1

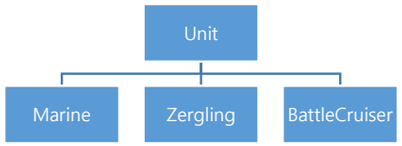
다음을 순차적으로 수행하여 상속 및 다형성의 핵심 개념을 이해합니다.

1. StarKHUraft 라는 게임의 유닛을 클래스로 디자인 해 봅시다.

게임에 등장하는 Marine 유닛, Zergling 유닛, BattleCruiser 유닛을 클래스로 디자인 해 보고 싶습니다.

게임에 등장하는 유닛들이 가질 수 있는 데이터로는 유닛의 위치(int pos[2]) , 공격력 (float attackPoint), 에너지 (float energy)가 있는데, 이는 모든 유닛이 공통적으로 가지고 있는 데이터입니다. 따라서 공통된 부분을 부모 클래스로 통합하는 디자인이 좋을 것 같네요. 이를 구현하면서 상속의 쓰임새 중의 하나인 공통된 부분 묶기 기능을 이해합니다.

다음 그림의 전체 구조로 4 개의 클래스를 디자인 합니다.



[Unit] 클래스

공통적으로 가지는 데이터를 통합하여 Unit 이라는 클래스에 모아 봅시다. 당연히 모든 변수는 private 입니다.

Unit 클래스에서 각 데이터를 초기화 하는 생성자도 만들어 봅시다. (기본 생성자, 4 가지 변수를 파라미터로 받아 초기화 하는 생성자)

각각의 데이터를 리턴해 주는 getXX, 변경해주는 setXX 함수들도 만듭니다. 그리고 모든 데이터를 화면에 출력해 주는 함수도 만듭니다.

[Marine] [Zergling] [BattleCruiser] 클래스

Marine, Zergling, BattleCruiser 클래스는 우선은 생성자만을 가집니다. 위치는 외부 파라미터로 받고, 공격력, 에너지는 다음 값으로 초기화 해 주는 생성자를 만듭시다. 예) Marine(int x, int y)

유닛	공격력	에너지
Marine	32.1	104.0
Zergling	21.5	50.0
BattleCruiser	82.4	250.0

메인에서 자식 클래스 3 가지의 객체를 각각 하나씩 만들고, 화면에 내용을 출력해 봅시다.

2. 1 번에서 int 의 배열로 표현했던 유닛의 위치를 대신해서, 2 차원 좌표를 표현하는 클래스 Pos2D 를 만들어 1 번의 클래스와 결합해 봅니다.

Listing 14.2 에서는 2 차원 좌표를 저장하는 간단한 클래스를 만든 적이 있습니다. 이 클래스를 확장해서 Pos2D 클래스를 만들고, 이 클래스의 객체를 Unit 클래스의 멤버 변수로 선언하여 유닛의 좌표를 표현할 수 있게 만듭시다. 이로써, 한 클래스가 다른 클래스를 가지고 있다는 것을 표현하는 방법을 이해합니다.

Unit 및 하위 클래스에서 수정이 필요한 부분들을 모두 고칩니다.

메인에서 자식 클래스 3 가지의 객체를 각각 하나씩 만들고 화면에 출력합시다.

3. 좌표를 표현할 때 2 차원 위치 뿐만 아니라 유닛이 평지에 있는지, 언덕위에 있는지, 하늘에 있는지도 표현해 주고 싶습니다.

이를 위해, 기존에 만들었던 Pos2D 클래스를 바로 수정해서 사용할 수도 있지만 Pos2D 클래스는 게임의 다른 부분에서도 사용하는 클래스라 바로 수정하면 문제가 많아 집니다.

따라서 Pos2D 클래스를 상속받아 Pos3D 라는 클래스를 만듭시다. 새롭게 추가되는 데이터인 높이를 표현하는 변수를 하나 추가하고, 이에 대한 필요한 함수들을 구현합니다. 이로써, 상속의 또 다른 쓰임새인 기존 클래스 변형해서 가져다 쓰기 기능을 이해합니다.

유닛 클래스 및 그 하위 클래스들의 내용도 이에 따라 수정해 주고, 메인에서 객체를 만들어 그 정보를 화면에 찍어봅니다.

4. 각 유닛의 필살기를 구현해 봅시다.

각 유닛은 자신만의 필살기가 있습니다. 이를 구현하기 위해 activateSpecialAbility() 라는 pure virtual 함수를 Unit 에 만들고, 이를 상속하는 클래스로 하여금 이를 꼭 재정의하여 사용하게 해 봅시다. 이로써 Abstract 클래스 및 순수가상함수의 용도를 이해합니다.

각 유닛 별 필살기는 다음과 같습니다.

- Marine 의 스팀팩: 스팀팩이 활성화 되면 공격 1 회에 대해서만 공격력이 1.5 배가 되고, 에너지가 20 이 깍임. 현재 스팀팩 상태인지 아닌지를 저장하는 (공격을 한번 사용했는지 아닌지를 저장하는) bool isSteamPacked 라는 변수를 Marine 클래스에 추가해야 함.

- Zergling 의 발업: 발업이 활성화 되면 3 번의 공격을 받는 동안 에너지가 1/2 만 깍임. 이 발업 Effect 가 몇 번 남았는지 저장하는 변수 int numberOfDefenceUp 변수가 Zergling 클래스에 추가로 정의되어야 함

- BattleCruiser 의 야마토포: 야마토포가 활성화되면 3 번의 공격동안 공격력이 300 이 됨. 몇 번의 야마토포가 남았는지 저장하는 변수 int numberOfYamato 가 BattleCruise 에 추가로 정의되어야 함.

자식 클래스 3 개에서 위의 설명에 따라 activateSpecialAbility 를 overriding 해 봅시다. 각 activateSpecialAbility 함수 안에서는 각각의 필살기와 관련된 변수들 (isSteamPacked, numberOfDefenceUp, numberOfYamato)을 세팅해 주고, 화면에 각각의 필살기가 활성화 되었다고 출력해 줍니다.

메인에서는 Unit* p; 를 하나 생성하고, 이 포인터에 각 자식 클래스의 객체를 순차적으로 동적 할당해 주면서 각각의 activateSpecialAbility 함수를 호출합니다. 화면의 결과가 다르게 나오는 것을 확인하면서 동적 바인딩을 이해합니다.

5. 유닛을 공격하는 기능을 추가해 봅시다.

attackUnit 이라는 외부함수를 만듭니다. 이 함수는 attacker, defender 라는 두 개의 파라미터를 받는데, 둘 다 Unit* 타입입니다. 이 함수는 attacker 유닛이 defender 유닛을 공격하는 함수이고, 기본적으로 attacker 의 공격력만큼 defender 의 에너지를 감소시키는 일을 수행합니다. 예를 들어, 위 함수는

```
defender->setEnergy(defender->getEnergy() - attacker->getAttackPoint());
```

정도의 코드가 들어가겠지요.

하지만 특정 유닛이 필살기가 활성화 되어 있는 상태라면 때에 따라서 공격자의 공격력 혹은 방어자의 에너지의 감소폭이 수정되어야 합니다. 이를 다형성으로 풀어봅시다.

Unit 클래스에서 setEnergy, getAttackPoint 함수를 virtual 함수로 만듭니다.

그리고 Marine 과 BattleCruiser 클래스에서 getAttackPoint 를 overriding 합니다. 예를 들어, Marine 에서는 만약 isSteamPacked 가 false 이면 그냥 attackPoint 를 리턴하지만 true 이면 attackPoint*1.5 를 리턴해 줍니다. BattleCruser 에서는 numberOfYamato 가 0 일때는 그냥 attackPoint 를, 0 이 아닐때는 300 을 리턴해 줍니다.

Zergling 클래스에서는 setEnergy 함수를 overriding 해 줍니다. numberOfDefenseUp 이 0 일때는 단순히 파라미터를 Energy 값에 대입해 주지만 0 이 아닐때에는 파라미터/2 값을 Energy 에 대입해 줍니다.

위의 virtual 함수를 재정의만 해 줌으로써, attackUnit 의 공격자 방어자 파라미터에 어떤 종류의 객체가 들어오던, 어떤 필살기가 활성화 되어 있건 간에 적절한 함수가 호출이 되어 공격의 계산이 적절히 이루어 집니다. 하나의 attackUnit 함수로 모든 다양한 상황을 퉁 칠 수가 있는 것이지요.

이를 구현함으로써, 다형성의 쓰임새를 이해했습니다.

attackUnit 함수의 끝에 공격의 결과를 적절히 화면에 출력해 주는 구문을 추가합니다. 유닛의 에너지가 0 이하이면 유닛이 파괴되었다는 출력도 해야 합니다.

메인에서는 다양한 종류의 객체를 생성하여 몇몇은 필살기를 활성화 시켜서 attackUnit 함수를 테스트 해 봅니다.

다음은 Main 코드의 한 예 입니다.

```
Marine M1(……);
BattleCruiser B2(……);
M1.activateSpecialAbility();
attackUnit(&M1, &B2)
```

2

17.5에서 17.16 까지의 코드를 하나하나 이해하면서 작성해 봅시다. 이 문제는 타 IDE 를 사용해 헤더파일과 구현파일을 분리해서 작성해 보는 연습을 합니다. 작성을 다 마치면, 모든 cpp 파일과 h 혹은 hpp 파일을 하나의 파일로 압축하여 제출합니다.