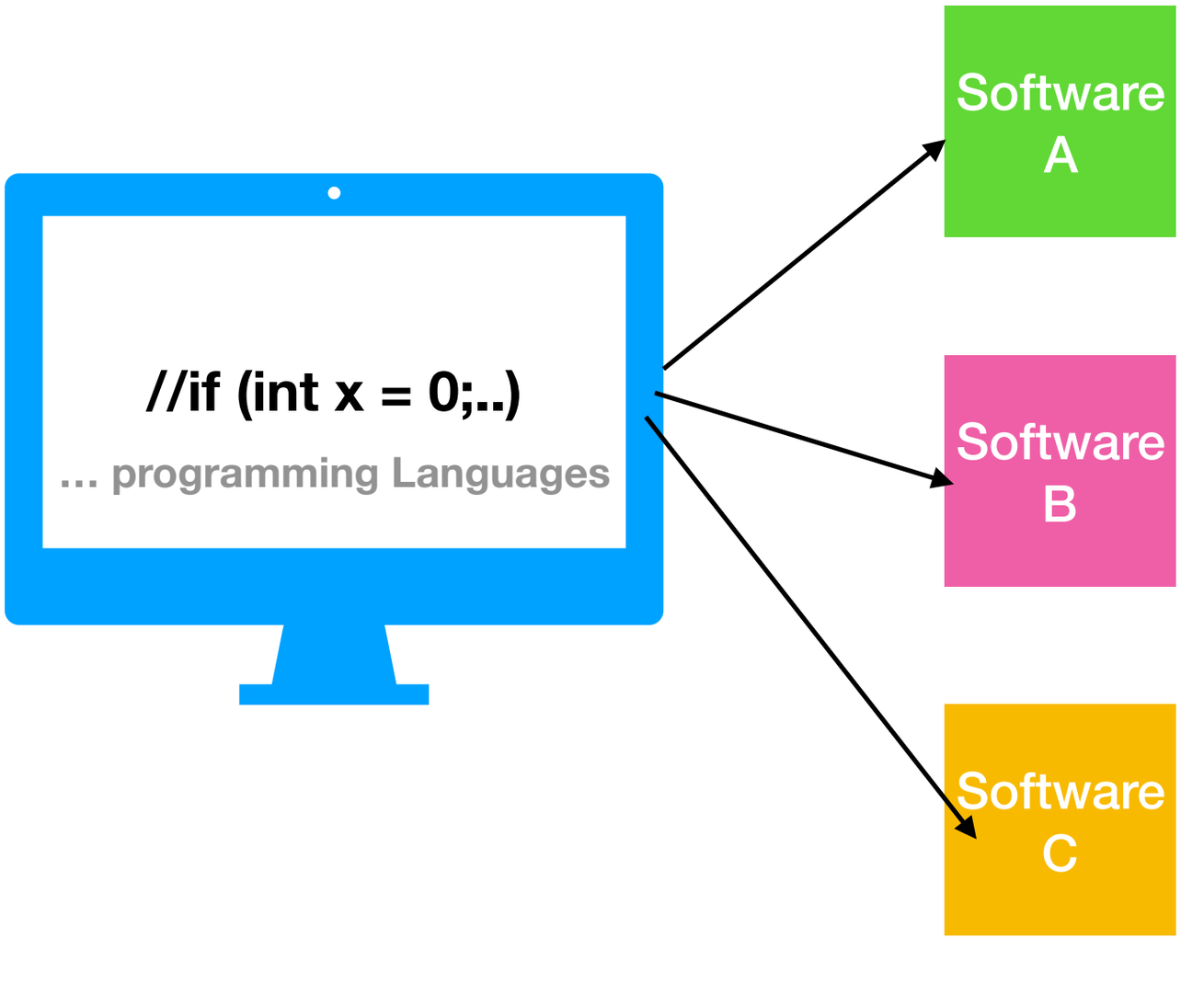
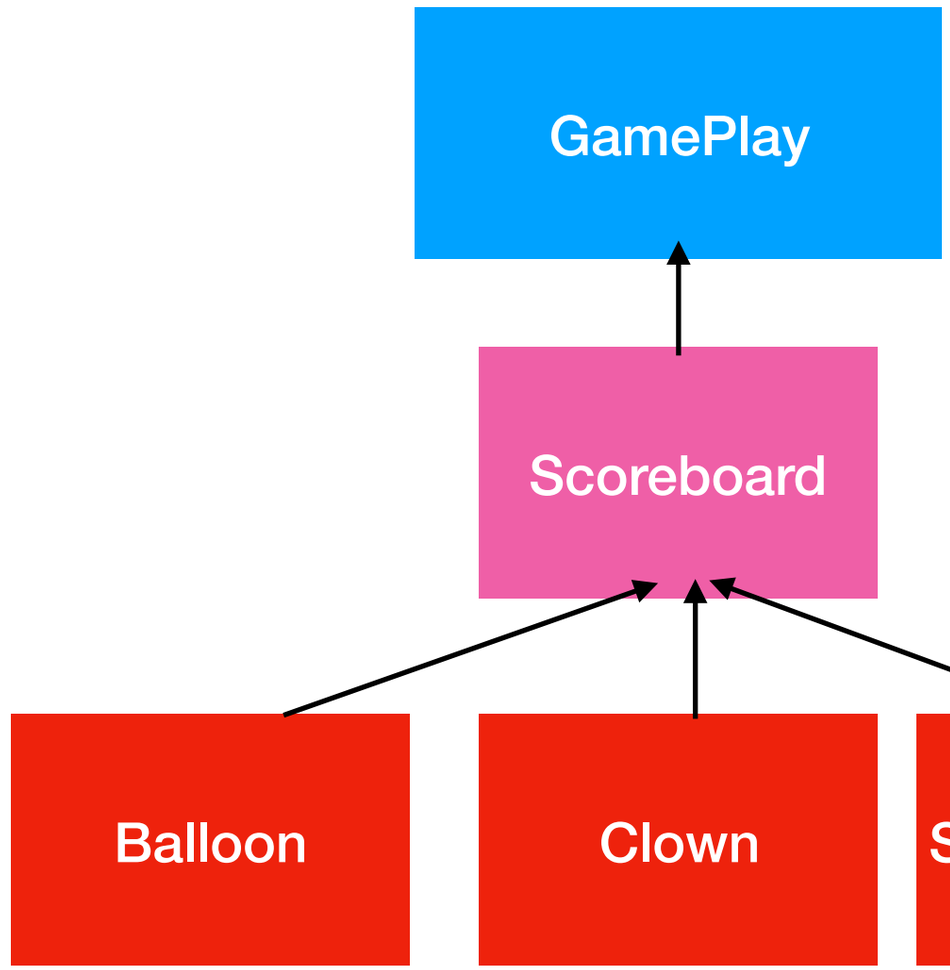
Design Patterns - Introduction

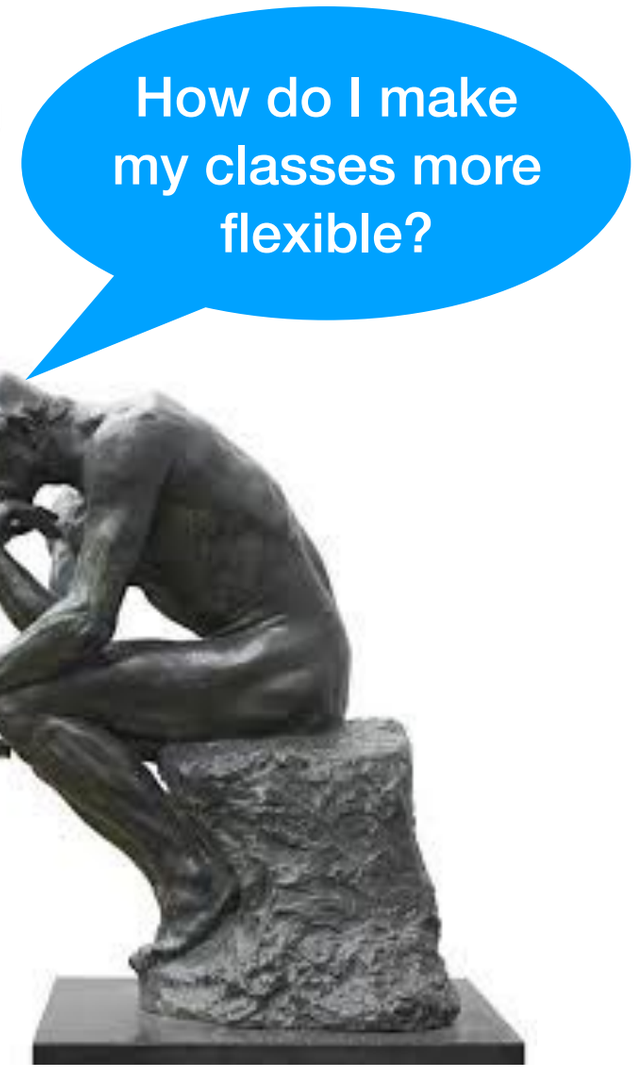
The Why


//if (int x = 0;..)
... programming Languages
Software A
Software B
Software C

“Way of Thinking”

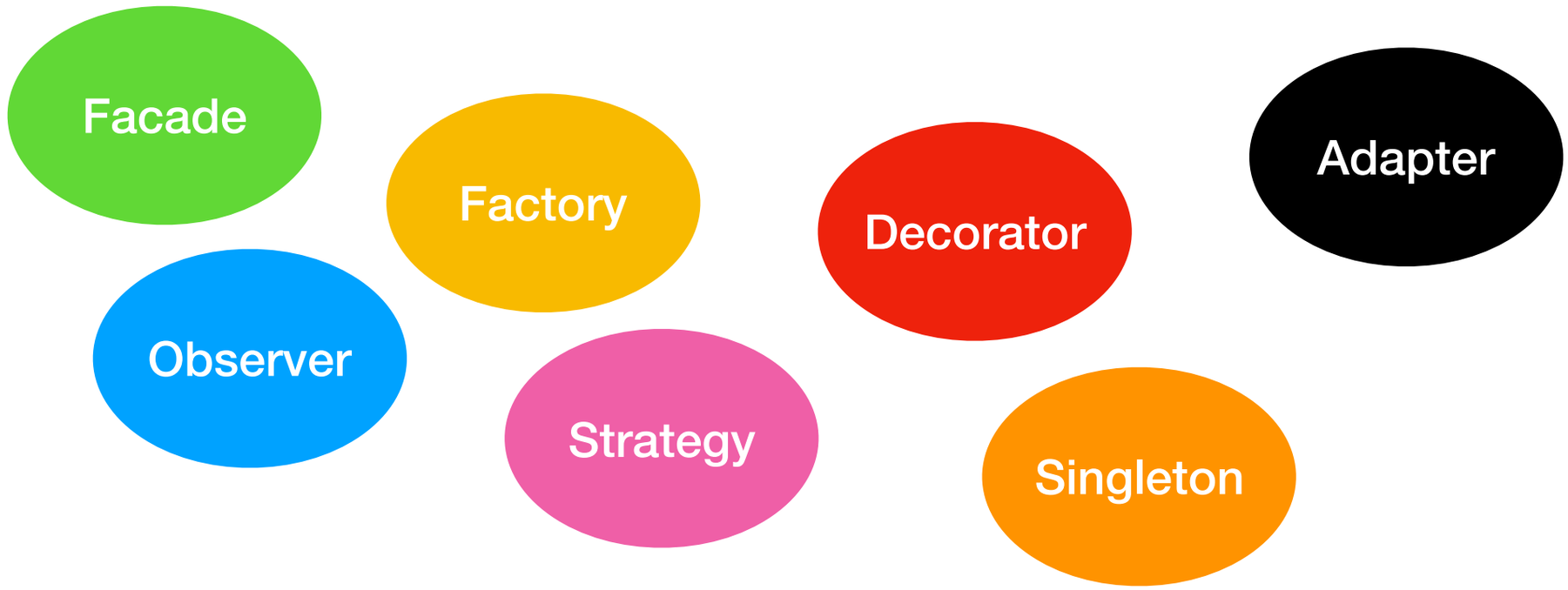
About Software Design (Code)





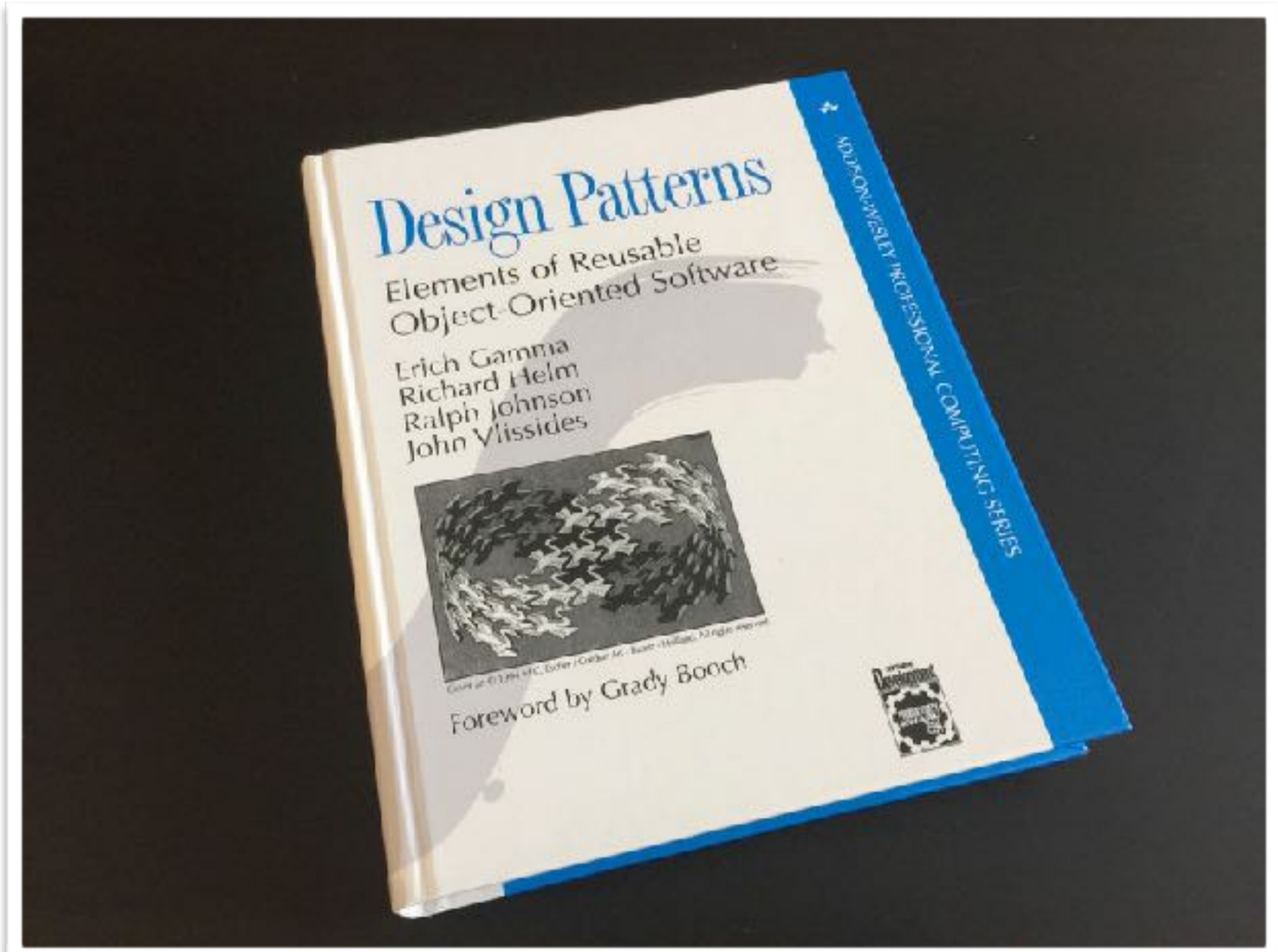
Design Patterns

Proven and Tested “ways of thinking” about Code Design



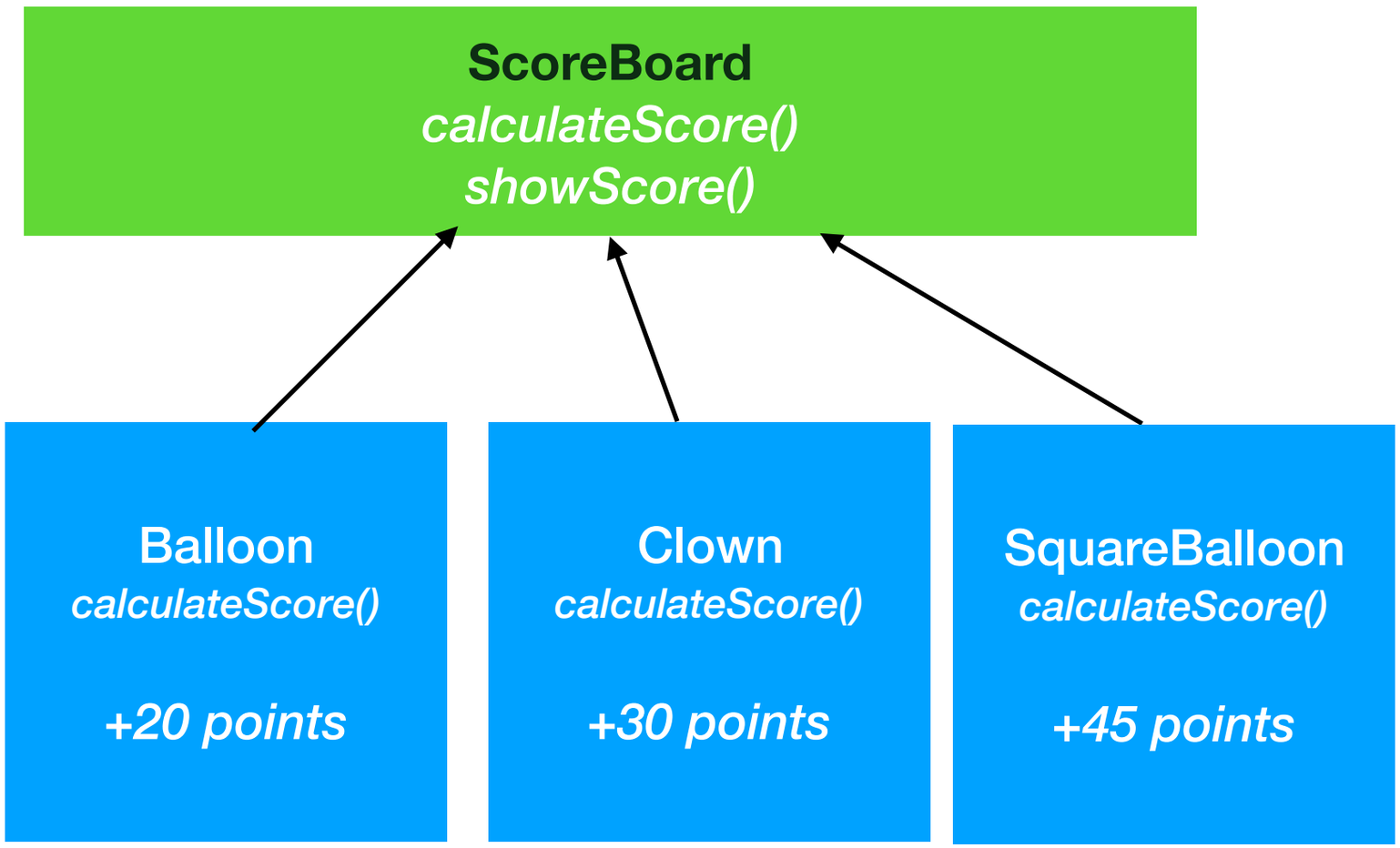
The Gang of Four

The Creators of Design Patterns

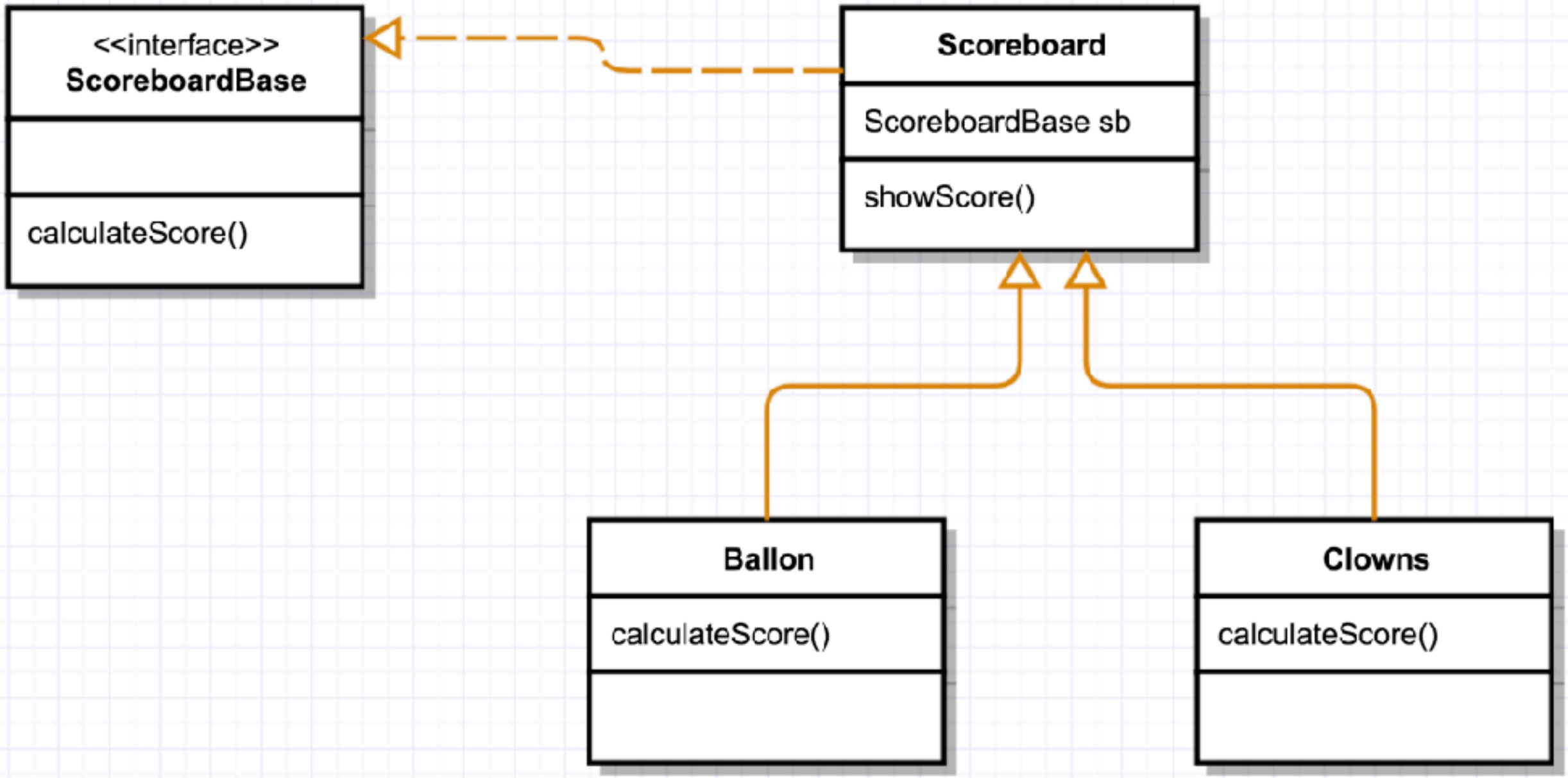


The Strategy Design Pattern

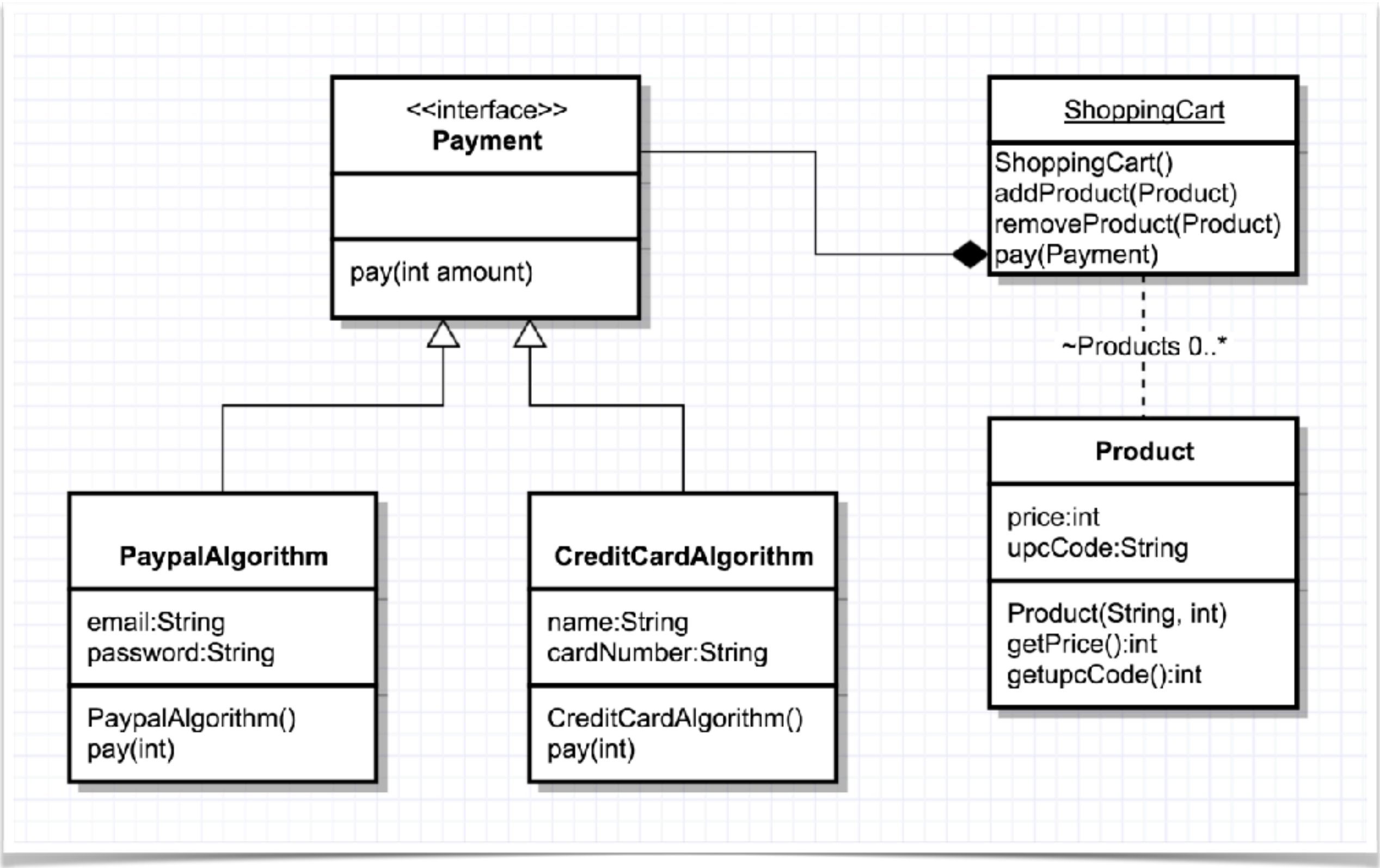
Algorithms for Different Cases…



Class Hierarchy

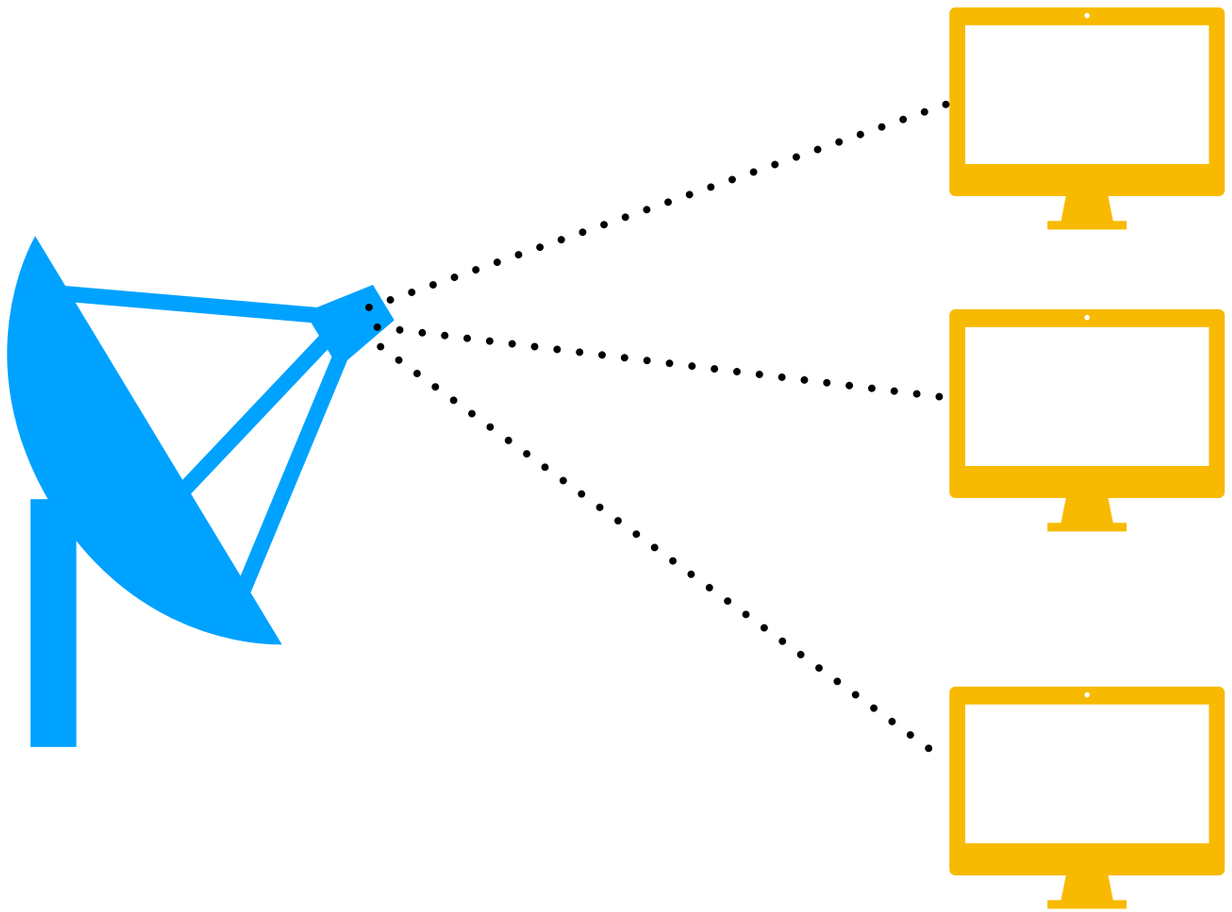


Example # 2


<<interface>>
Payment
pay(int amount)
ShoppingCart
ShoppingCart()
addProduct(Product)
removeProduct(Product)
pay(Payment)
~Products 0..*
Product
price:int
upcCode:String
Product(String, int)
getPrice():int
getupcCode():int
PaypalAlgorithm
email:String
password:String
PaypalAlgorithm()
pay(int)
CreditCardAlgorithm
name:String
cardNumber:String
CreditCardAlgorithm()
pay(int)

The Observer Design Pattern

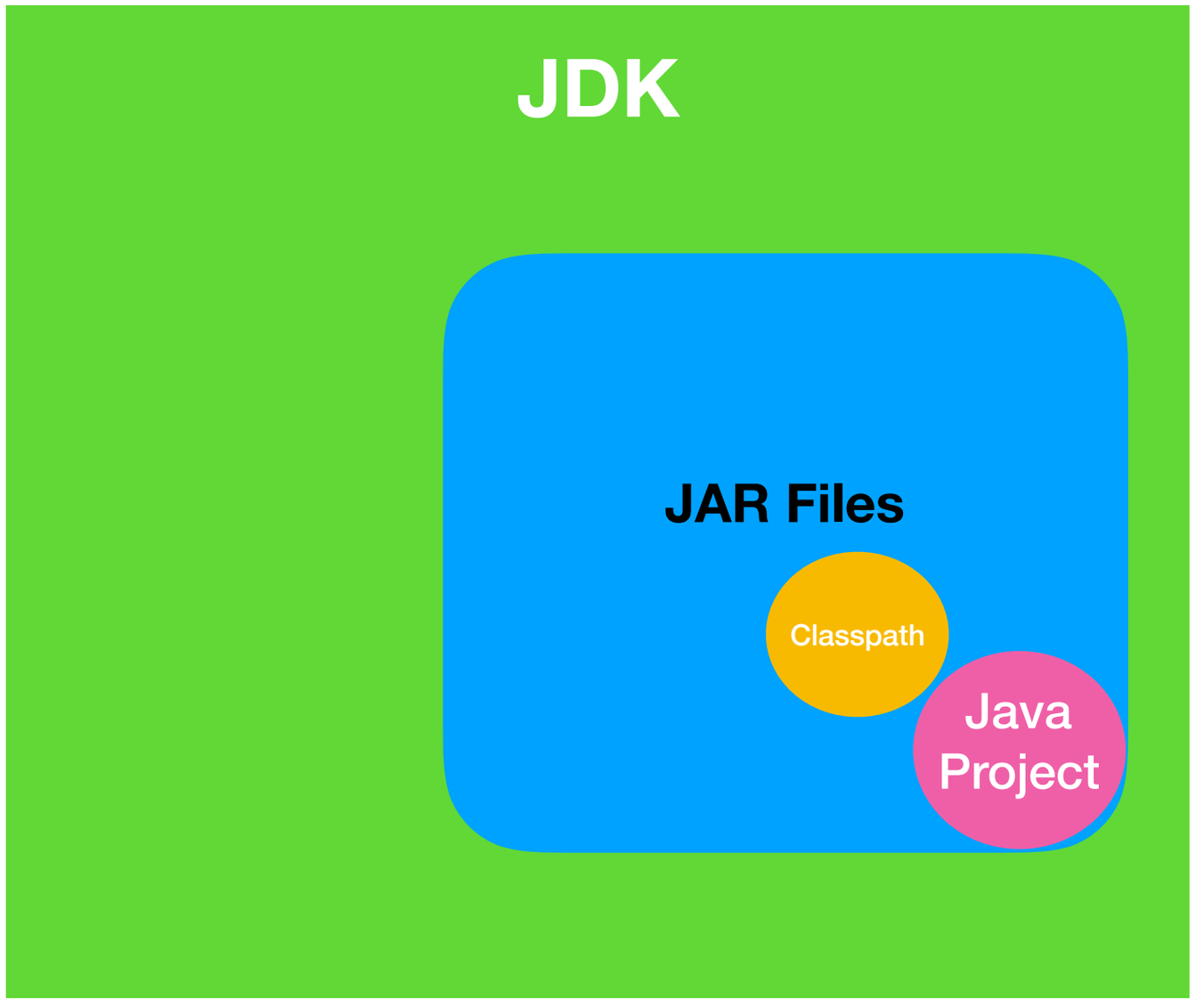
Observer Design Pattern



… defines a *1-to-many* dependency between objects so that when one object *changes state*, all of its dependents are *notified and updated automatically*

Observer Pattern Example

Problem with Current State…

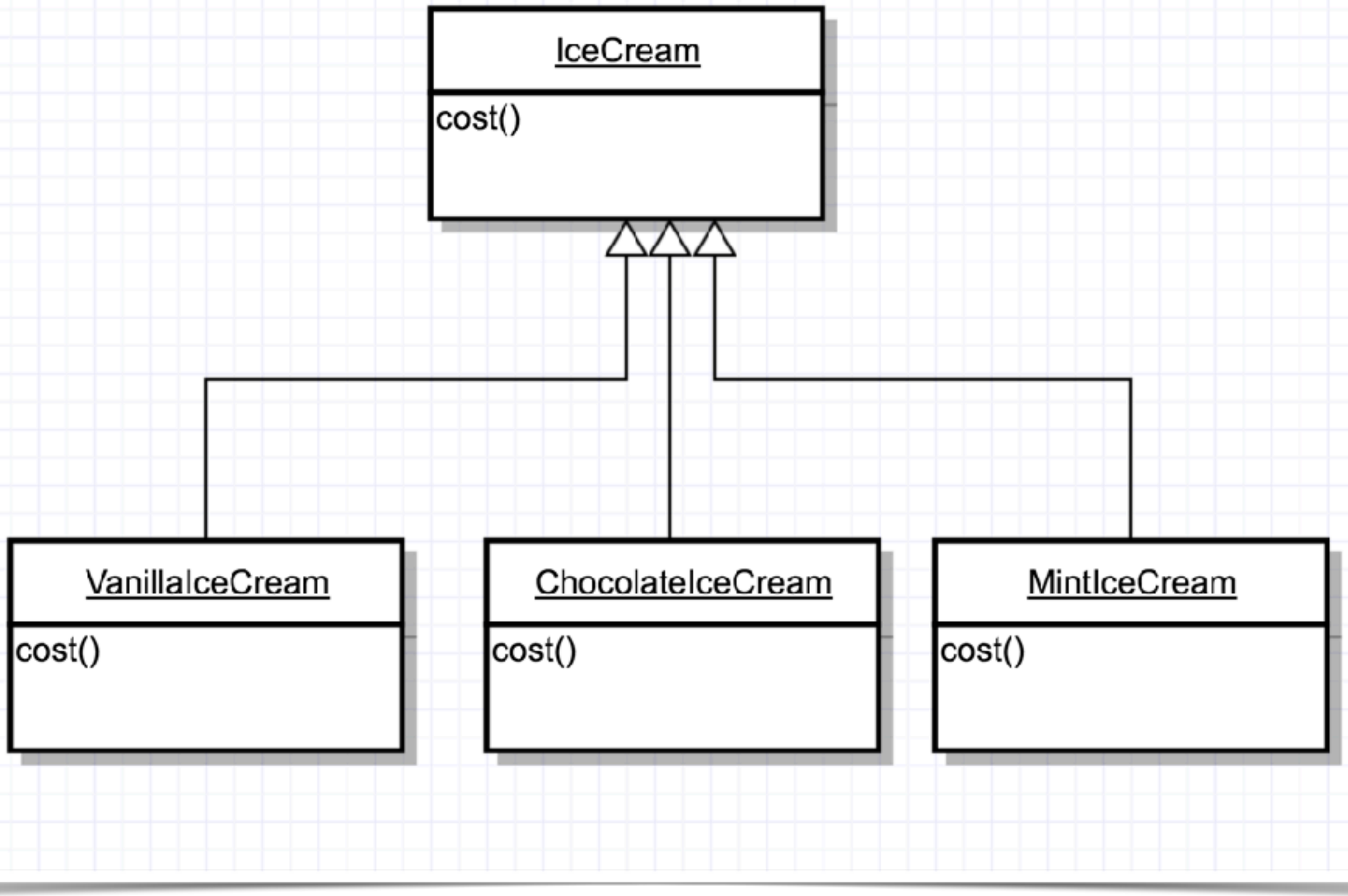


Project grows - more JAR Files

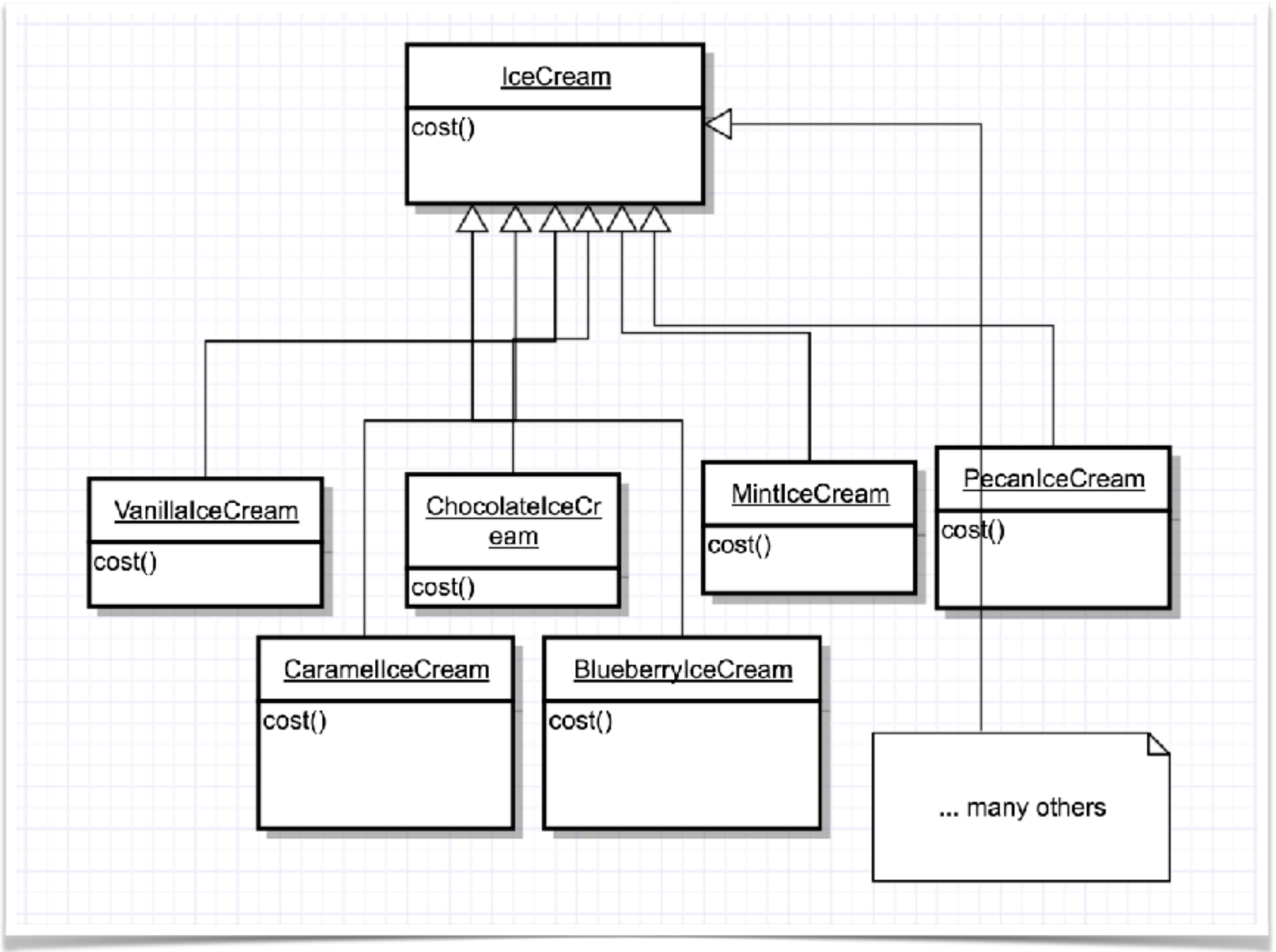
More JAR files - more class path issues

The Decorator Pattern

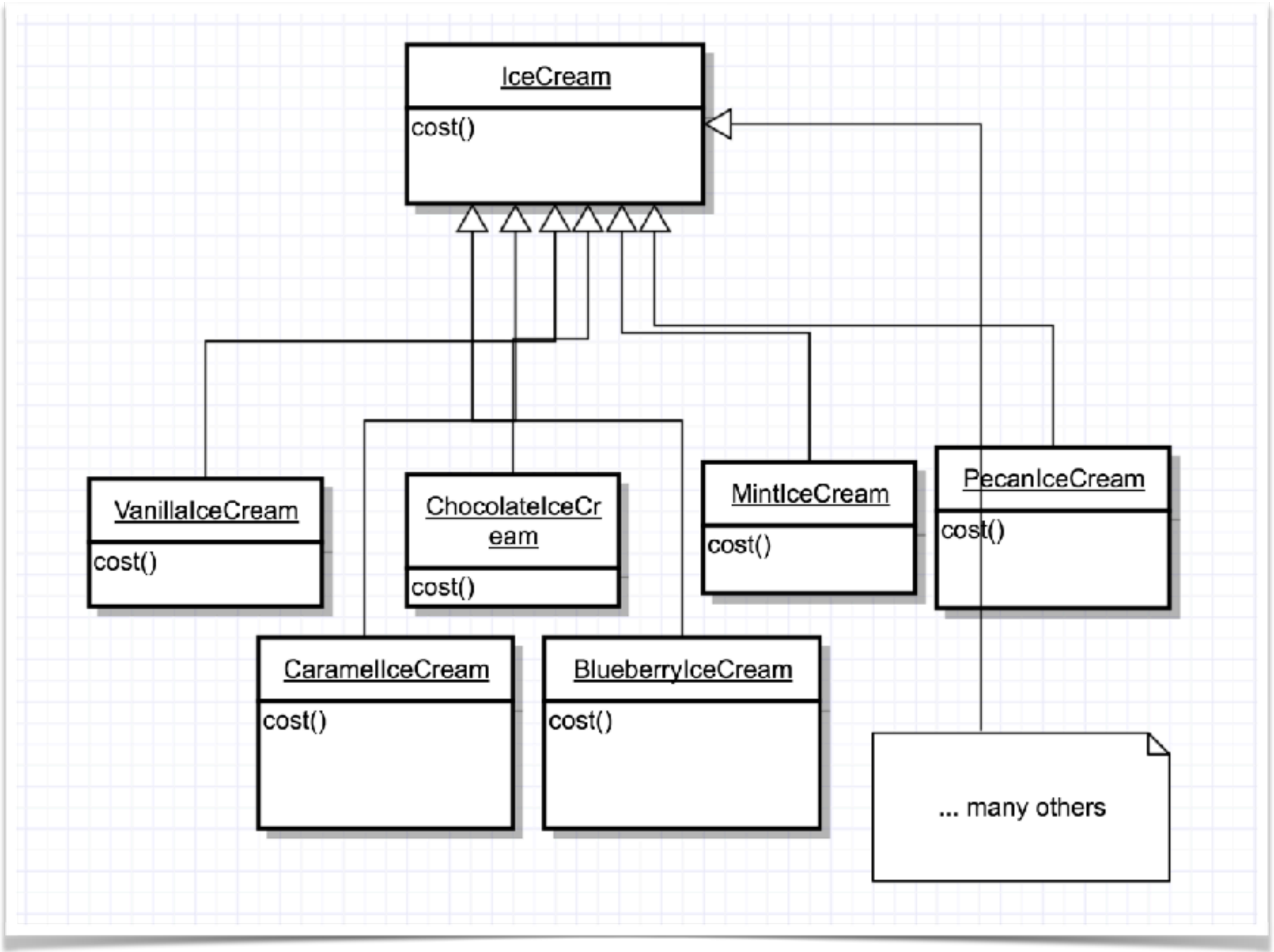
Ice-cream Shop



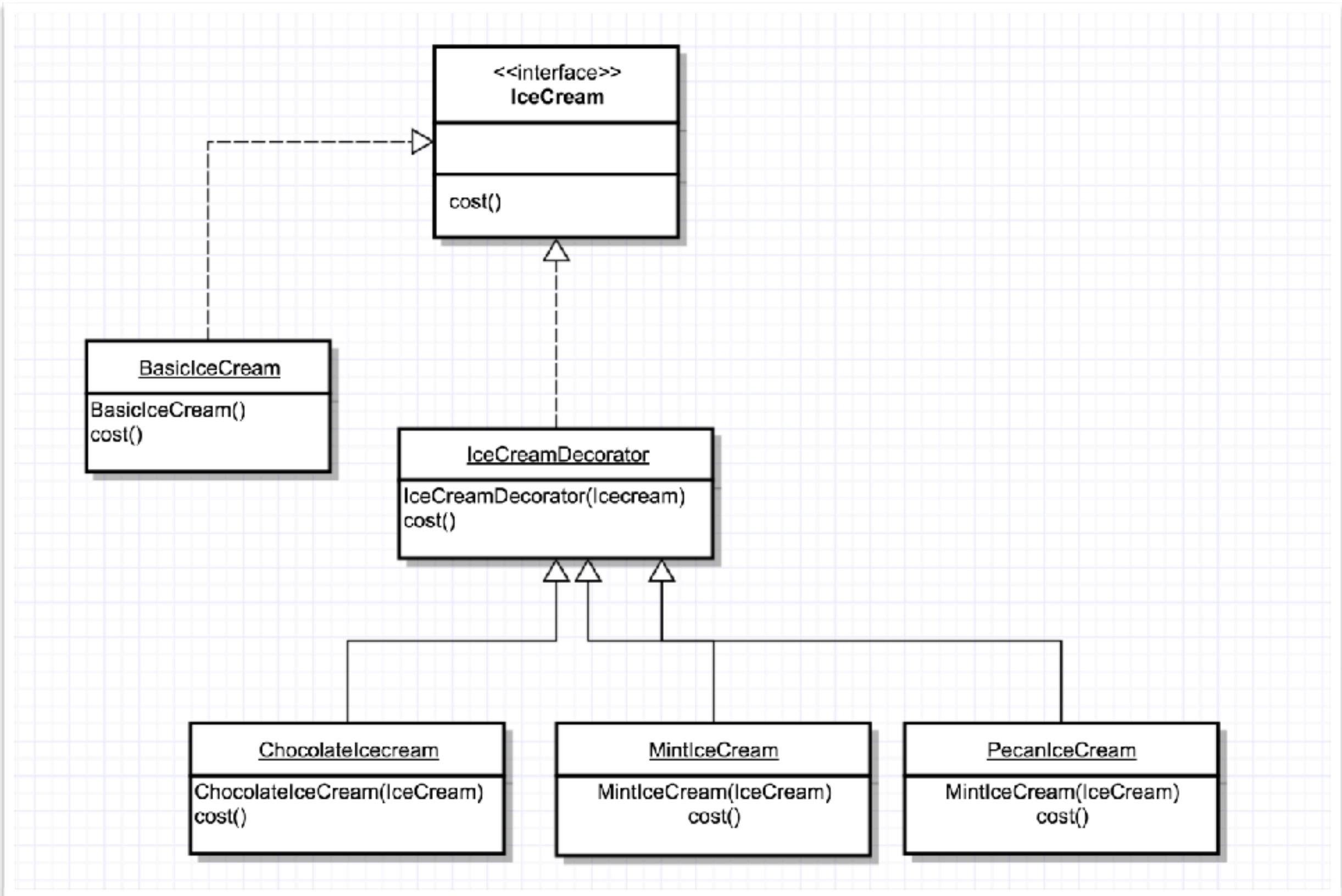
Ice-cream Shop



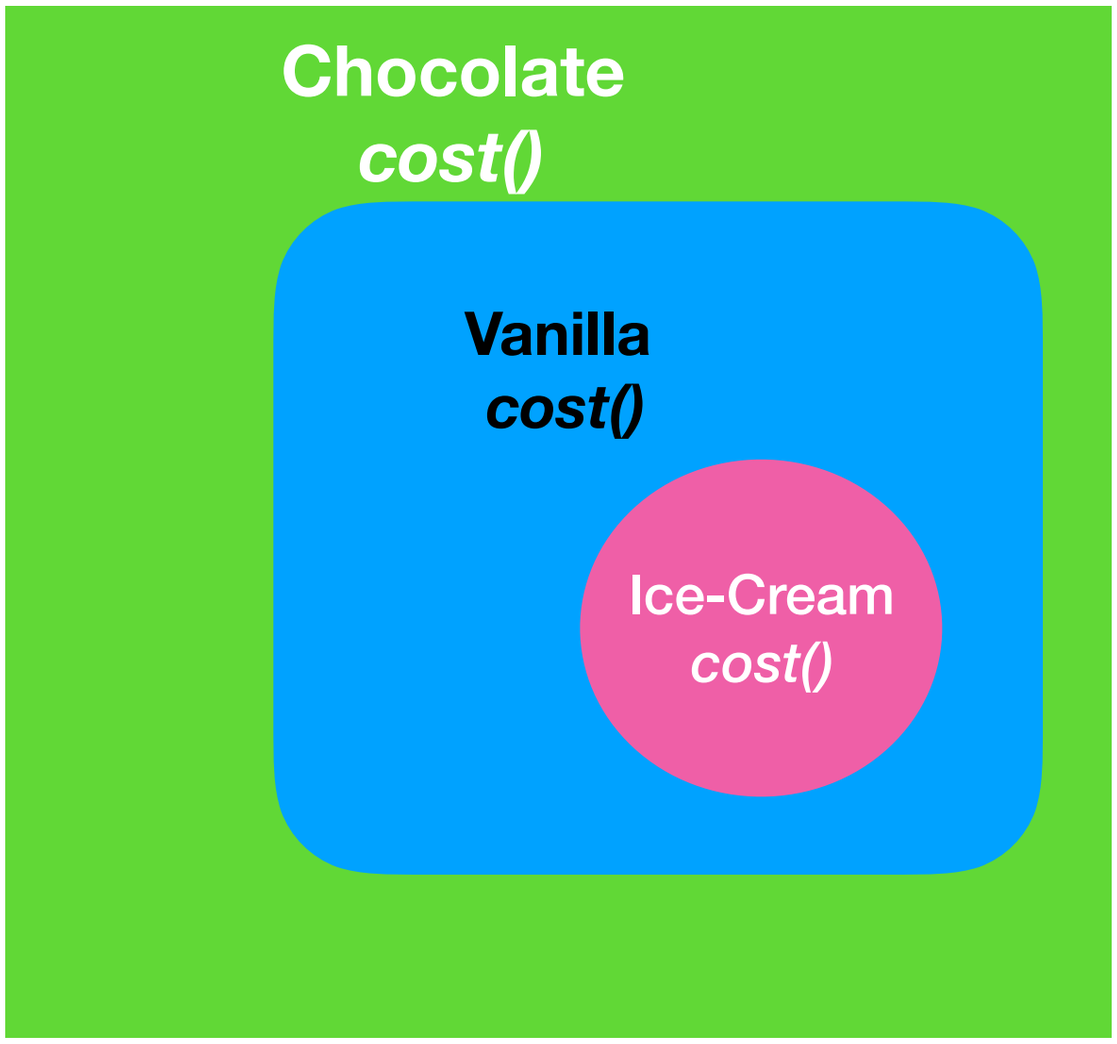
Problem…


IceCream
cost()
VanillaIceCream
cost()
ChocolateIceCr
eam
cost()
MintIceCream
cost()
PecanIceCream
cost()
CaramelIceCream
cost()
BlueberryIceCream
cost()
... many others

Solution


<<interface>>
IceCream
cost()
BasicIceCream
BasicIceCream()
cost()
IceCreamDecorator
IceCreamDecorator(Icecream)
cost()
ChocolateIcecream
ChocolateIceCream(IceCream)
cost()
MintIceCream
MintIceCream(IceCream)
cost()
PecanIceCream
MintIceCream(IceCream)
cost()

The Result…



cost() will give us the final total cost

We wrap each ice-cream type with other toppings or types

The Factory Pattern*

**Introducing the Simple Factory "Pattern"*

HamburgerStore

```
class Hamburger {
      void prepare() { }
      void cook() { }
      void box(){ }
}
```

```
Hamburger orderHamburger() {

    Hamburger burger = new Hamburger();
    burger.prepare();
    burger.cook();
    burger.box();

    return burger;
}
```

We need more than 1 type of burger…

```
Hamburger orderHamburger(String type) {

    Hamburger burger = new Hamburger();

    //We add types
    if (type.equals("cheese")){
        burger = new CheeseBurger();
    }else if (type.equals("greek")){
        burger = new GreekBurger();
    }else if (type.equals("meatLover")){
        burger = new MeatLover();
    }

    burger.prepare();
    burger.cook();
    burger.box();

    return burger;
}
```

```
else if (type.equals("veggie")){
    burger = new VeggieBurger();
else if (type.equals("bunLess")){
    burger = new BunLessBurger();
```

Solution - Simple Factory

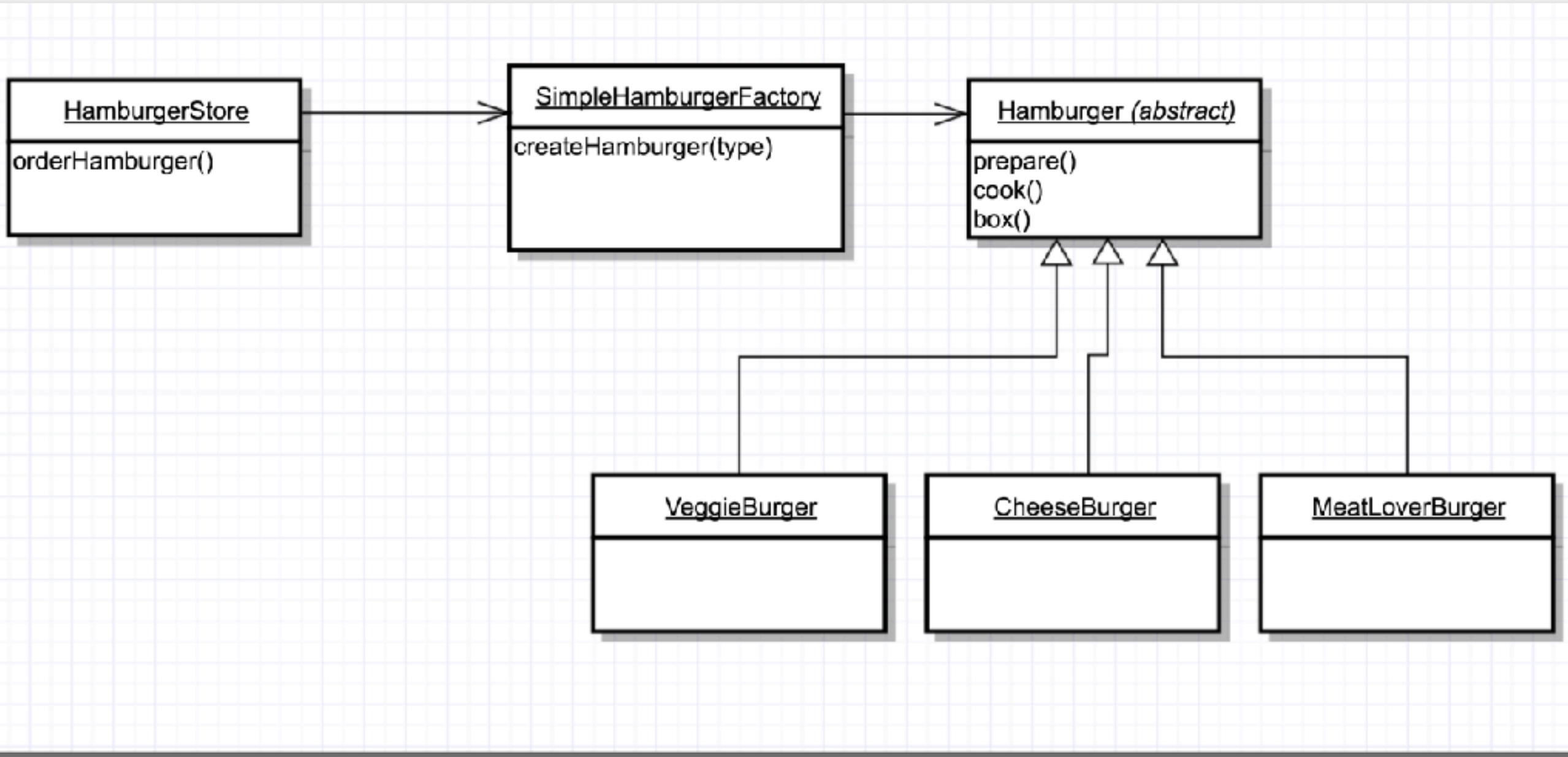
```
class SimpleHamburgerFactory {

    public Hamburger createHamburger(String type) {
        Hamburger burger = null;

        //We add types
        if (type.equals("cheese")){
            burger = new CheeseBurger();
        }else if (type.equals("greek")){
            burger = new GreekBurger();
        }else if (type.equals("meatLover")){
            burger = new MeatLover();
        } else if (type.equals("veggie")){
            burger = new VeggieBurger();
        }else if (type.equals("bunLess")){
            burger = new BunLessBurger();
        }

        return burger;

    }
```

```
Hamburger orderHamburger(String type) {

    Hamburger burger = new Hamburger();

    //We add types
    if (type.equals("cheese")){
        burger = new CheeseBurger();
    }else if (type.equals("greek")){
        burger = new GreekBurger();
    }else if (type.equals("meatLover")){
        burger = new MeatLover();
    }

    burger.prepare();
    burger.cook();
    burger.box();

    return burger;
}
```

Problem solved… (For now)

```
class HamburgerStore {
    SimpleHamburgerFactory factory;

    public HamburgerStore(SimpleHamburgerFactory factory) {
        this.factory = factory;
    }

    Hamburger orderHamburger(String type) {

        Hamburger burger;

        //We now user our factory! Not the if statements.
        burger = factory.createHamburger(type);

        burger.prepare();
        burger.cook();
        burger.box();

        return burger;
    }
}
```

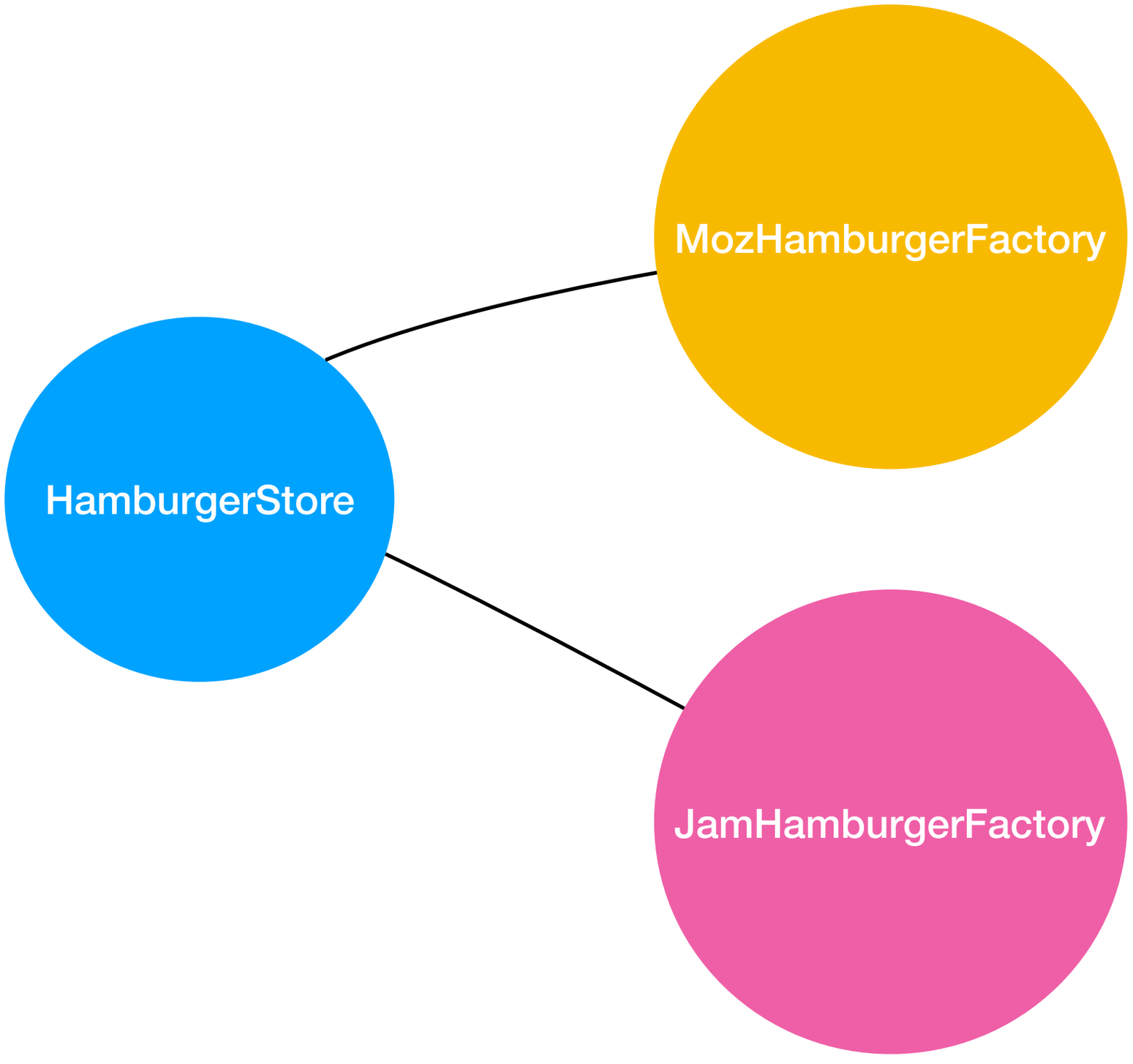
Solution - Class Diagram



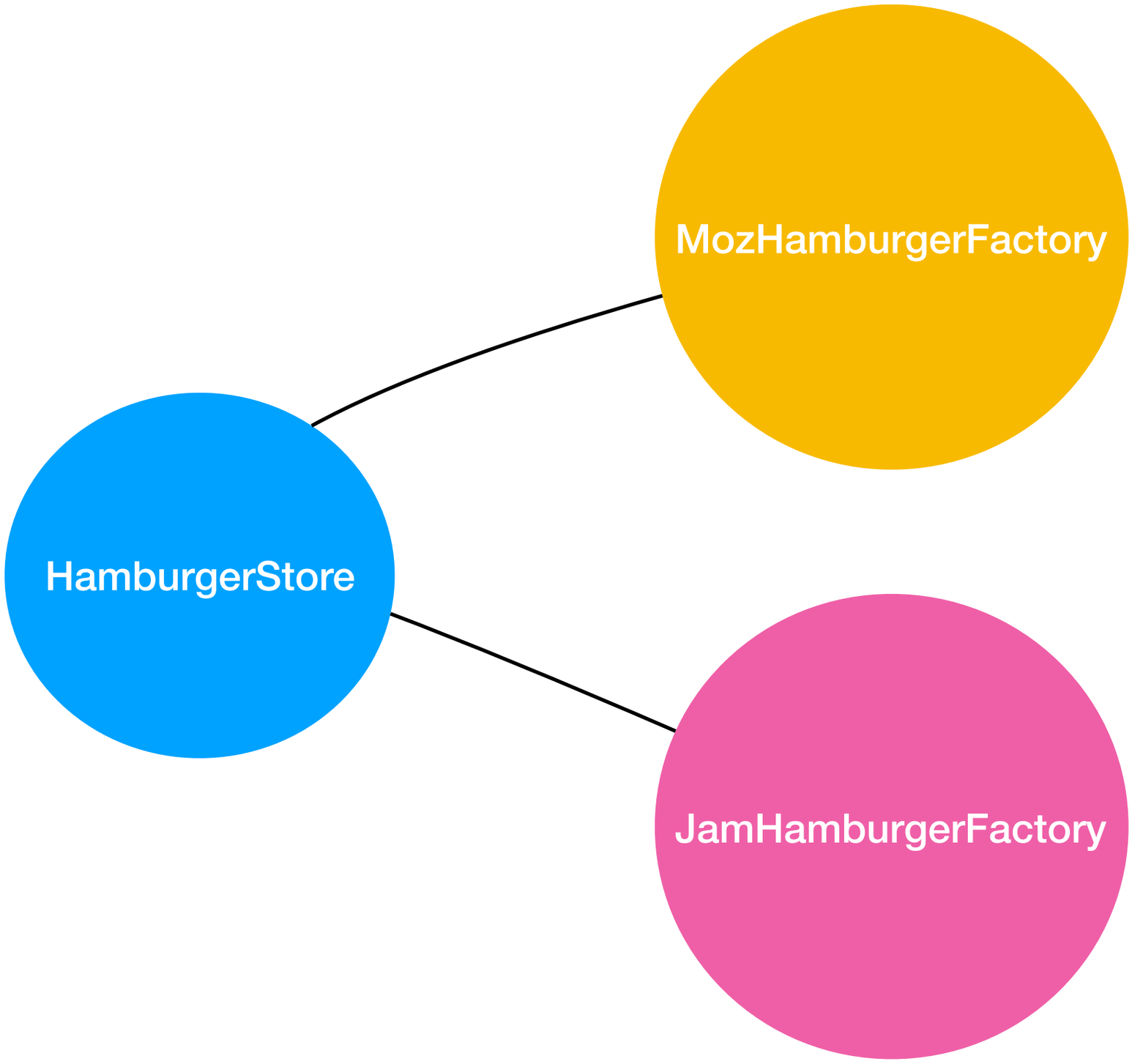
The Factory Pattern*

*The real thing!

Franchise our Store!



Create our Franchises…



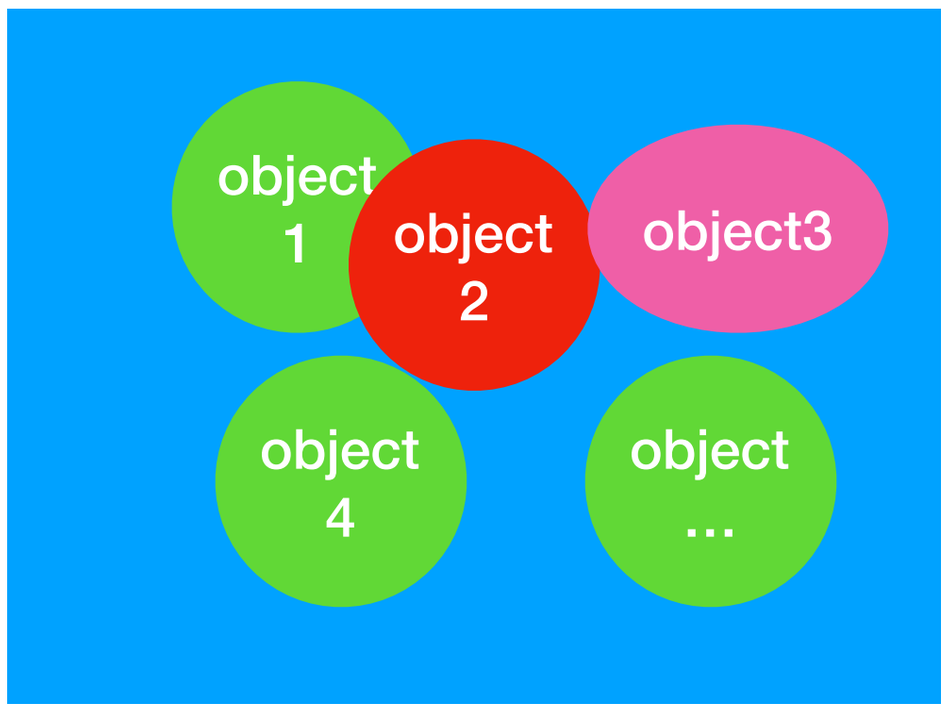

```
JamHamburgerFactory jFactory = new JamHamburgerFactory();
HamburgerStore jamaicanBurgerStore = new HamburgerStore(jFactory);
jamaicanBurgerStore.orderHamburger("Cheese");
```

The Singleton Pattern

Object Creation

CheeseBurger cheeseBurger = ***new*** CheeseBurger();

Calling CheeseBurger() ***public*** constructor



Motivation: Too Many Object…

Sometimes we only need ONE instance of an object.

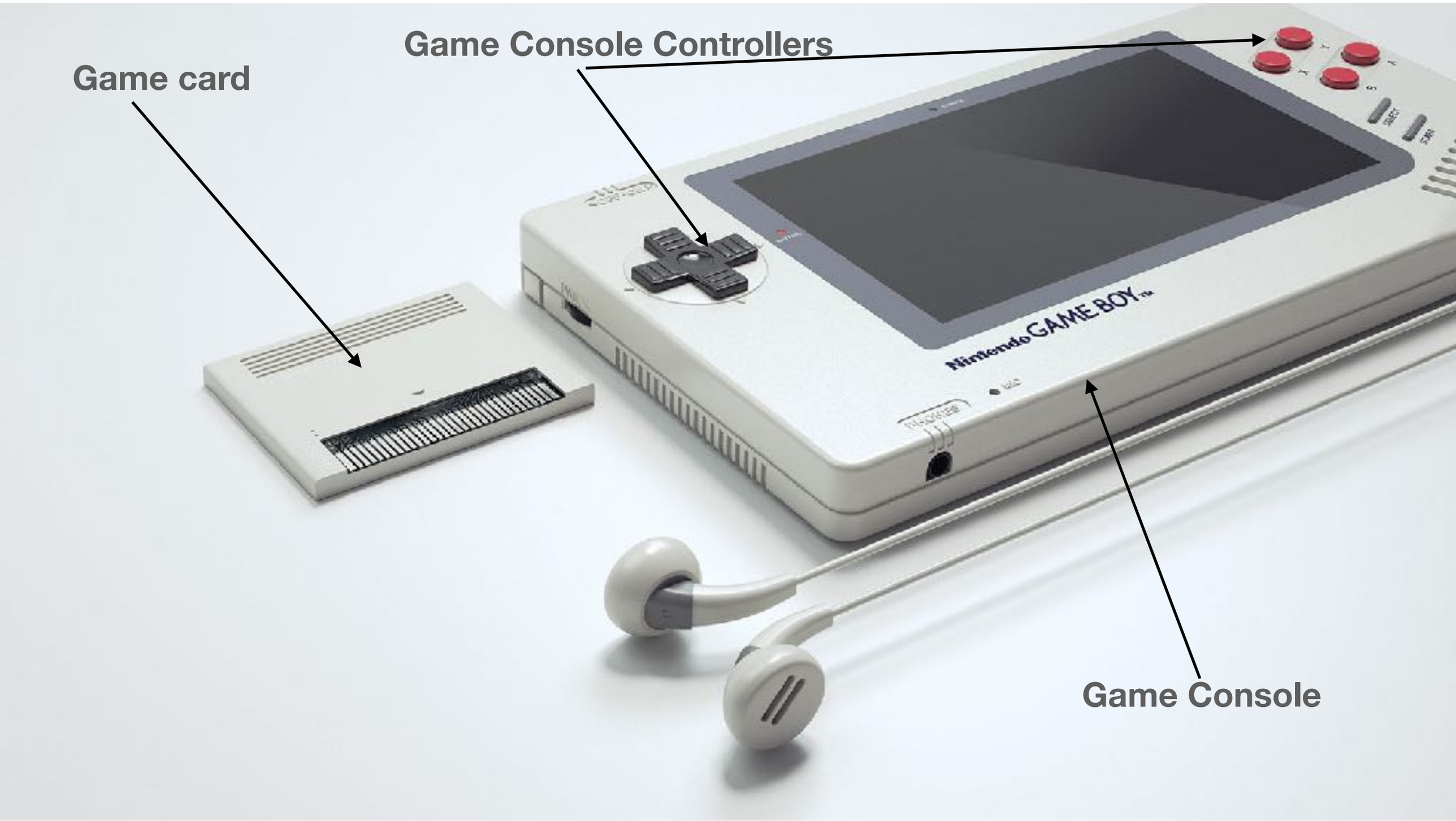
Solution

Singleton

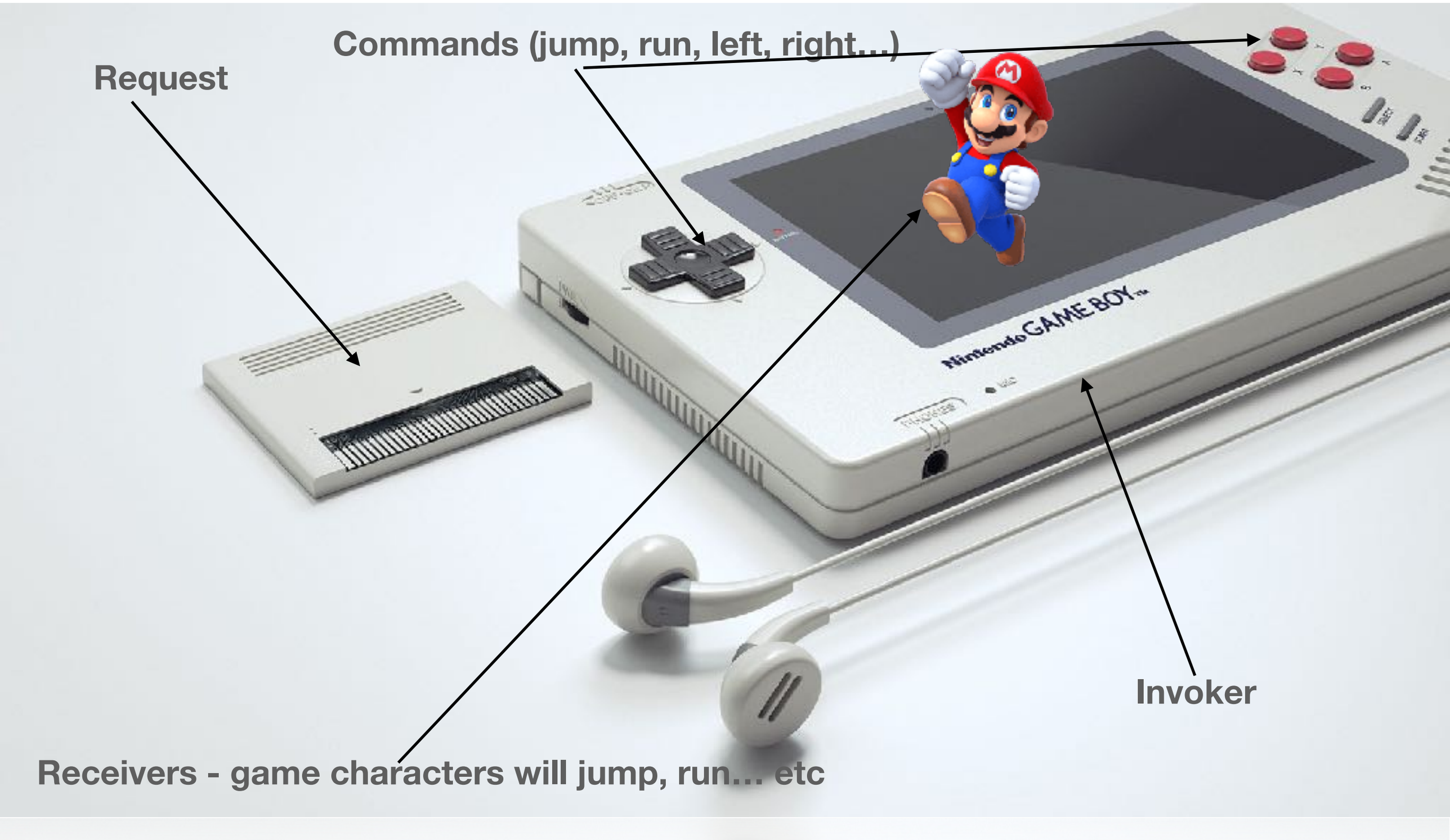
“Ensures a class has only one instance, and provide a global point of access to it.”

The Command Pattern

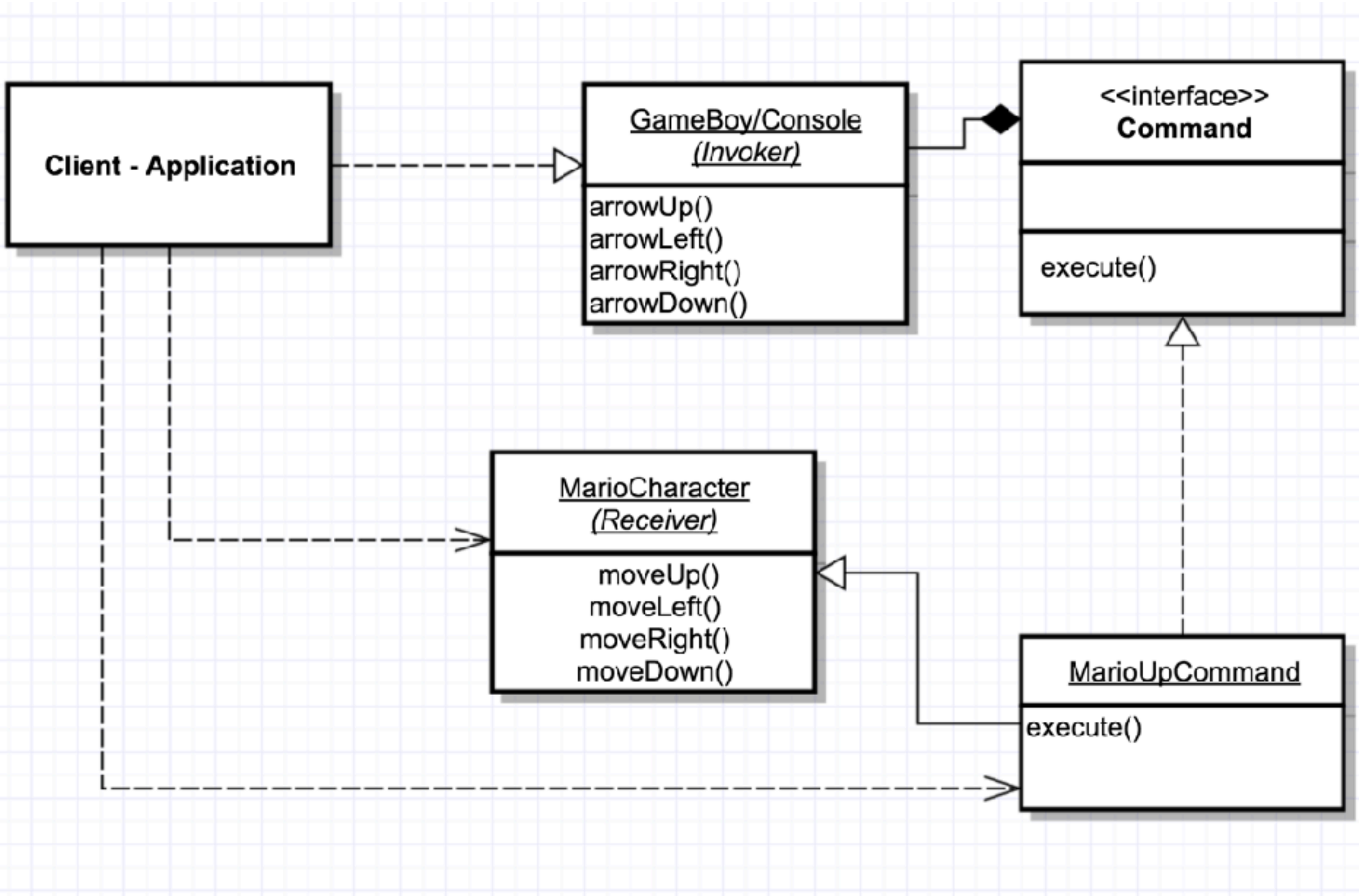
Deconstructing the System



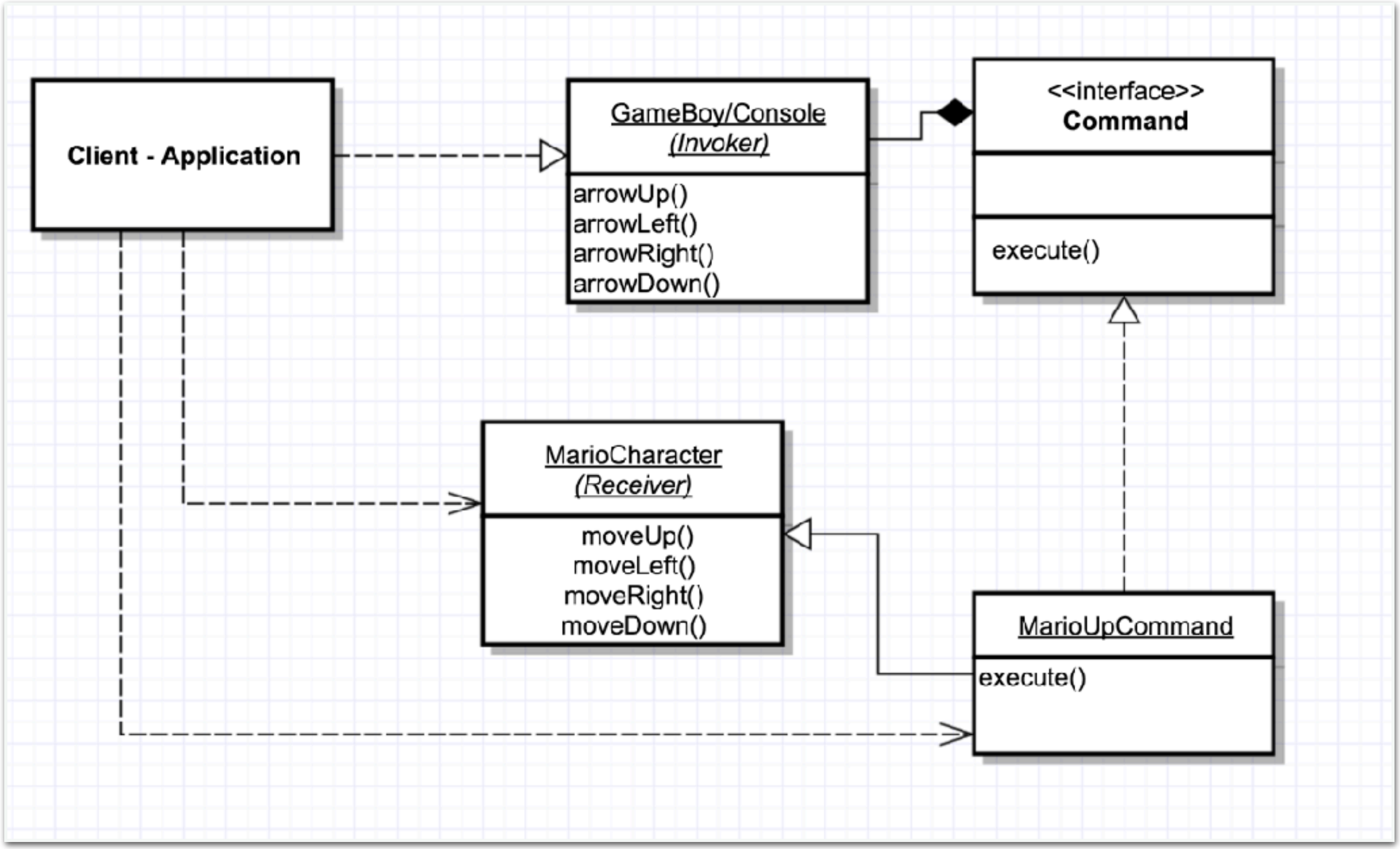
The Command Pattern System



The Command Pattern Diagram



Definition…

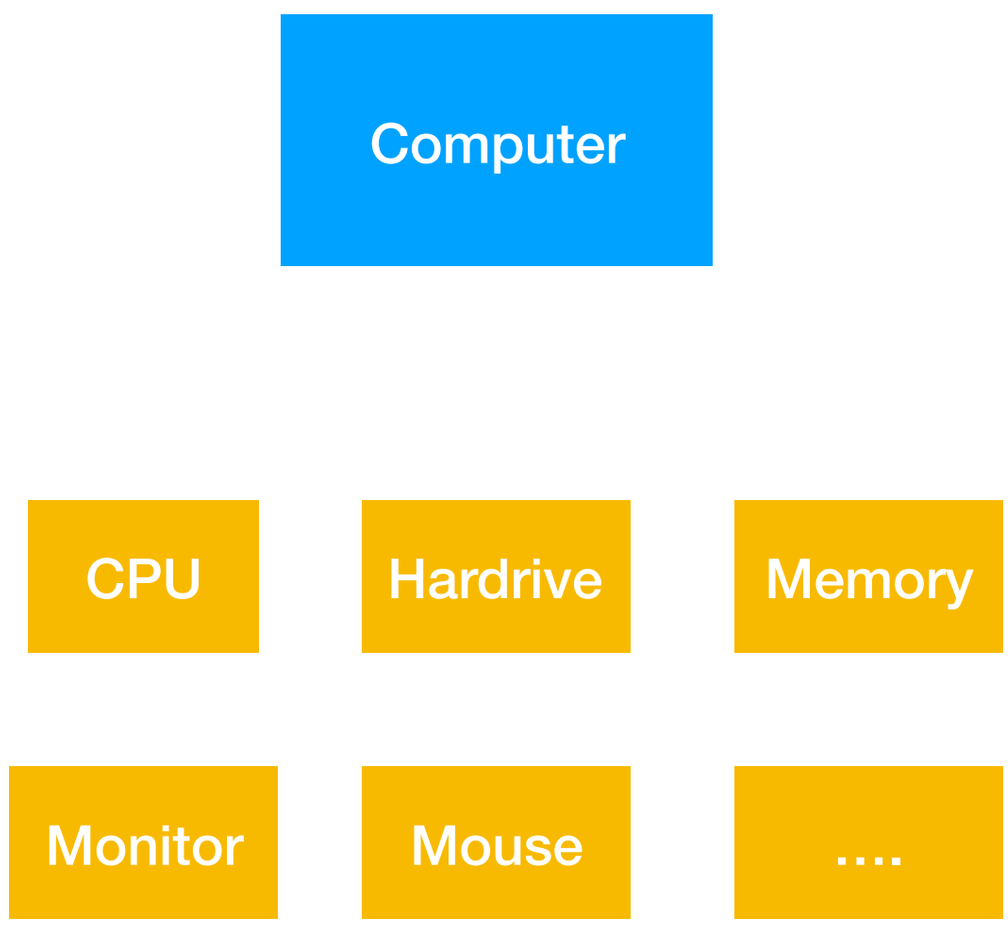


The Command Pattern:

encapsulates a request as an object (the game card), thereby letting you parameterize other objects with different requests, queues or log requests.

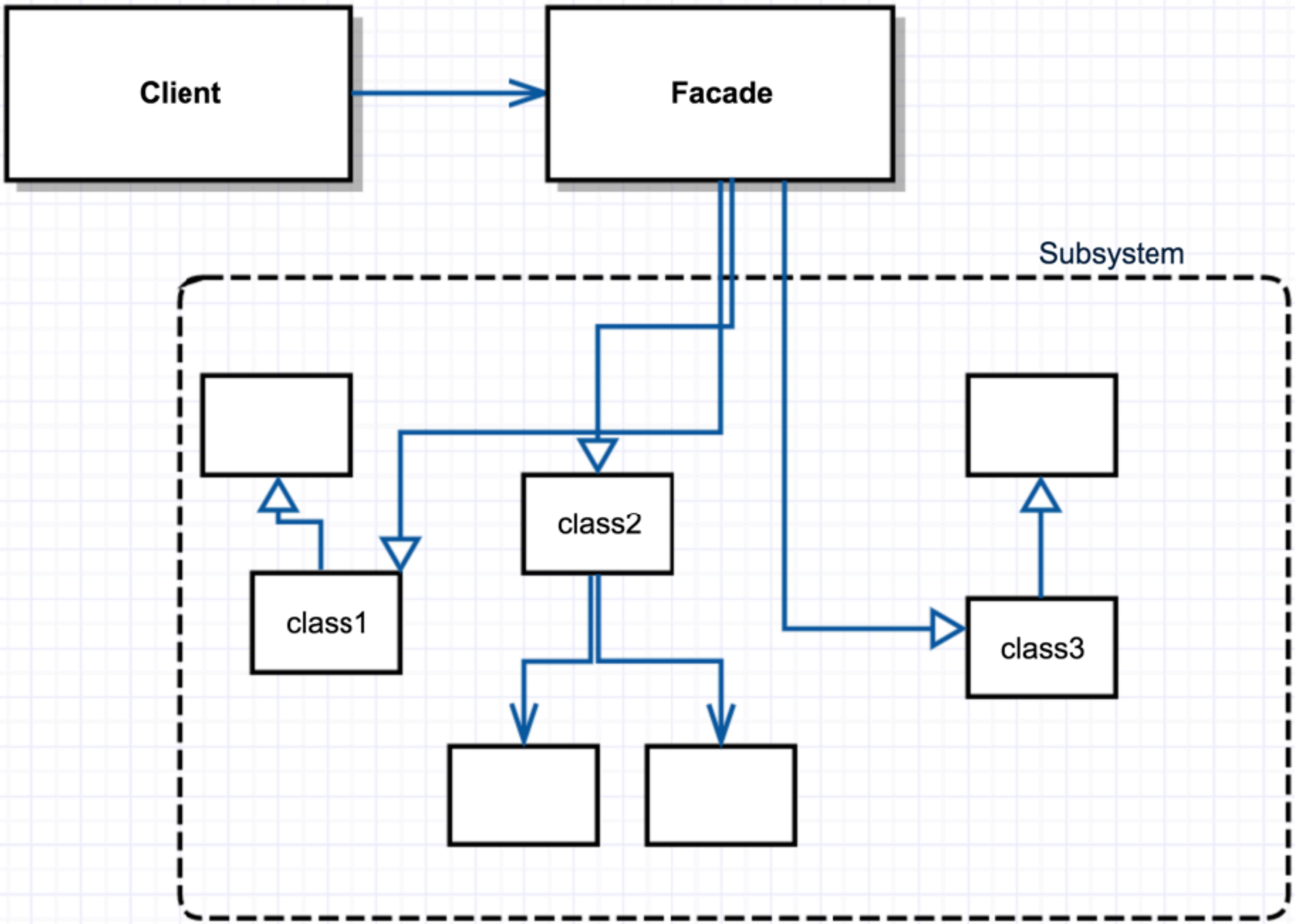
The Facade Pattern

Real-World Example

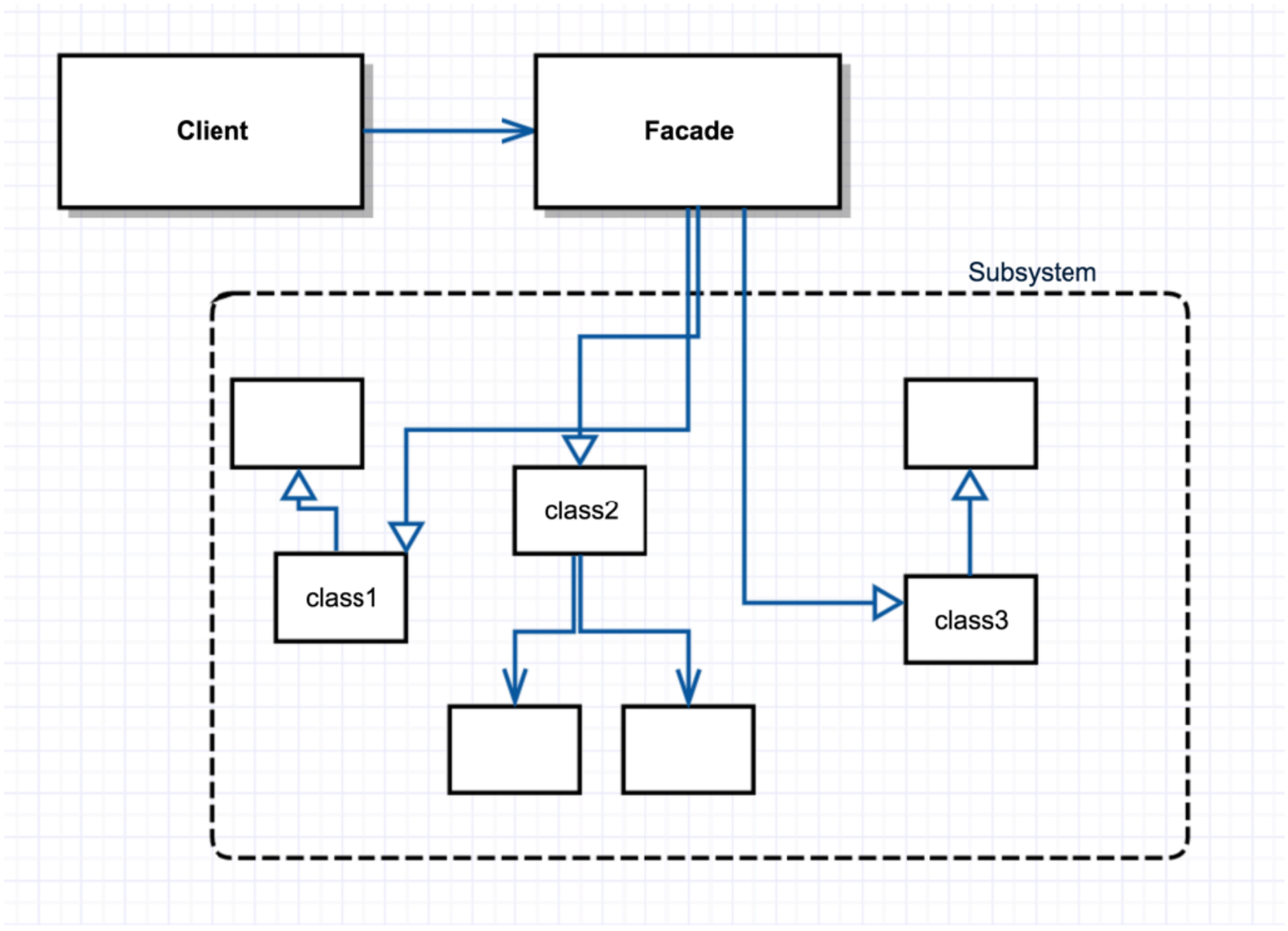


Too complex….

UML Facade Diagram



Definition…

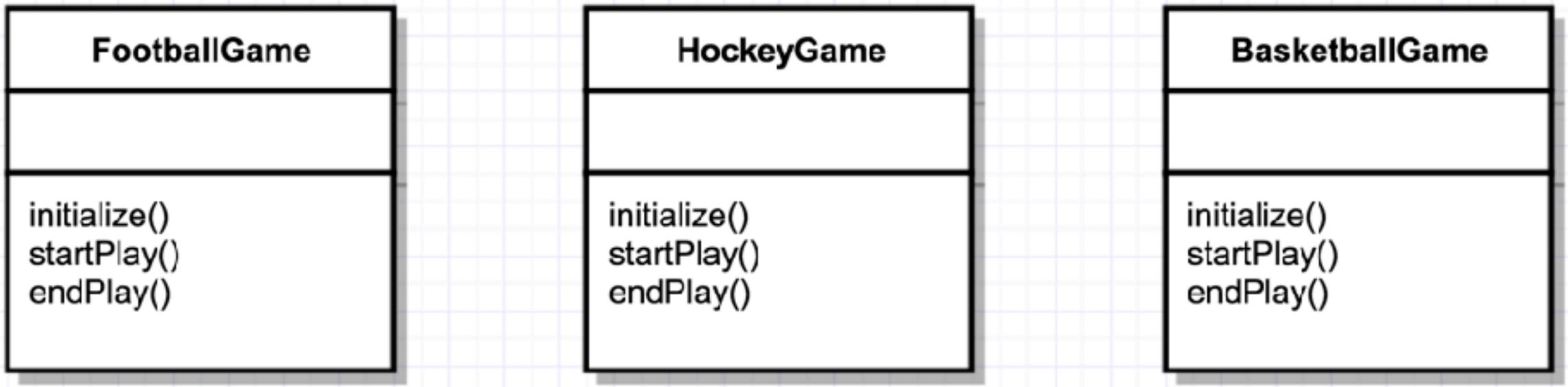


The Facade Pattern:

provides a unified interface
to a set of interfaces in a subsystem.
Facade defines a higher-level interface
that makes the subsystem easier to use.

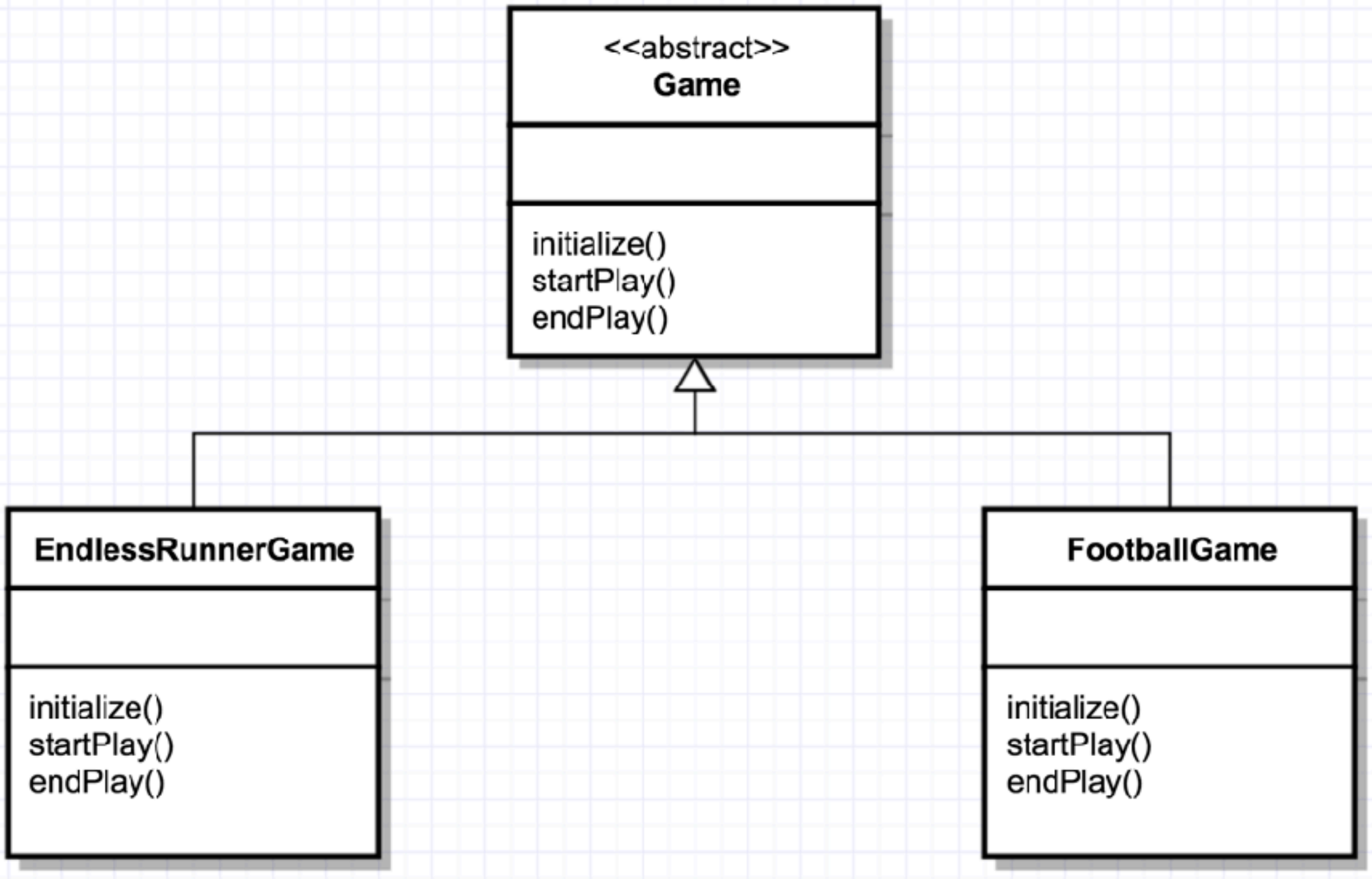
Facade - Definition

Motivation…



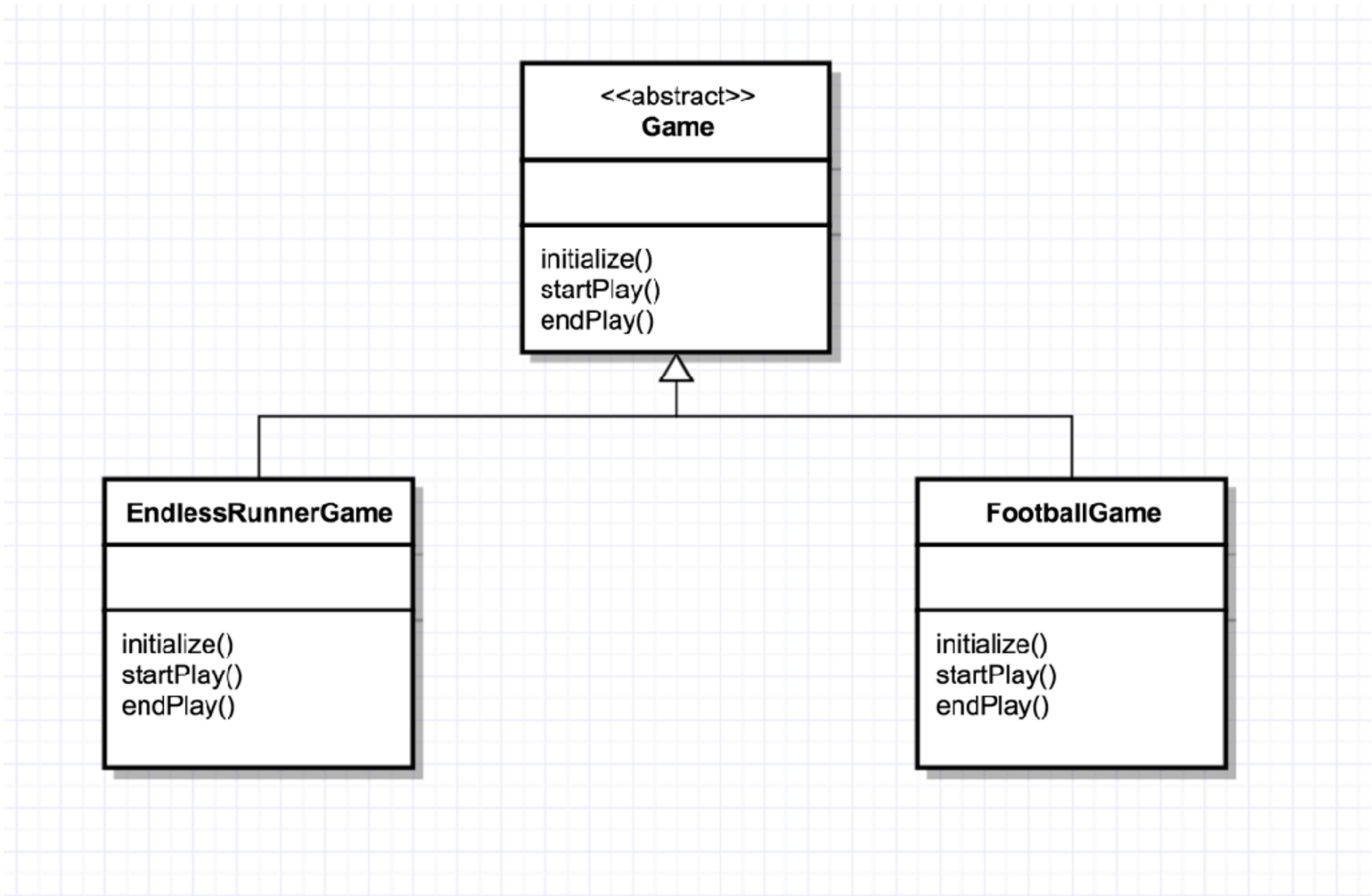
Redundant…..NOT Good!

Solution…


<<abstract>>
Game
initialize()
startPlay()
endPlay()
EndlessRunnerGame
initialize()
startPlay()
endPlay()
FootballGame
initialize()
startPlay()
endPlay()

The Template Method Pattern

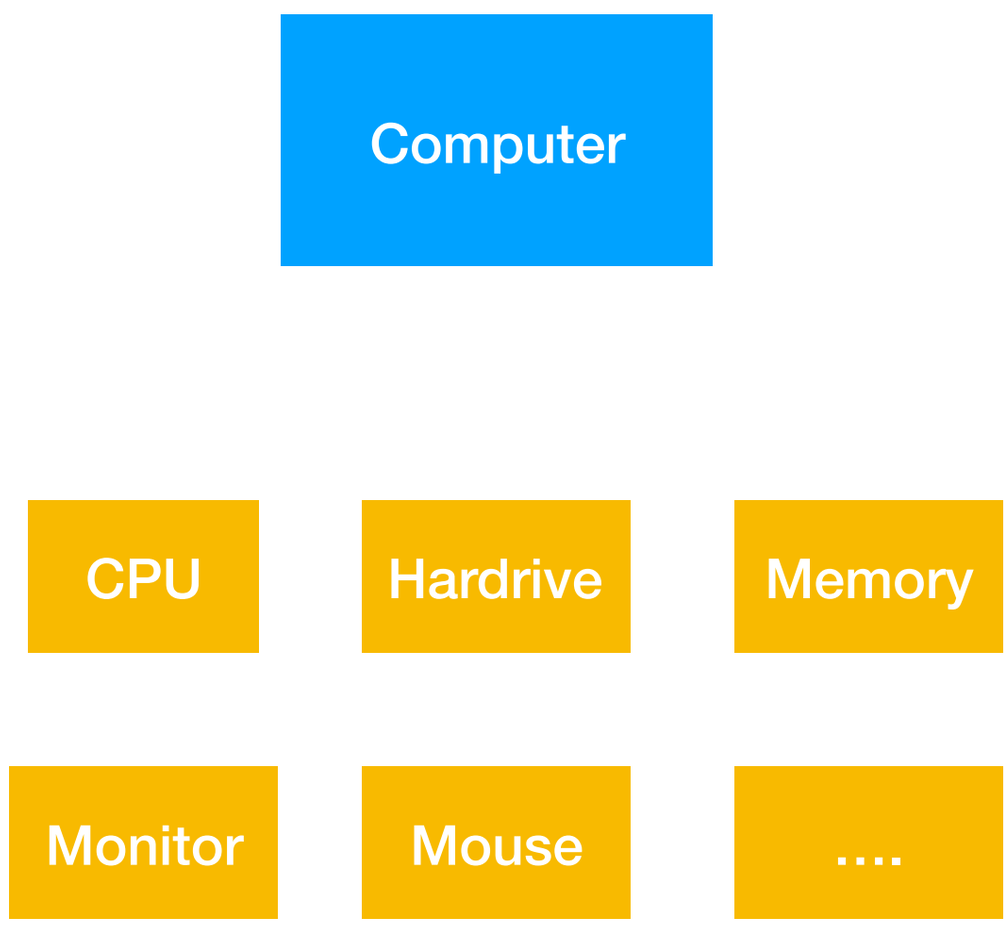
Definition…



The Template Method Pattern:

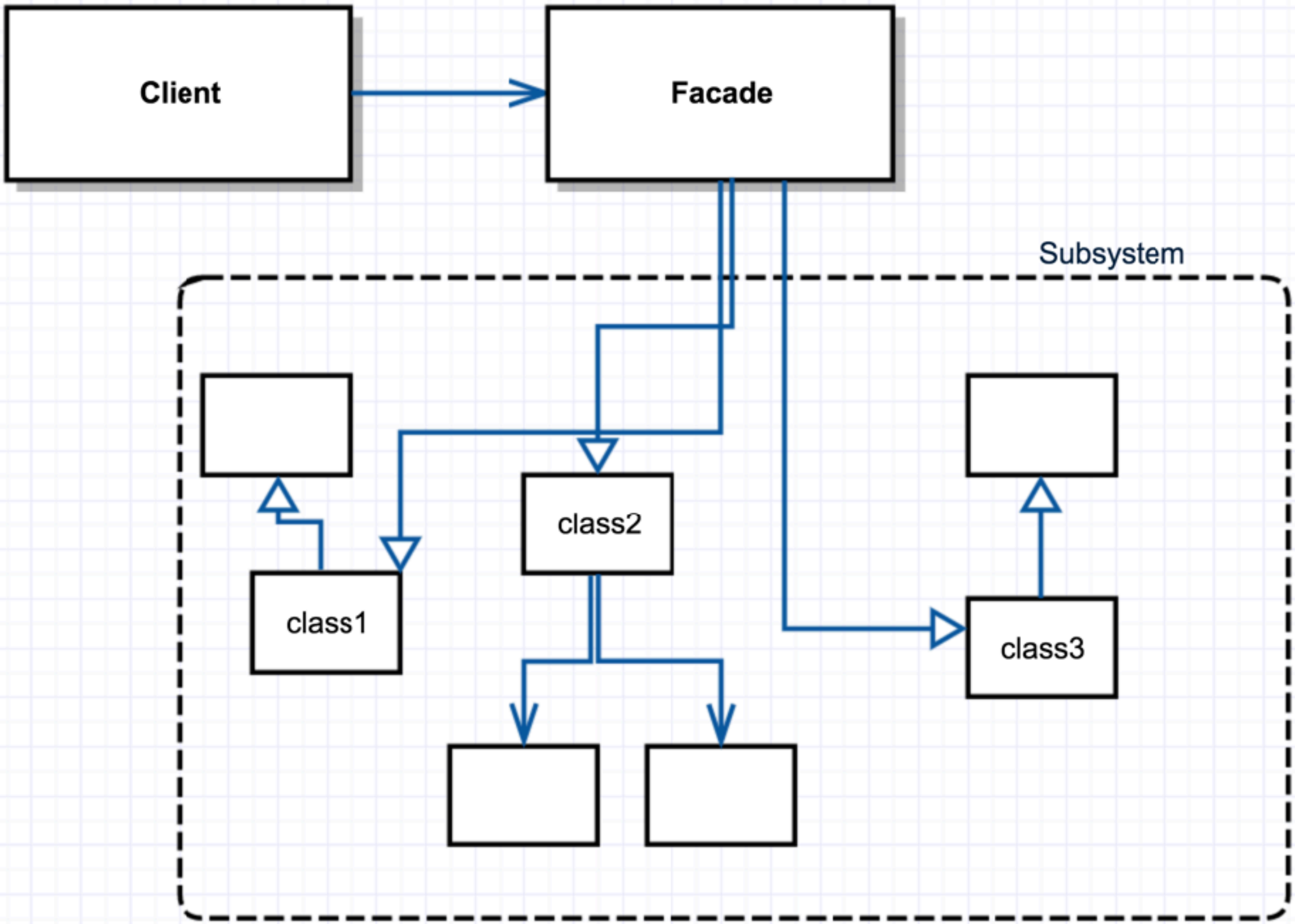
defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without changing the algorithms's structure.

Real-World Example



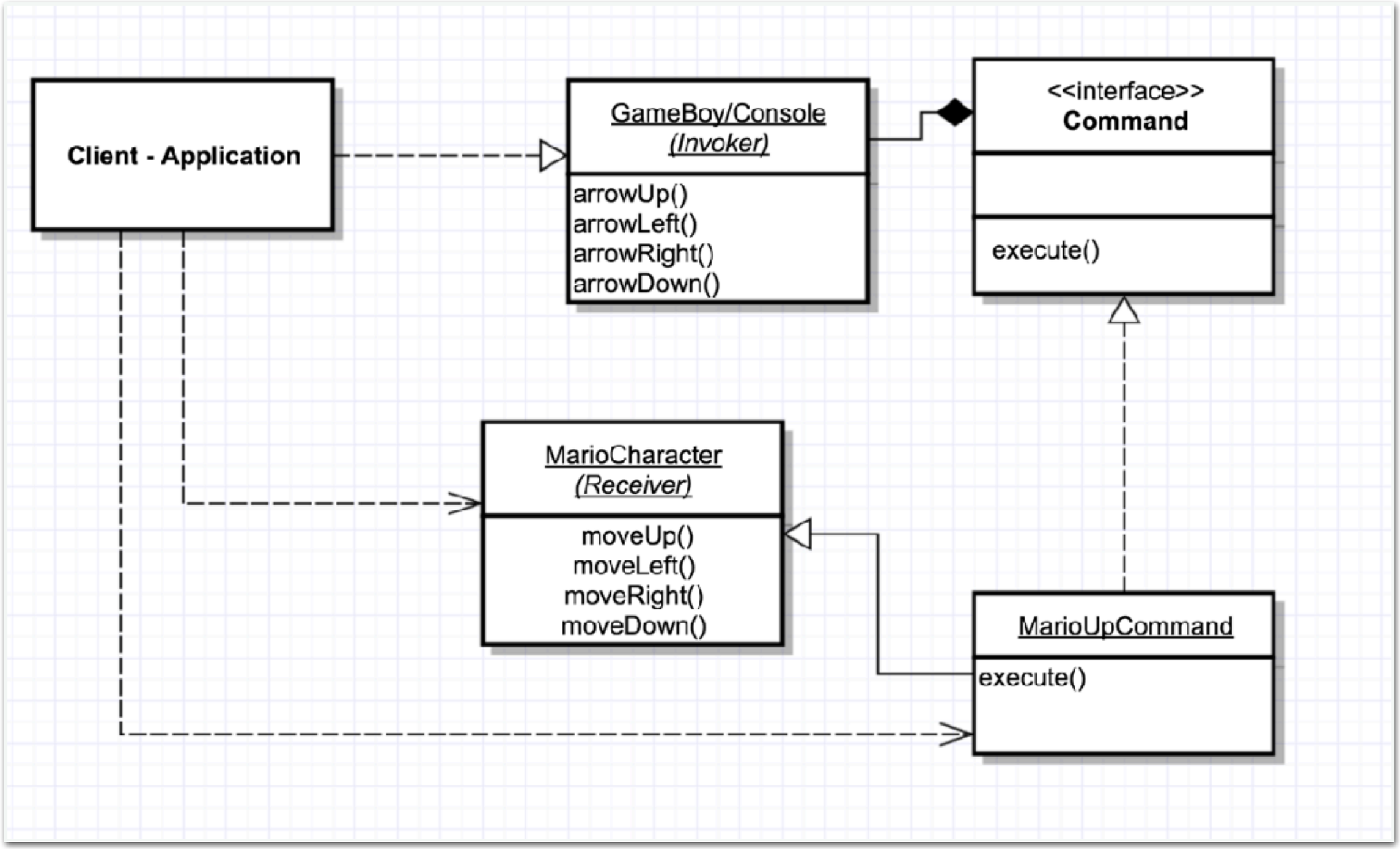
Too complex….

UML Facade Diagram



Facade - Definition

Definition…



The Command Pattern:

encapsulates a request as an object (the game card), thereby letting you parameterize other objects with different requests, queues or log requests.

The Iterator Pattern

Motivation

ArrayList

Array

Problem

ArrayList Array

Iterating through Products...

Obvious Solution

Two different catalogs …

Geeky Inventory/
Products

Dev Inventory/
Products

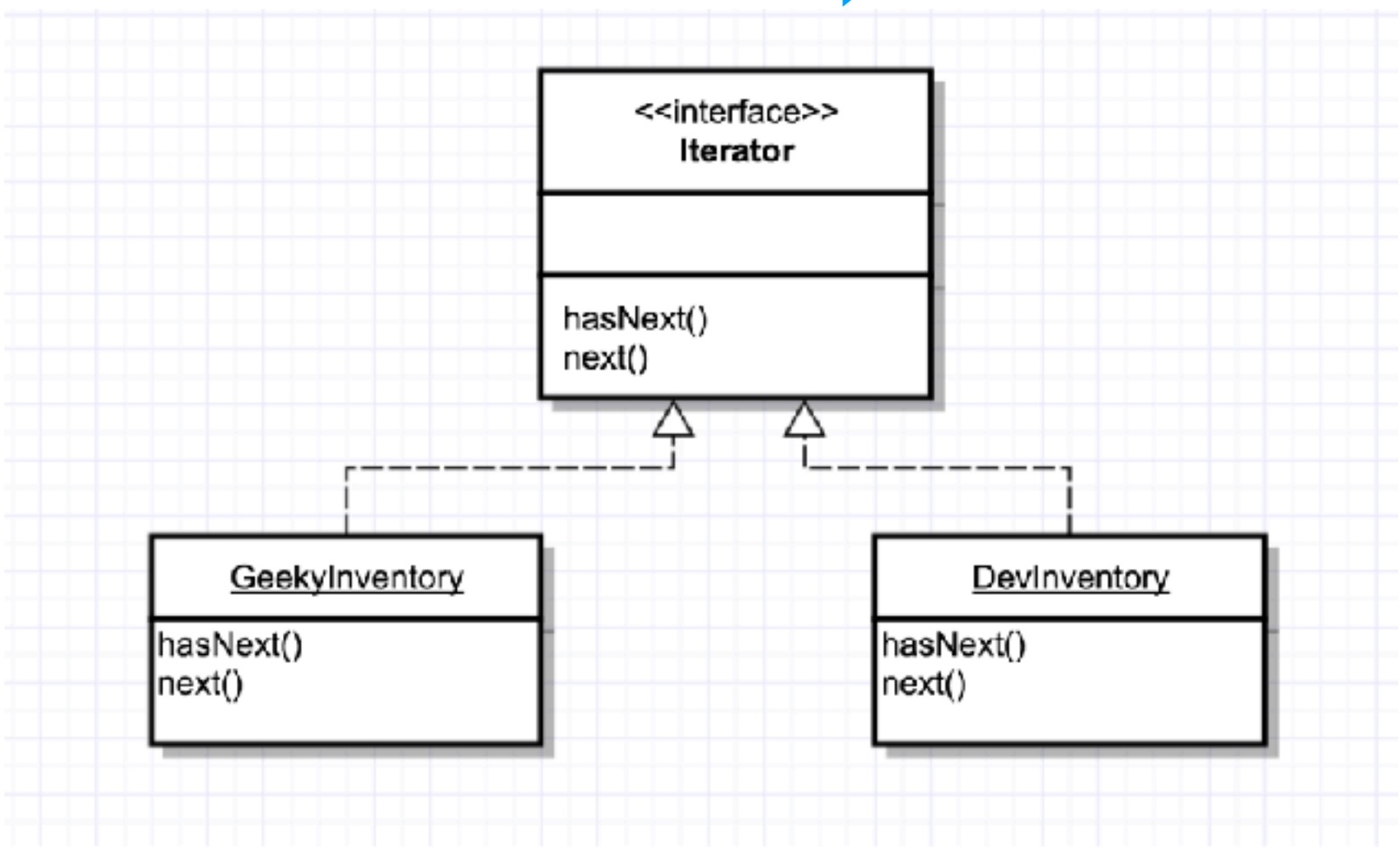
for loop 1 …

for loop 2 …

for loop 3 … etc

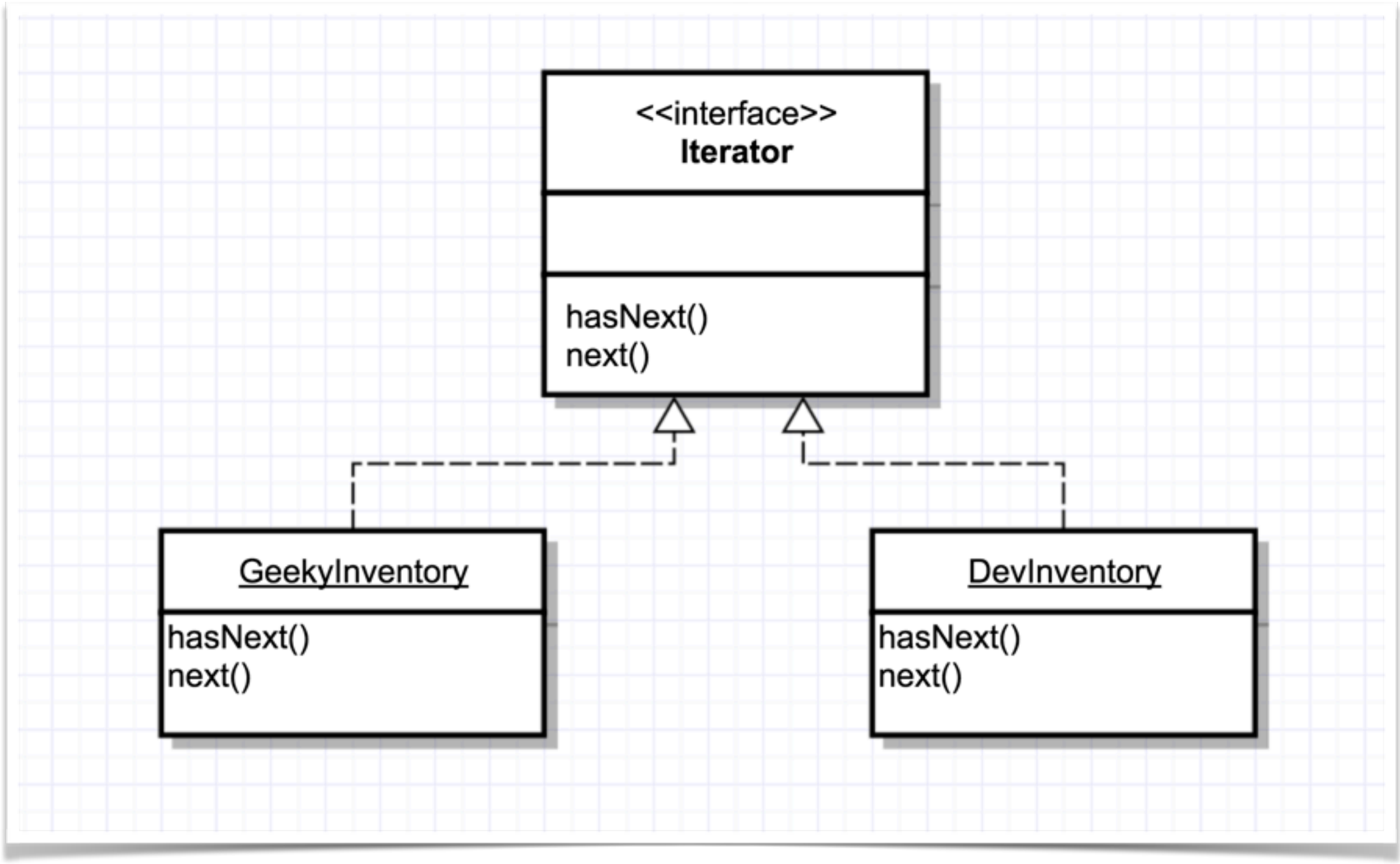
The Solution

Iterator



… to encapsulate the way we iterate through a collection of objects

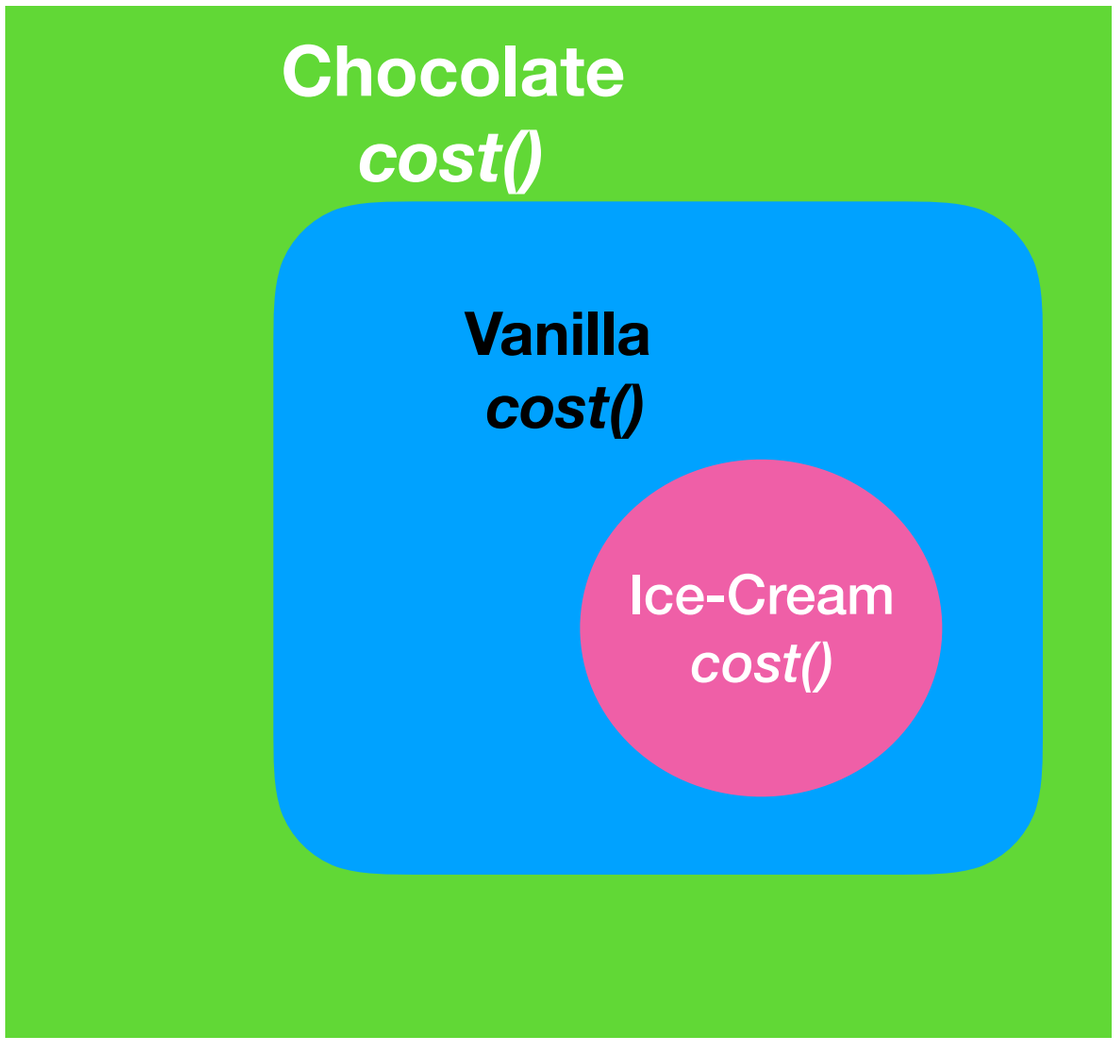
Definition…



The Iterator Pattern:
provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Create our Franchises…

The Result…



cost() will give us the final total cost

We wrap each ice-cream type with other toppings or types