**1. 애플리케이션에서 달라지는 부분을 찾아 내고, 달라지지 않는 부분으로부터 분리시킨다.**

첫번째 원칙은 스트래티지 패턴에 해당하는 장에서 등장했다.

새로운 요구사항이 추가로 들어왔다. 혹은 기존 화면의 요구사항이 변경됬다.

이럴때 우리는 클래스의 메소드나 생성자를 고치고 추가하는 작업을 한다.

그 다음에 해당 클래스를 사용하는 모든 코드를 고친다. 생각만해도 끔찍하다.

나중을 위해... 코드에 바뀌는 부분이 있다면, 그 행동을 기존 코드에서 분리시켜야한다.

\* 바뀌는 부분을 따로 뽑아서 캡슐화한다.

\* 이를 지키면 나중에 바뀌지 않는 부분에 영향을 미치지 않을 채로 그 부분만 고치거나 확장할 수 있다.

Class A {  
    public methodA() {

바뀌지 않는 부분

**바뀌는부분 -> 이 부분을 다른 클래스로 캡슐화한다.(분리)**

바뀌지 않는 부분

}

}

**2. 상속보다는 구성을 활용한다.**

두번째 원칙도 스트래티지 패턴에서 등장했다.

객체지향에서는 무엇보다 재사용성, 확장성을 중요하게 여기는거 아닌가?

어떤 클래스를 상속받아서 확장하면 부모 클래스의 모든 것들을 재사용할 수 있으니까 상속이 좋은거 아니야?

라고 생각했었다. 책에서 사부와 제자의 대화 형식으로 상속과 구성에 대한 내용이 있다.

즉 제자의 생각이 위에서 나의 생각과 비슷했는데, 상속을 통해서 개발 시간을 단축할 수 있다는것이다.

여기서 사부는 질문을 던진다. 개발이 끝나기 전과 개발이 끝난 후 어느 쪽 코드에 시간을 더 많이 쓰게 되느냐?

나는 아직 프로젝트 유지보수를 해보지 않았고, 첫 프로젝트를 진행하고 있다.

당연히 개발이 끝나기 전이라고 생각했다. 짧게는 3개월 ~ 길게는 1, 2년의 개발 기간이 결코 짧다고 생각하지 않았기 때문에..

헌데 사부와 제자의 대화는 달랐다. 개발이 끝난 후 개발기간보다 개발이 완료된 소프트웨어를 유지보수 하는데

더 많은 시간과 고민이 필요하다는 것이다. 때문에 사부는 관리의 용이성과 확장성을 고려해야한다고 한다.

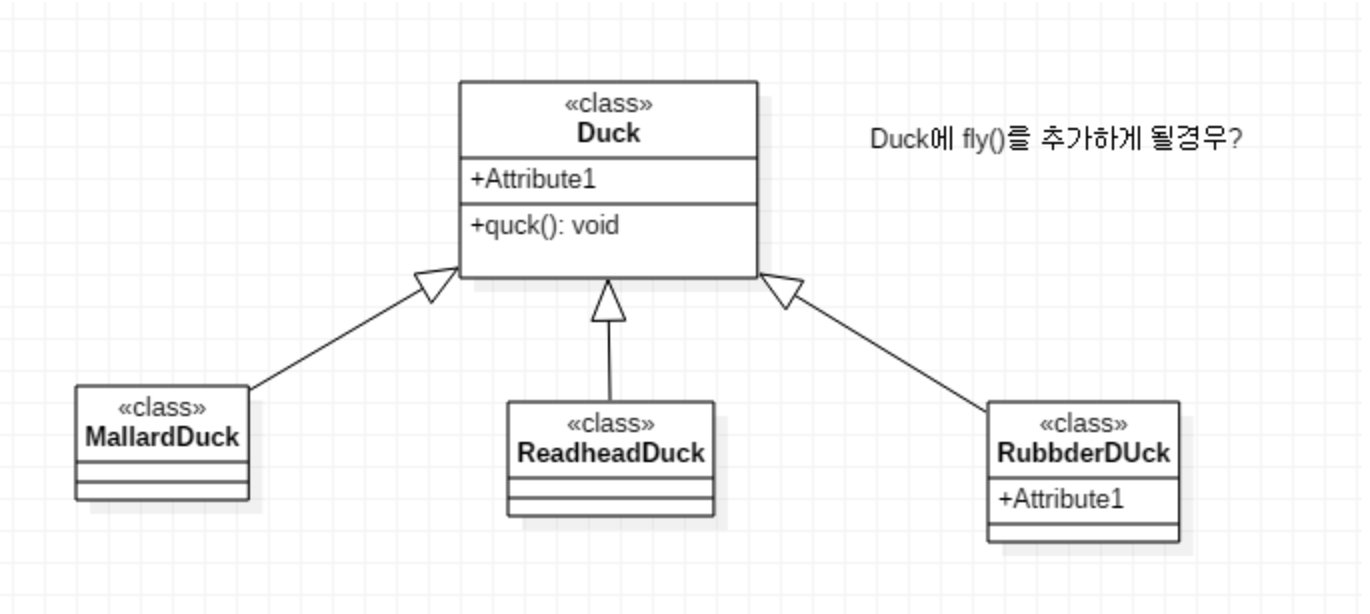
당연하지만 유지보수에서 관리의 용이성과 확장성이 중요하다.

상속을 사용할 때는 여러가지 단점이 있다고 한다.

(1) 서브클래스에서 코드가 중복되는 경우가 생길 수 있다.

(2) 실행시에 특징(행동, 실제 인스턴스의 타입)을 바꾸기가 힘들다.

(3) 코드를 변경했을 때 다른 클래스에 의도치 않은 영향을 끼칠 수 있다.



결국, 코드의 한 부분만을 바꿨을 뿐인데 프로그램에 오류나 이상한 결과를 유발할 수 있다는 것이 중요한 것 같다.

책에서는 quack()라는 메소드를 갖는 Duck클래스와 이를 상속받는 각종 오리들의 예를 보여주었다.

헌데 Duck클래스에 fly()라는 메소드를 추가했을 때,

날 수 없는 오리들까지도 이를 상속받아 고무로된 오리 인형이 날아다니는 어처구니 없는 결과가 나왔다.

그러면 RubberDuck에서 fly()메소드를 오버라이드해서 아무것도 하지 않거나 예외를 날리면 되잖아?

오리의 종류가 잦은 주기로 추가된다면 이러한 행위를 계속해야한다.

나무로 된 오리, 오리인척하는 거위, 오리알.... 등등

오리가 아닌것 같지만 오리로 취급 해야하니까... 규격은 계속 변경될 것이고

그럴 때마다 우리는 Duck의 메소드와 서브클래스들의 메소드를 일일히 살펴보고 상황에 따라

Duck을 상속할지 서브클래스의 서브클래스로 만들지를 정하고 또 어떤 메소드를 오버라이드해야할지 정해야한다.

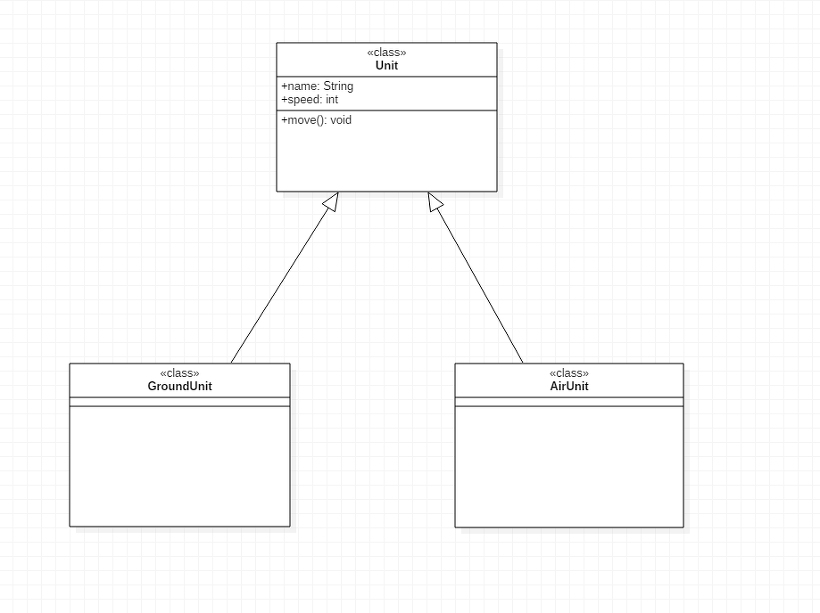
개발자들은 이런 무의미하면서 반복적인 행위를 없애기 위해 노력한다!  
일부 형식의 오리만 날거나 꽥꽥거릴 수 있도록 미리 정의하는 더 깔끔한 방법을 찾아야 한다고 한다.

그 방법은 3번째 원칙과 스트래티지 패턴에서 등장한다.

**3.구현이 아닌 인터페이스 맞춰서 프로그래밍한다.**

세번째 원칙 또한 스트래티지 패턴에서 등장했다.

스타크래프트의 테란 종족을 생각해보자.



(1) 모든 유닛을 의미하는 Unit

(2) 지상 유닛을 의미하는 GroundUNit

(3) 공중 유닛을 의미하는 AirUnit

편의상 움직이는 부분만 정의했다.

지상유닛에는 SCV, 탱크, 마린 등등

공중유닛에는 배틀크루저, 스카우터 등등이 있다.

SCV라는 유닛에는 다른 유닛을 수리할수 있는 기능이 있다.

이 기능은 기계유닛에게만 사용할 수 있다는 조건이 있는데,

위 클래스 다이어그램에서 어떻게 기계 유닛을 구분할까?

단순히 오버로딩을 이용해 기계유닛의 수만큼 메서드를 만들면된다.

?? 기계유닛이 얼마나 있는줄 알고 그걸 다만들어;

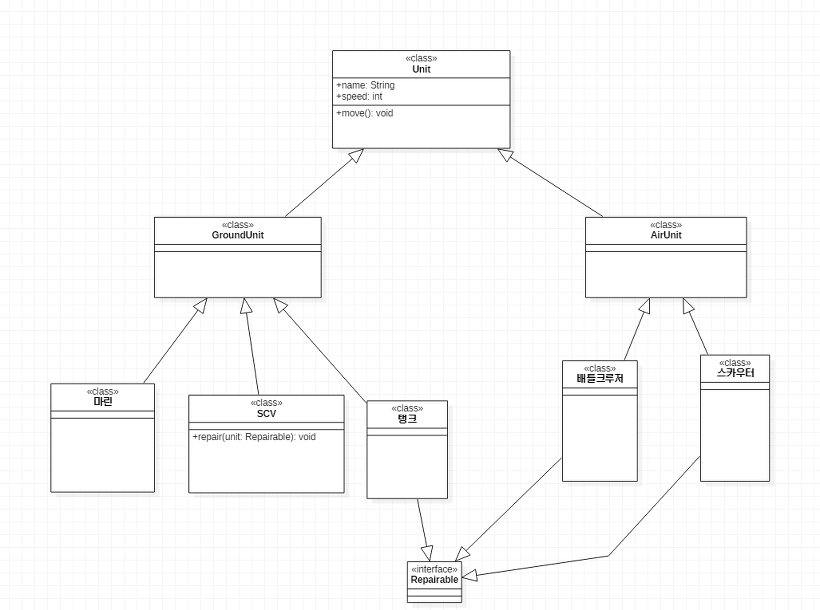
그럼 RepairableUnit 클래스 만들면 되잖아! 다중 상속 안된다.

자바에서 Comparable 과 같은 인터페이스가 있다.

이 녀석은 정렬이 가능한 녀석이라는 특성을 주기 위한 인터페이스다.

우리는 Repairable라는 인터페이스를 만들면 해당 인터페이스를 구현하는 클래스는

수리가 가능한 녀석이야! 라고 표현할 수 있겠다.



Repairable이라는 인터페이스를 만들고 기계유닛인 탱크, 배틀크루저와 스카우터에서 해당 인터페이스를 구현했다.

Repariable은 아무것도 정의되어 있지 않은 인터페이스다. 단순히 구현만 해주면 된다.

이제 우리는 SCV클래스의 repair라는 메소드의 파라미터를 제한할 수 있게 되었다.

SCV scv = new SCV();

scv.repair(마린) -> 에러!

scv.repair(배틀크루저) -> 호출

메소드를 인터페이스에 맞춰서 프로그래밍한것이다.

public repair(탱크); public repair(배틀크루저); public repair(스카우터)

와 같이 특정 구현에 맞추게 되면 repair를 변경하면 각각 다 변경해줘야한다.

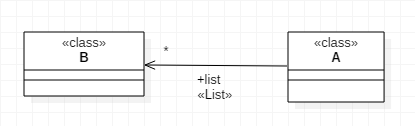
인터페이스에 맞추게 되면 하나의 메소드만 수정하면 된다.

**4. 서로 상호작용을 하는 객체 사이에서는 가능하면 느슨하게 결합하는 디자인을 사용해야 한다.**

**(Loose Coupling)**

네번째 원칙은 스트래티지패턴에 이어 등장한 옵저버 패턴에서 소개됐다.

여기서 상호작용이란 일대일, 일대다, 다대다 관계에 속한 클래스간의 의존도를 말하는것 같다.



A클래스가 B클래스에 강하게 의존하고 있다.

B가 주제가 되는 입장인데, B에서 상태가 변경되거나 코드를 직접적으로 수정한다면

A클래스의 상태도 같이 변화하거나 메소드 호출 결과가 달라진다거나 할 수 있다.

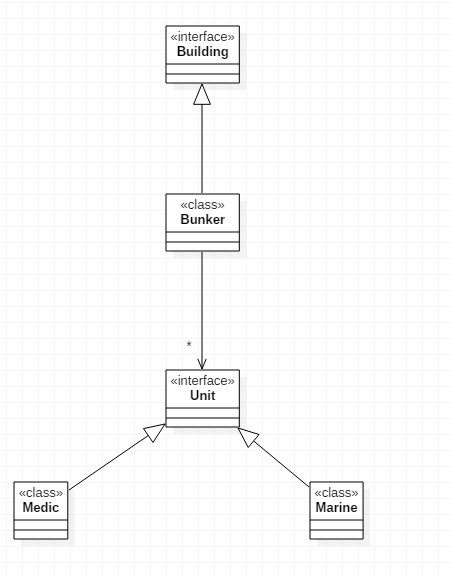
만약 B에서도 A의 기능을 사용해야한다면, 또 그런 B를 다른 클래스에서 사용한다면

A나 B를 수정하는 것만으로 전체 프로그램에 악영향을 미칠 수 있다.

때문에 Loose Coupling(느슨한 결합)을 항상 고려해야 한다고 한다.

두 객체가 느슨한 결합 상태라는 것은 서로 상호작용을 하긴하지만 인터페이스가 중재한다거나

하는 경우 서로에 대해 잘 모른다는 것을 의미한다.



위와 같이 구체적인 구현이 아닌 인터페이스와 같이 추상적인 것에 의존하도록 하게 하라는 의미인것 같다.

Bunker에서는 실제 유닛이 Medic이든 Marine이든간에 그저 자신의 안으로 집어넣기만 하면 되니까

실제 구현 클래스에 의존하지 않게 된다.

실제로 이렇게 짜여져있진 않겠지만 벙커에 탑승할 수 있는 실제 유닛을 추가한다고 해도

그냥 Unit을 구현하는 클래스이기만 하면 된다. 벙커쪽을 손댈 필요가 전혀 없다.

갑자기 상사가 저글링을 벙커에 넣을 수 있도록 하라고 요구한다.

다른 종족유닛을 어떻게 벙커에 집어넣어... 실제로는 그렇겠지만 만약에 그러한 경우가 생기면

저글링 클래스파일로 가서 implements에 Unit만 추가해주면 된다.

Bunker가 변경되면? Unit이라는 인터페이스의 인스턴스를 알고만 있다면 무엇이 변하든

마린과 메딕은 저놈이 뭘 하든 신경쓰지 않는다.

근데 앞에서 말한 원칙을 지키면 결국 얘가 되는거 아닌가..?

**5. 클래스는 확장에 대해서는 열려 있어야 하지만 코드 변경에 대해서는 닫혀 있어야 한다.**

**(OCP: Open-Closed Principle)**

다섯번째 원칙은 기존의 코드는 그대로 두고 확장을 통해서 행동이나 기능을 추가하기 위한

데코레이터 패턴에서 등장했다.

만약 기존의 코드가 해피콜의 종류별 일단위 수신율을 내보내는 코드였는데,

해피콜의 종류가 수정되면 기존 코드를 수정해야한다.

해피콜의 종류가 추가되면 상속을 이용하거나 새로운 메소드를 구현해야한다.

뭐 등등의 문제가 있겠다.

이렇게 수정사항이 올때마다 코드를 변경해야 하면 무척 곤혹스럽겠다

상속을 활용한 재사용성이 강력하긴 하지만 무조건 유연하고 관리하기 쉬운 디자인이 만들어지는건 아니라고 한다.

앞선 게시글에서 생각해봤듯이 대충

(1) 서브클래스에서 코드가 중복되는 경우가 생길 수 있다.

(2) 실행시에 특징(행동, 실제 인스턴스의 타입)을 바꾸기가 힘들다.

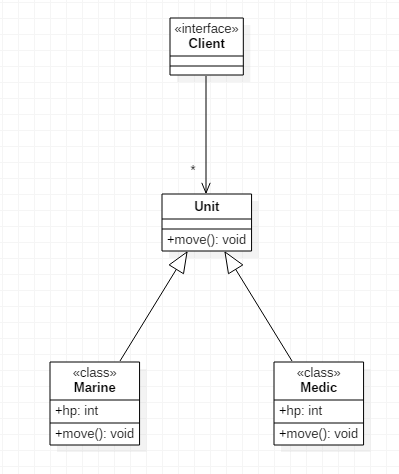
(3) 코드를 변경했을 때 다른 클래스에 의도치 않은 영향을 끼칠 수 있다.

이정도의 단점이 나타난다고 한다.

또 하나 등장한 단점은 서브클래스를 만드는 방식으로 행동을 상속받으면 그 행동은 컴파일시에 완전히 결정된다는 것이다.

게다가 모든 서브클래스에서 수퍼클래스의 똑같은 행동을 상속받아야 한다.

그러나 구성을 활용해서 객체의 행동을 확장하게 되면 실행중에 동적으로 행동을 설정할 수 있다.



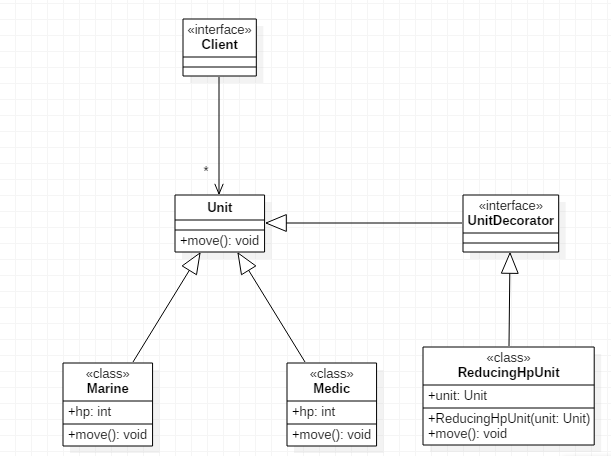
이번에는 실제 게임 이용자가 유닛을 컨트롤하는 상황이다.

부대를 지정해서 마린과 메딕을 한번에 움직이는 요청이 들어왔다.

그런데 유닛들이 이동하거나 공격할 때 일정 시간마다 체력이 깍이게 하고싶다.

이 책에서는 이러한 기능추가 해법의 하나로 데코레이터 패턴을 소개한다.

기능을 추가하고 싶은 객체를 감싸는 또 다른 데코레이터 객체를 만드는 것이다.



이동하면서 일정 시간마다 hp가 감소하게 하고싶으면 ReducingHpUnit으로 해당 유닛을 감싼다.

ReducingHpUnit의 move()메소드는 Unit의 move()메소드를 오버라이드한다.

그 다음 super.move()를 우선 실행시키고 그 다음 일정 시간마다 hp를 감소시키는 로직을 추가하면 된다.

이게 맞나..? 맞겠지 뭐

일단 어떠한 상황에 빗대서 생각해보고 추후에 "아 이건 이게 아니고 이거였네"

라고 고칠 수만 있으면 된다고 생각한다.

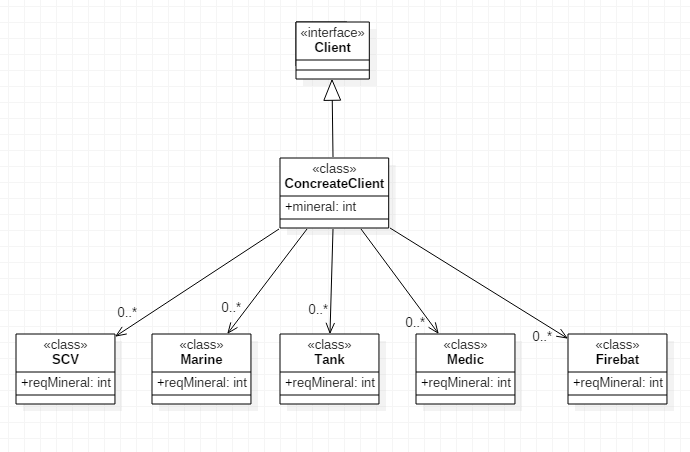
**6. 추상화된 것에 의존하도록 만들어라. 구상 클래스에 의존하도록 만들지 않도록 한다. (DIP: Dependency Inversion Principle)**

객체를 생성하는 팩토리 패턴 부분에서 등장한 여섯번째 원칙이다.

만약 내가 만든 코드를 사용하는 개발자가 직접 객체의 인스턴스를 만든다면

내가 제공하는 구상클래스에 의존해야 한다.

구상클래스를 여러개 제공한다거나 하면 개발자는 필요한 모든 객체의 인스턴스를 일일히 생성해야된다.



클라이언트가 게임을 시작하고 주어진 SCV로 미네랄을 모았다.

이제 마린이랑 메딕을 뽑아서 벙커를 짓고 방어태세를 구축하고 싶다.

개발자가 테란의 유닛클래스를 전부 찾아보고 필요한 미네랄의 양을 외우고

클라이언트의 요청에 맞는 유닛을 생성하기 위해 이러한 저러한 판단을 하고

로직을 추가해야한다면 꽤나 곤욕스럽겠다.

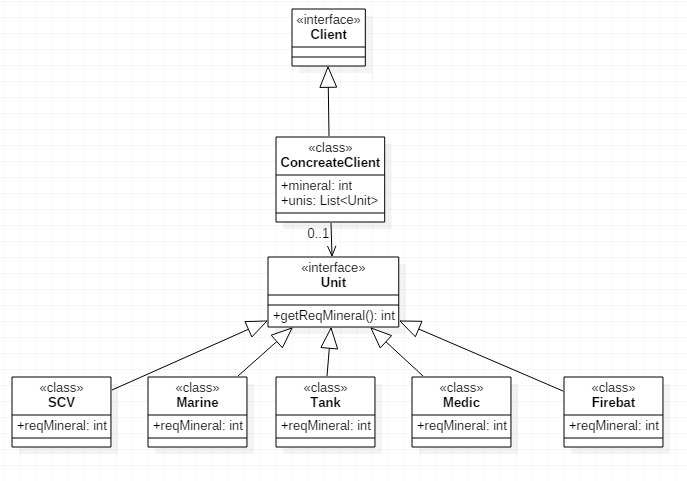
난 배럭과 같은 유닛생상공장이 팩토리 패턴의 훌륭한 예시라고 생각한다.

아 지금은 의존성 뒤집기원칙.......

어쨋든 저렇게 구상클래스에 의존하도록 디자인하지 말라는 의미다.

클라이언트가 모든 유닛객체들을 직접 생상해야 하므로 모든 유닛객체에 각각 강하게 의존한다.

유닛에 메소드가 추가되거나 상태를 추가, 제거하게 되면 클라이언트의 코드 또한 수정해야 할 수 있다.



이제 클라이언트는 Unit이라는 추상화된 것에 의존하게 되었다.

이 역시도 첫번째 스트래티지패턴에서 등장한 3가지 원칙과 비슷해보인다.

특히 인터페이스에 맞춰서 프로그래밍하라....

책에서는 의존성 뒤집기가 좀 더 높은 추상화를 요구한다고 한다.

고수준 구성요소(클라이언트)가 저수준 구성요소(SCV, Marine, Tank......Firebat)에 의존하면 안되고

항상 추상화된 것(Unit)에 의존하도록 강조한다고 한다.

뒤집기라는 말이 애매한데, 단방향으로만 흐르던 의존성을 고수준 구성요소 -> 추상화인터페이스 <- 저수준 구성요소처럼

무언가 뒤집어놓은 모양세라는 의미라고 한다.

**\* 의존성 뒤집기 원칙을 지키는 데에 도움이 될만한 가이드 라인**

(1) 어떤 변수에도 구상 클래스에 대한 래퍼런스를 저장하지 말자.(new 키워드를 직접 쓰지 않도록 팩토리로 제공하는 방법을 고려하라는 의미)

(2) 구상 클래스에서 유도된 클래스를 만들지 말자.(인터페이스나 추상화된것으로부터 클래스를 만들라)

(3) 베이스 클래스에 이미 구현되어 잇던 메소드를 오버라이드하지 말자.

(이미 구현되어있는 메소드를 오버라이드 해야하는 경우 수퍼 클래스의 추상화가 잘못되었다는 의미다.)

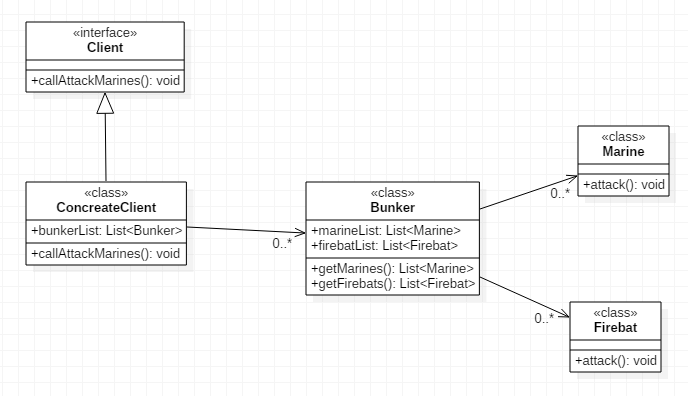
**7. 정말 친한 친구하고만 얘기하라.(최소 지식 원칙)**

**(Principle of Least Knowledge) = (Law of Demeter)**

일곱번째 원칙은 퍼사드 패턴에서 등장했다.

객체 사이의 상호작용, 즉 어떤 객체들이 다른 객체를 사용할 때는

아주 가까운 "친구"사이에서만 허용하는 것이 좋다는 원칙이다.

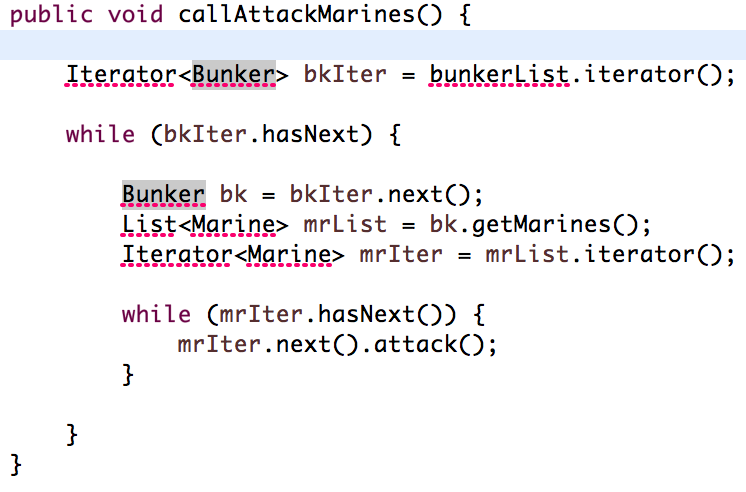


클라이언트는 여러개의 벙커를 소유하고 있고,

그 벙커는 다시 여러개의 마린과 파이어벳을 소유하고 있다.

이런 상황에서 클라이언트가 벙커의 모든 마린들이 공격하게 명령하는

callAttackMarines()를 살펴보자.



클라이언트의 메소드에서 벙커의 메소드를 호출하여 멤버인 List<Marine>을 가져왔다.

그런 다음 다시 Marine의 attack()메소드를 호출했다.

여기서 클라이언트와 친한 친구는 직접 관계를 맺는 List, Bunker뿐이다.

Bunker로부터 얻어온 Marine은 클라이언트와 친하지 않음에도 불구하고 Marine과 메시지를 주고 받는다.

Marine입장에선 기분도 나쁘고 클라이언트도 어색하고 그러겠지.

이러한 상황을 클래스들이 복잡하게 얽혀있다고 한다.

이런 상황에서 시스템의 한 부분을 변경하면 다른 부분까지 줄줄이 고쳐야될 수도 있다.

또한, 관리하기가 어렵고 남들이 보기에도 이해하기 어려운 코드가 되고 만다.

마린의 attack()메소드를 클라이언트뿐 아니라 벙커도 사용할 수 있다면

attack()의 메소드를 변경하거나 할 때 벙커와 클라이언트를 모두 고쳐야하는 상황이 된다.

아예 Bunker에서만 호출할 수 있게하고, 클라이언트는 간접적으로 Bunker의 callAttackMarines()를 호출하도록 하면

attack()메소드가 변경되었을 때 Bunker에서만 고쳐주면 된다.

그런데 이를 어떻게 매번 고민해서 지킨단 말인가. 책에서는 네가지 가이드라인을 제시한다.

(1) 객체 자체의 메소드

(2) 메소드에 매개변수로 전달된 객체

(3) 그 메소드에서 생성하거나 인스턴스를 만든 객체

(4) 그 객체에 속하는 구성요소

객체 자체의 메소드에서 (2), (3)의 원칙을 지키지 않는다면 당연히 이 원칙에도 위반된다.

객체 자체의 메소드에서도 메소드 내부에서 생성되거나 매개변수로 만든 객체의 메소드만 호출하거나

구성요소(A has B)의 메소드만 호출해야한다.

하지만 이 원칙에도 단점이 있다.

객체들 사이의 의존성을 줄이고, 관리의 용이성을 얻는 반면에

다른 구성요소의 메소드 호출을 처리할 때 래퍼(감싸는) 클래스를 만들어야 할 수도 있다.

그러다 보면 래퍼 클래스를 찾고 다시 그 래퍼 클래스가 어떤 클래스를 감싸는지 일일이 확인해야한다.

때문에 시스템이 더 복잡해지고 개발 시간도 늘어나고 성능이 떨어질수도 있다.

**8. 먼저 연락하지 마세요. 저희가 연락 드리겠습니다.(헐리우드 원칙)**

**(Hollywood Principle)**

8번째 원칙인 헐리우드 원칙은 템플릿 메소드 패턴에서 등장했다.

고수준 구성요소에서 저수준 구성요소로만 연락이 가게끔 하라는 것이다.

저수준 구성요소에서 시스템에 접속할 수는 있지만 고수준 구성요소를 직접 호출할 수 없게 하라고 한다.

그니까 저수준 구성요소는 프로그램에 참여만 하고 언제 어떤식으로 쓰이는지는 고수준 구성요소에서만 결정하라는 것이다.

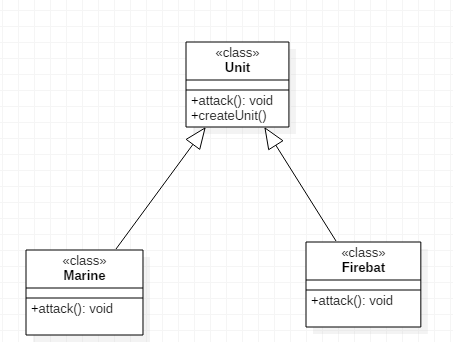
의존성 부패(dependency rot)를 방지하기 위함인데,

의존성 부패란 고수준 구성요소가 저수준 구성요소에 의존하고

다시 그 저수준 구성요소가 다른 고수준 구성요소에 의존하고

다시 그 고수준 구성요소가 다른 구성요소에 의존하고.....

와 같이 의존성이 복잡하게 얽혀있는 것을 일컫는다.



여기서 Unit은 추상클래스이며 고수준 구성요소이다.

유닛을 만드는 알고리즘을 장악하고 있고,

메소드의 구현이 필요한 상황(유닛을 만들때 특별한 로직이 필요하다던가..)에서만 서브클래스의 메소드를 호출한다.

반면에 Marine과 Firebat은 저수준 구성요소이다. Unit으로부터 호출 당하기 전에는

고수준 구성요소인 Unit를 직접 호출하지 않는다.

**9. 클래스를 바꾸는 이유는 한 가지 뿐이어야 한다.(단일 역할 원칙)**

**(SRP: The Single Responsibility Principle)**

마지막 원칙은 이터레이터 패턴에서 등장했다.

클래스에서 원래 그 클래스의 역할외에 다른 역할을 처리하도록 하면

두 가지 이유로 클래스가 바뀔 수 있다.

첫번째 역할때문에 클래스를 바꿔야 할 수도 있지만

두번째 역할때문에 클래스를 바꿔야 될수도 있다.

어느 클래스의 역할이 객체 생성이었는데 여기다가 생성된 객체를 관리하는 기능까지 추가하면

나중에 객체 생성 로직이 변경되고, 객체를 관리하는 컬렉션(집합체)가 변경되면 두 가지 이유로 인해

클래스를 변경해야 한다.

클래스를 고치는 것은 최대한 피해야 한다.

5번째 원칙인 OCP에서도 클래스는 변경에 대해서는 닫혀있어야 한다고 했다.

코드를 변경할 만한 이유가 두 가지 이상이 되면 그만큼 추후에 코드를 고칠 가능성이 크다는 것을 의미한다.

또한 해당 클래스를 변경할 때, 디자인에 있어서 두 가지 부분에 동시에 영향이 미치게된다.

이를 방지하려면 하나의 클래스에서는 한 역할만 맡도록 해야한다.

이 원칙을 제대로 지키는 방법은 디자인을 열심히 살펴보고 클래스가 바뀌는 부분에 주의하는 수밖에 없다고 한다.

아마 가장 어려운 원칙이 아닐까..