프로그램 개발의 일반적 요구

- 프로그램 개발 단계
  - 요구 사항 분석, 설계, 코딩(개발), 운영 및 Feedback
  - 새로운 요구 및 여러 상황 변화에 따라 위의 과정을 반복 실행함
- 좋은 프로그램
  - 여러 상황 변화에 프로그램 수정 없이 (혹은 프로그램 수정을 쉽게) 대응함.
  - 프로그램을 수정하는 경우에도 기존에 개발된 코드를 가능한 유지하면서, 그 변화의 충격을 최소화 할 수 있도록 설계 개발됨
  - 노력을 적게하면서도 변화에 적응할 수 있음
- 어떤 프로그램 개발 방법론?
  - 좋은 디자인 패턴을 사용함으로써 가능함

Strategy 패턴

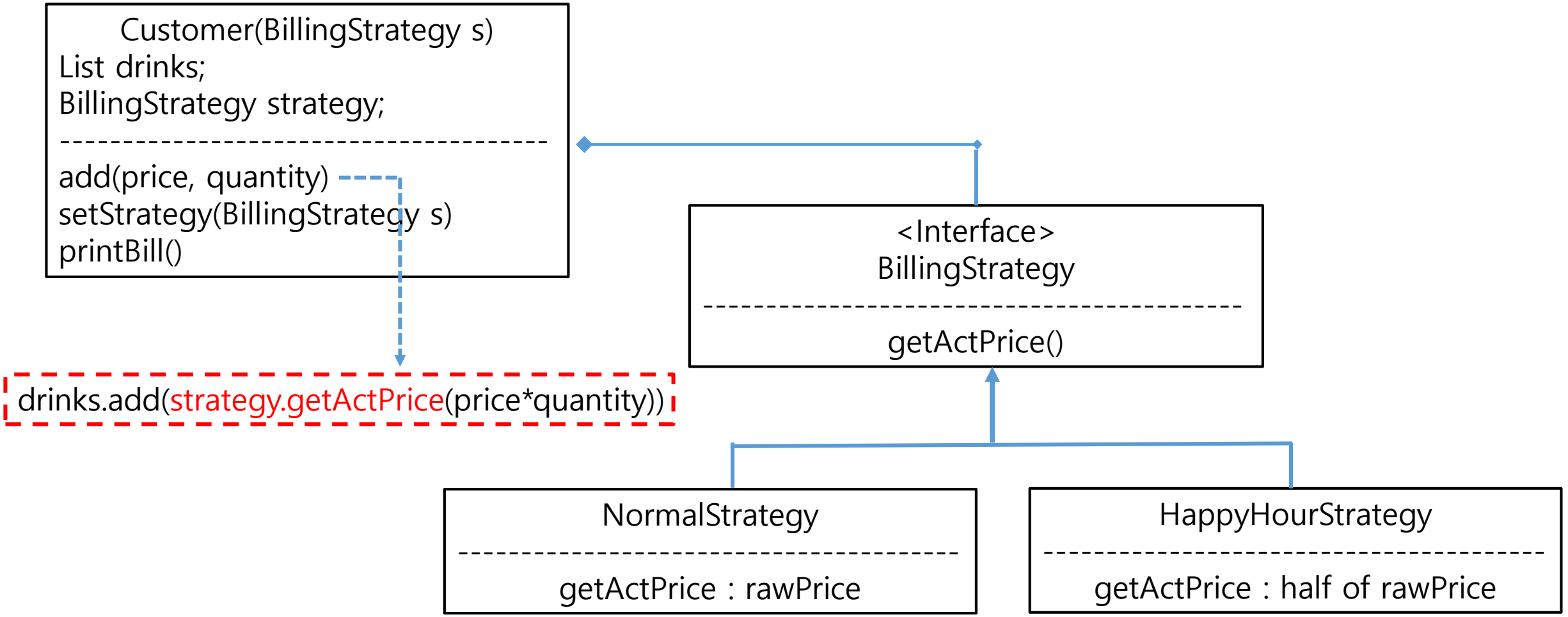
- Behavioral 디자인 패턴

- 프로그램 실행 중 상황에 맞는 알고리즘을 바꿔서 적용함
  - 적용 가능한 여러 알고리즘을 준비하고, 상황에 따라 적절한 알고리즘 선택

- 알고리즘 선택을 코딩할 당시가 아닌 프로그램 수행 과정 (동적)에서 이루어지도록 함.
  - 상황이 변화에 따른 코드의 변화를 줄일 수 있음

- Interface 이용
  - Interface에 구현될 알고리즘을 추상 메소드로 등록
  - 이 메소드를 구현할 알고리즘에 따라 Interface를 implements 하는 class 정의
  - 각 클래스에 대한 객체를 생성하고 이들을 List(혹은 Collection)에 저장
  - List의 각 객체는 그에 해당되는 메소드를 수행함으로써, 그에 따른 알고리즘을 실행

카페 운영 프로그램
UML Diagram



프로그램 배경 설명

- 각 customer가 주문한 각각의 drink * 개수를 계산함
- Drink의 가격은 두 가지 상황에 따른 가격으로 계산함
  - normalStrategy – 정상 가격
  - happyHourStrategy – 정상 가격의 반 값
- 각 customer 마다 normalStrategy, 혹은 happyHourStrategy 둘 중에 하나의 정책이 주어지는데,
- 이 정책은 고정된 것이 아니라 상황에 따라 수시로 변경될 수 있음 (즉, 동적으로 결정됨)
  - 즉, normalStrategy로 시작하였지만, happyHouerStrategy로 변경될 수 있으며, 그 반대의 경우도 발생할 수 있음
- 각 drink 값의 계산은 이 strategy에 따라 계산됨.

동적 정책 변화를 위한 Interface 정의 및 구현

- Strategy에 근거하여 값을 계산하는 getActPrice() 메소드
- 이 메소드는 normalStrategy인 경우 주어지는 값 그대로를, happyHourStrategy인 경우 주어지는 값의 반 값을 계산함
- 상황에 따라 동적으로 변화하는 strategy를 반영하는 계산을 위해서, getActPrice() 메소드를 BillingStrategy Interface에 등록시키고
- 각 strategy에 따라 다르게 계산하는 방법을 적용하기 위해 이 인터페이스에 대한 두 개의 class인 Normal Strategy와 HappyHourStrategy를 implement한다. (객체가 액션?)
- 이 두 class에 대한 객체들의 타입을 그 클래스의 superclass인 BillingStrategy으로 정한다. 즉, 이 타입을 갖는 객체는 normal, 혹은 happyHour를 표현하는 strategy를 의미한다.

Customer의 속성(Attribute)

- 각 customer는 두 가지 속성을 갖는다.
  - List<Double> drinks : 주문한 drinks 들의 값들을 저장함
  - strategy : 값 계산에 적용되는 normal 혹은 happyHour 정책 중 하나
    - 이 값은 상황에 따라 동적으로 바뀔 수 있음

- Customer의 객체
  - 객체마다 List<Double> 형태의 drinks와 strategy가 주어짐.
  - 객체 생성시 strategy 를 초기화함
  - 필요에 따라 setStrategy 메소드를 적용하여 이 값을 조정할 수 있음

Strategy 패턴의 특징

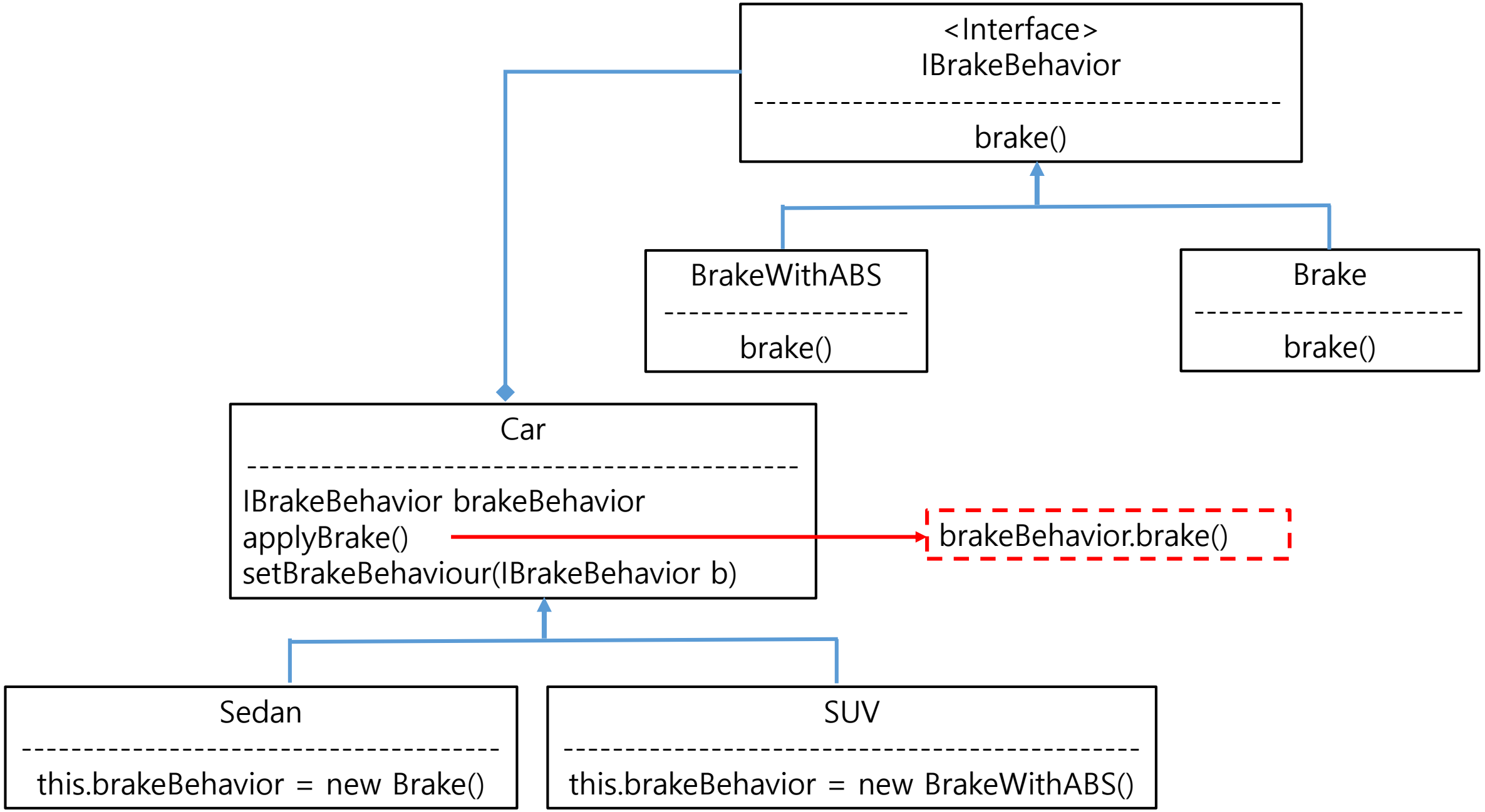
- Customer 클래스 중에서 동적인 변화가 필요한 getActPrice 메소드 관련 부분을 Customer 클래스에서 분리시키고, 이를 Interface (즉, BillingStrategy)로 위임함(delegate).
- Customer 클래스에는 BillingStrategy 객체를 표현하는 instance 변수(strategy)를 포함시키며, 이 변수의 값을 동적으로 변화시킴으로써 적용할 strategy를 바꿀 수 있음
- BillingStrategy Instance에 대해서 각 정책에 따라 구현이 달라질 부분들을 class로서 implements함
- 결과적으로 strategy가 다양한 형태로 변화되더라도, Customer 부분의 코드에는 영향을 미치지 않음. Interface를 이용한 encapsulation이 적용됨.

Strategy 변화에 따른 코드의 수정

- 현재의 strategy는 normal과 happyHour 두 가지이다. 여기에 새로운 strategy 인 member를 추가한다.
- Member customer에게는 모든 drinks를 30% 할인된 금액으로 제공하기로 한다.
- 핵심 기술 (코드의 변환 여부)
  - 새로운 strategy가 추가됨에 따라 그에 대한 BillingStrategy의 class가 정의되어 implements가 되어야 한다.
  - 그렇지만, 이것이 기존 코드에 영향을 미치지는 않는다.
  - 결과적으로 strategy가 변화하더라도 이에 대응하는 코딩은 매우 flexible 하게 처리될 수 있다.

Strategy and open/closed principle

- The behaviors of a class should not be inherited
- Instead, they should be encapsulated using interfaces
- Open/closed principle(OCP)
  - Classes should be open for extension but closed for modification
- Car class
  - Behaviors are defined as separate interfaces and specific classes that implement these interfaces
  - Better decoupling between the behavior and the class that uses the behavior
  - Behavior can be changed without breaking classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.
  - Behavior can also be changed at run-time as well as at design-time.


<Interface>
IBrakeBehavior
brake()
BrakeWithABS
brake()
Brake
brake()
Car
IBrakeBehavior brakeBehavior
applyBrake()
setBrakeBehaviour(IBrakeBehavior b)
brakeBehavior.brake()
Sedan
this.brakeBehavior = new Brake()
SUV
this.brakeBehavior = new BrakeWithABS()

Interface 타입에 의한 동적 수행

```
interface I { void ma(); }

class A implements I { public void ma() { System.out.print("A"); } }
class B implements I { public void ma() { System.out.print("B"); } }
class C implements I { public void ma() { System.out.print("C"); } }
class D implements I { public void ma() { System.out.print("D"); } }

public class InterfaceTest {
    public static void main(String[] args) {
        ArrayList<I> arrs = new ArrayList<> ();
        arrs.add(new D()); arrs.add(new B()); arrs.add(new A()); arrs.add(new C());
        Iterator<I> it = arrs.iterator();
        while (it.hasNext()) { I obj = (I) it.next();  obj.ma(); }
        for (I obj : arrs) obj.ma();
    }
}
```