디자인 패턴 정리

디자인 패턴 사용: 연관관계를 최소화해서 수정되는 부분 최소화 하는 것이 목적

에디터 사용: 스케치 객체 추가 부분은 에디터 형으로 만들어서 추가 가능 하게 하자

인터페이스: 사람과 컴퓨터를 연결하는 장치 // 키보드 마우스

+ 기능에 대한 선언과 구현 분리

+ 기능을 사용 통로

* A인터페이스 선언 A 클래스에 A인터페이스 구현 A클래스에서 A인터페이스 기능 사용

델리게이트: 위임 코드 // 어떤 기능을 구현할 때 그 책임을 다른 객체에 떠넘기는 것 혹은 빌려 사용함

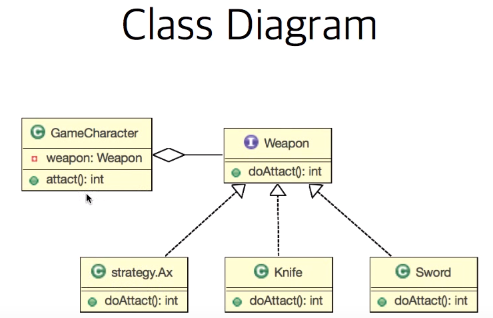
패턴 정리

1. 전략패턴: 여러 알고리즘을 하나의 추상적인 접근점을 만들어 접근 점에서 서로 교환 가능하도록 하는 패턴

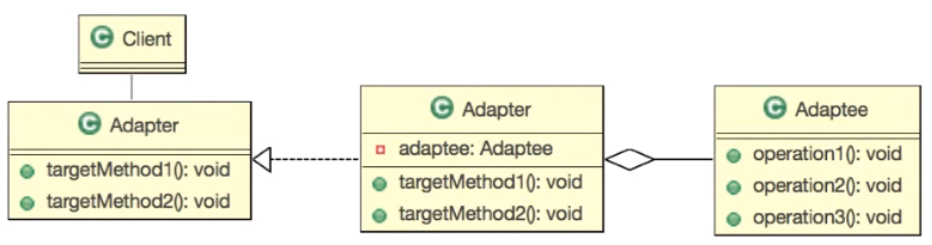
무기라는 인터페이스를 만들고 공격이라는 기능을 구현

공격이라는 기능을 구현한 여러 클래스 // 나이프 도끼.. 등등 클래스를 정의

클라이언트 클래스인 게임 캐릭터에서 weapon을 사용하면 해당 인터페이스를 통해서 오브젝트 교환가능



1. 어댑터 패턴 Adapter



다목적사용을 위한 부가기구 다른 곳에서 사용 가능 하게함

미리 주어지거나 구현된 기능을 같은 인터페이스로 사용할 수 있게 해주는 패턴

연관이 없는 두 부분을 연결할 때 사용

1. 템플릿 메소드 패턴 Template Method Pattern

알고리즘의 구조를 메소드에 정의하고 하위클래스에서 알고리즘 구조의 변경없이 알고리즘을 재정의 하는 패턴

조건

일정한 프로세스를 가진 요구사항을 템플릿 메소드 패턴을 이용하여 구현할 수 있다.

구현하려는 알고리즘이 변경 가능성이 있다.

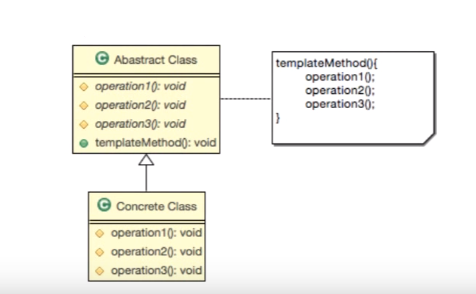
방법론

알고리즘을 여러 단계로 나눈다.

나눠진 알고리즘을 메소드로 선언한다.

알고리즘을 수행할 메소드를 만든다.

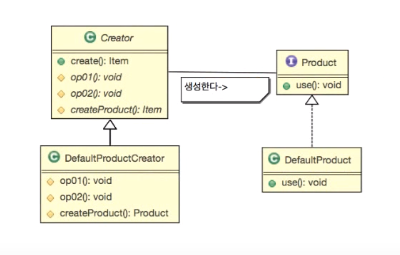
하위클래스에서 나눠진 메소드들을 구현한다.



1. 팩토리 메소드 패턴

템플릿 메소드패턴이 사용되어있다.

구조와 구현의 분리를 이해하고 구조와 구현의 분리의 장점파악



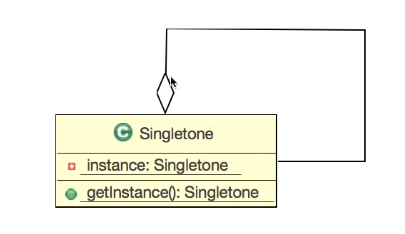
생성자와 생성객체를 나눔

Create 과정을 여러단계로 나눔 // 템플릿 메소드 형식 프레임 워크

생성 중간과정에 뭔가 새로운 알고리즘이 붙어도 메인 함수에서는 Create 만 사용해서 수정없이 사용가능 유연성 증가

1. 싱글턴 패턴

인스턴스를 하나만 생성하는 것

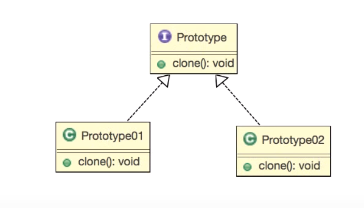


하나만 접근 가능하다 ex) 스피커 볼륨조절할 때 사용

1. 프로토타입 패턴

생산 비용이 높은 인스턴스 생산 시 사용

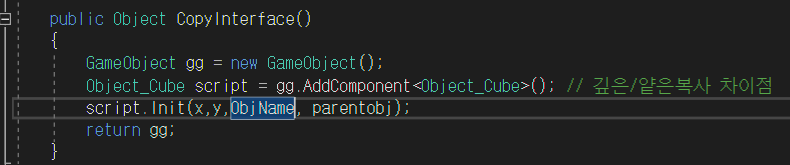
* 종류가 많아서 클래스로 정리되지 않는 경우
* 클래스로부터 인스턴스 생성이 어려운 경우



Copy 인터페이스를 만들어서 구현한 뒤 메인에서 사용

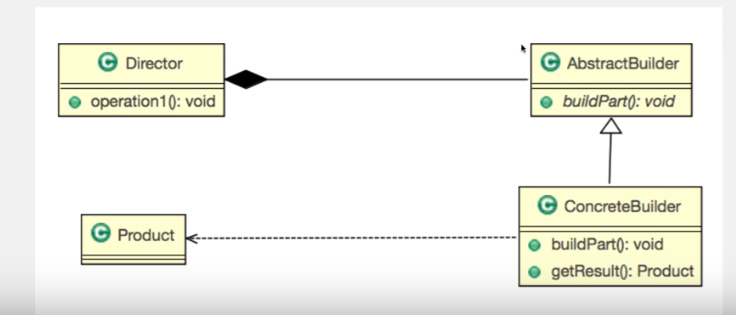
Copy방법은 언어와 클래스에 따라 다름 단 깊은 복사와 얕은 복사의 개념은 이해필요.

자료형 메모리는 복사가 되지만 사용자 메모리 class 같은 데이터는 구현부분에서 다시 할당해줘야한다.



1. 빌더 패턴

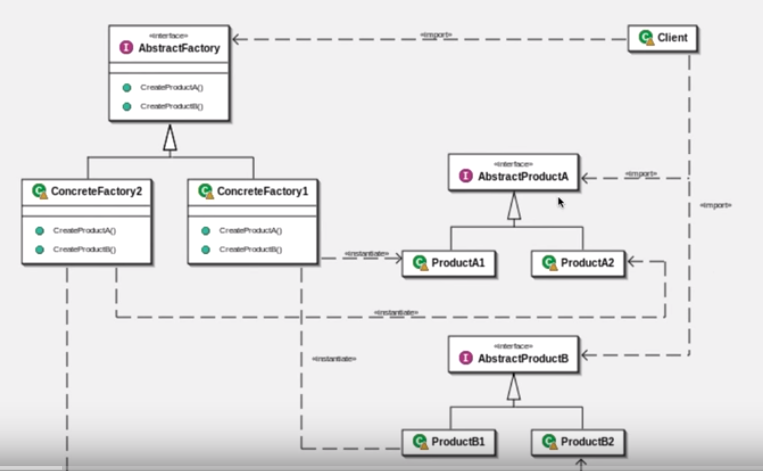
복잡한 단계의 객체 생성을 서브클래스에 넘겨주는 패턴



복잡한 단계의 객체를 여러 개의 클래스에 생성을 위임해 생성시켜 주는 패턴

1. 추상 팩토리 패턴

관련 있는 객체를 가상화해서 생성한다.



팩토리부분을 가상화해서 구체적인 부분을 가려주고 클라이언트는 가상화된 제품을 생성하는 패턴

어렵진 않지만 이해하기 복잡한 패턴이었음

예제

// 각 부품 인터페이스

public interface IChest

{

}

public interface IHead

{

//보는기능

void See();

}

// 부품구현

public class EastChest : IChest

{

}

public class EastHead : IHead

{

void IHead.See() //보는기능 구현

{

Debug.Log("EastHeadSee");

}

}

public class WestChest : IChest

{

}

public class WestHead : IHead

{

void IHead.See() //보는기능 구현

{

Debug.Log("WestHeadSee");

}

}

//Factory 인터페이스

public interface IFactory\_Insect

{

IChest Create\_Chest();

IHead Create\_Head();

}

//Factory 구현

public class Factory\_EastInsect : IFactory\_Insect

{

public IChest Create\_Chest()

{

return new EastChest();

}

public IHead Create\_Head()

{

return new EastHead();

}

}

public class Factory\_WestInsect : IFactory\_Insect

{

public IChest Create\_Chest()

{

return new WestChest();

}

public IHead Create\_Head()

{

return new WestHead();

}

}

// main

public class Main\_AbstractFactory : MonoBehaviour

{

private void Start() //main

{

// IFactory\_Insect factory = new Factory\_EastInsect();

IFactory\_Insect factory = new Factory\_WestInsect();

IHead head = factory.Create\_Head();

IChest chest = factory.Create\_Chest();

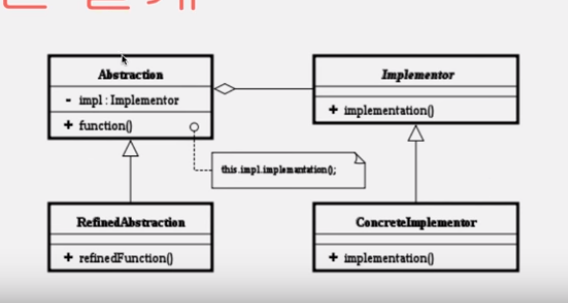
head.See();

}

}

1. 브릿지 패턴

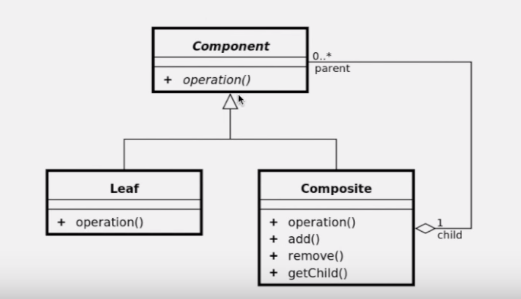
기능계층과 구현계층의 분리



잘 사용하지는 않음 예제코드 참고

1. 컴포지트

컨테이너와 내용물을 동일하게 다루기.



안에 리스트를 만들어서 소수로 묶거나 전체로 묶거나 가능

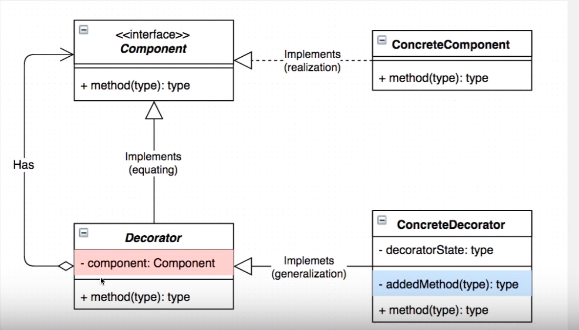
1. 데코레이터 패턴

상황 용도에 따라 객체에 필요한 요소들을 동적으로 추가한다.

기능 확장이 필요한 경우 클래스 상속대신 유연하게 사용가능

하나의 객체를 여러 개의 데코레이터를 추가할 수 있다.

너무 많으면 객체를 파악하기 어렵다.

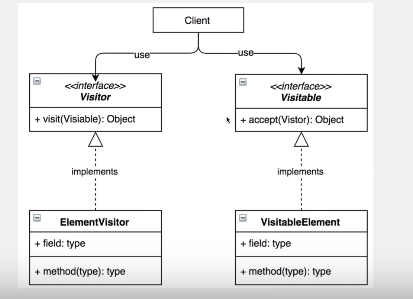


커피제조 예제

+설탕 시럽추가에 따라 가격이 결정된다.

1. 방문자패턴

객체의 처리부분을 분리하여 구현한다.

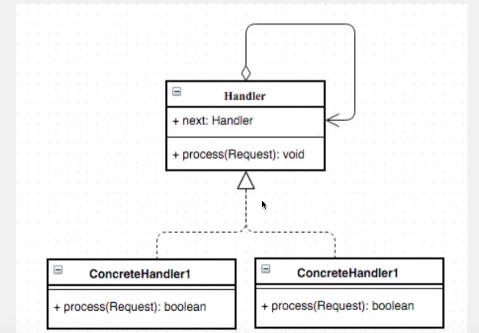


이미 구현된 객체에 새로운 기능을 추가할 때 사용

EX) 맴버변수의 총 값을 알고 싶다.

1. 책임사슬 패턴

다양한 처리방식을 유연하게 연결할 수 있다.



다음처리를 할 핸들러를 가지고 있고 처리에 실패하면 다음 처리로 연결한다.

Ex) 사칙연산 처리 / 특정 토큰을 넣고 프로세스를 돌리면 토큰비교후에 맞는 연산을 실행한다.

공격력 방어력 비교 공격값을 넣고 해당 플레이어의 착용 장비에따라 최종 데미지값을 연산한다.

1. 옵저버 패턴

상태변환 관찰하기 이벤트 발생 후 객체 외부에서 처리가능

