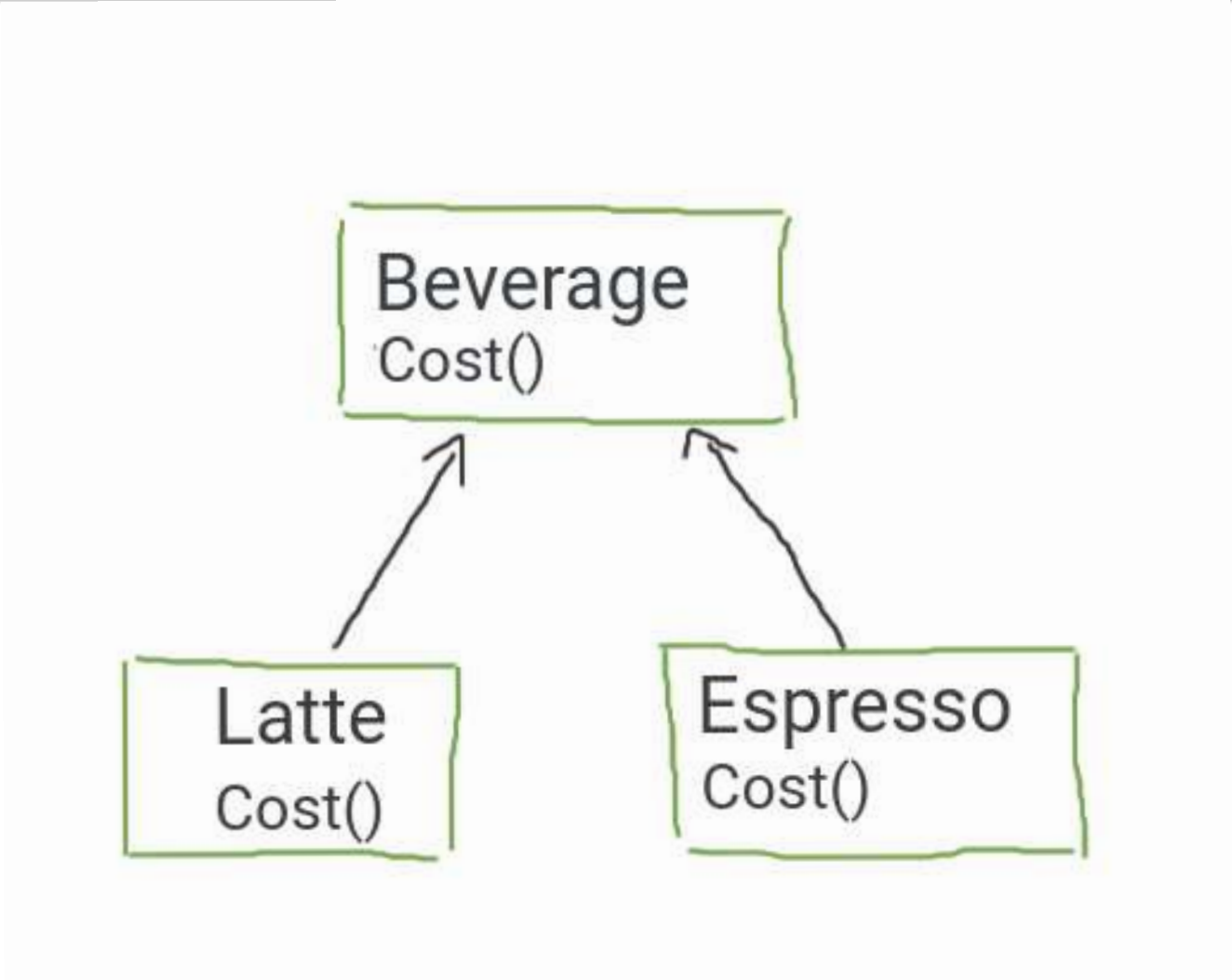
MOTIVACE



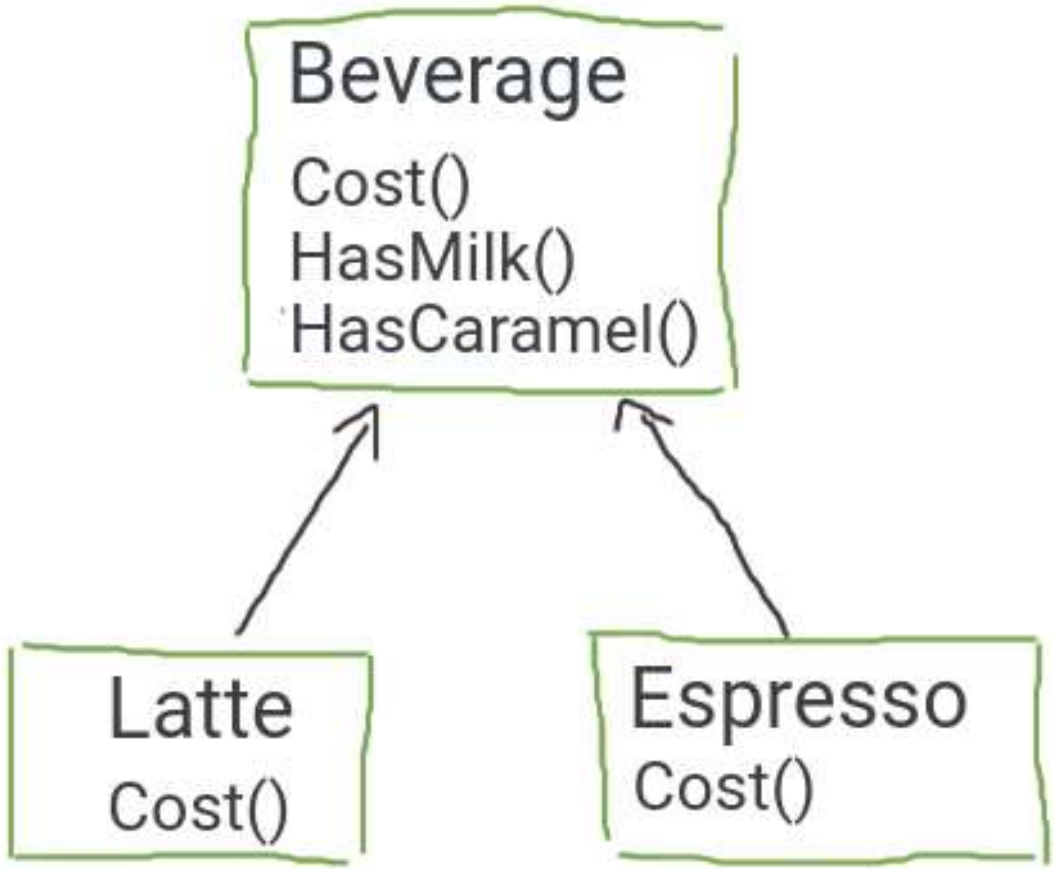
MOTIVACE



PŘÍKLAD

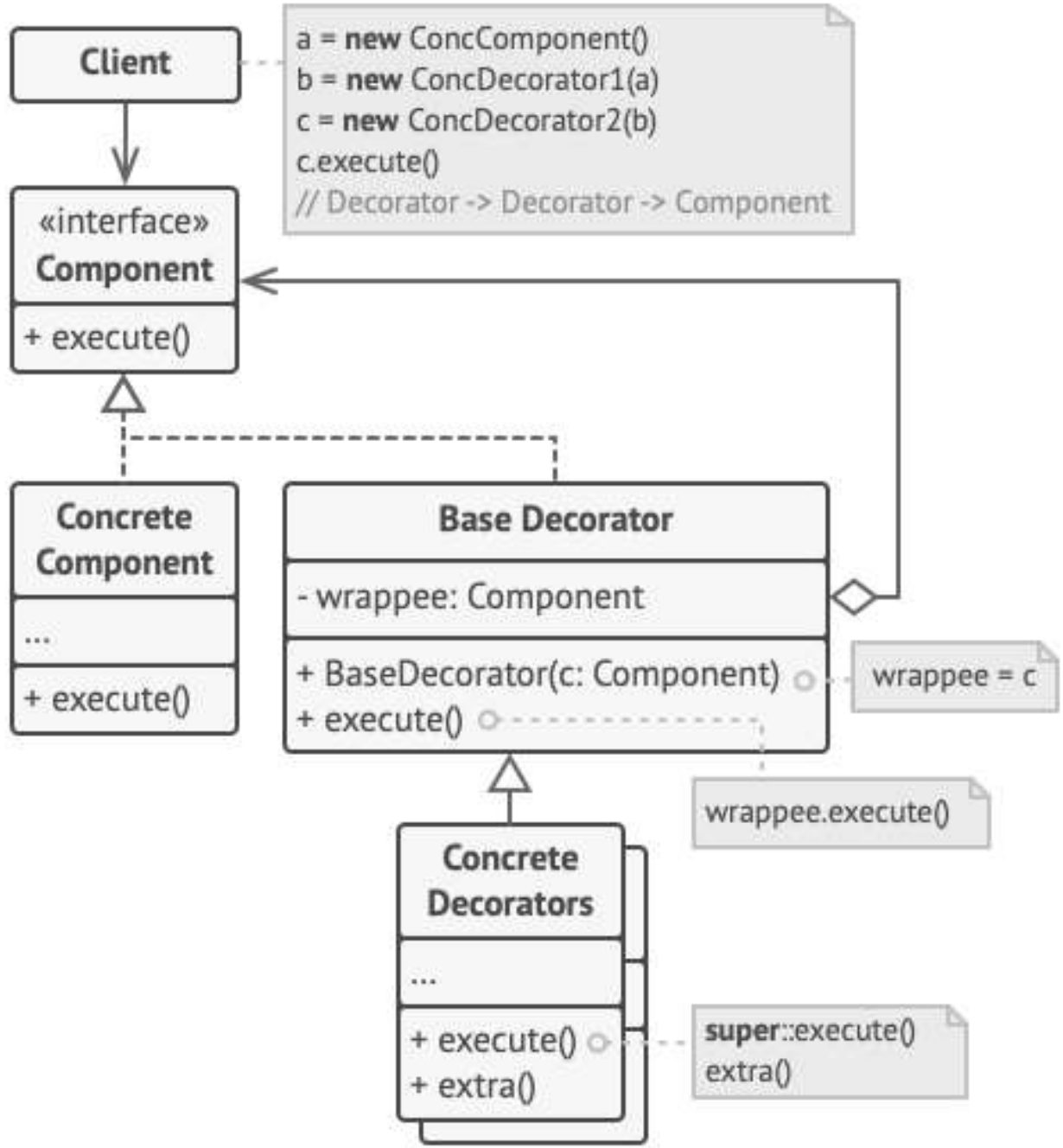


PŘÍKLAD

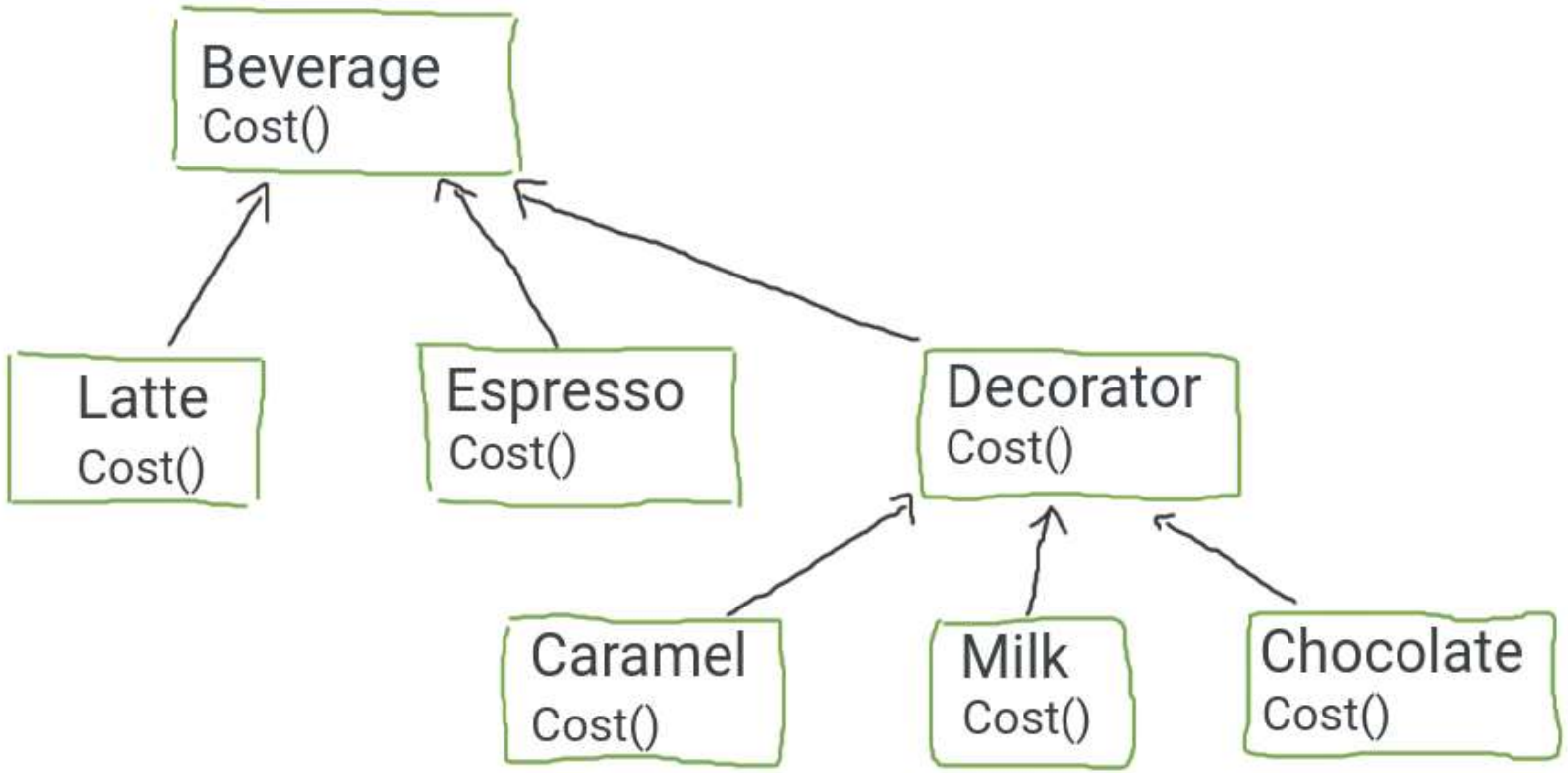

Beverage
Cost()
HasMilk()
HasCaramel()
Latte
Cost()
Espresso
Cost()

DEKORÁTOR

- Přidání nových vlastností zabalením třídy
- Kompozice místo dědičnosti
- **Dynamické změny chování objektu**
- Dekorátory **lze rekurzivně vrstvit**
  - Dekorátor a rozšiřovaná komponenta implementují stejný interface
- Vhodné použití
  - Rozšiřování pomocí podtříd není praktické
  - Implementace objektů s mnoha různými kombinacemi


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
«interface»
Component
+ execute()
Concrete
Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
Concrete
Decorators
...
+ execute()
+ extra()
super::execute()
extra()

ŘEŠENÍ PŘÍKLADU



IMPLEMENTACE

```
abstract class Beverage {
        public abstract int Cost();
}
```

```
class Espresso : Beverage {
        public abstract override int Cost() return 10;
}

class Latte : Beverage {
        public abstract override int Cost() return 9;
}

class Beer : Beverage {
        public abstract override int Cost() return 8;
}
```

IMPLEMENTACE

```
abstract class Decorator : Beverage {
        public abstract override int Cost();
}

class Caramel : Decorator {
        private Beverage beverage;

        public Caramel(Beverage b) => beverage = b;
        public override int Cost() => beverage.Cost()+5;
}

class Cream : Decorator {
        private Beverage beverage;

        public Cream(Beverage b) => beverage = b;
        public override int Cost() => beverage.Cost()+5;
}
```

```
Beverage b = new Espresso();

b = new Cream( b );
b = new Caramel( b );

Console.WriteLine($"Price: {b.Cost()}");
```

VYUŽITÍ

- UI návrhový vzor
- IOStream – Java, C#
- Změny stavu (pattern state)

VÝHODY VS. NEVÝHODY

Výhody

- Dynamické přidávání povinností objektům
- Úspora paměti oproti dědičnosti
- Kombinace chování díky vrstvení dekorátorů
  - Dekorátory jsou transparentní vůči komponentě
- „Single Responsibility Principle“
- „Open Closed Principle“
- Umožňuje rozšířit chování objektu bez vytváření podtříd

Nevýhody

- Mnoho podobných malých tříd
- Složité odstranění konkrétního dekorátoru
- Často závislé na pořadí vrstev
- Dlouhý řetěz dekorátorů může zhoršit výkon

DEKORÁTOR

- **Adapter**
  - Mění interface podle požadavků klienta
- **State**
  - Vzor lze implementovat pomocí decorátorů (místo objektů podtřídy zapouzdřujících měnící funkce)
- **Chain of Responsibility**
  - Podobná struktura s nezávislými operacemi
- **Facade**
  - Nepřidává nové funkce
  - Lze s decorátorem kombinovat