**소프트웨어 공학 HW3**

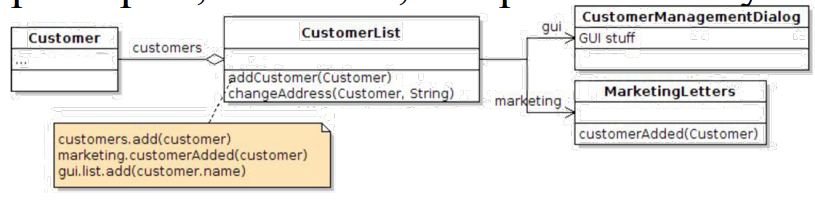
2017030055 윤상현

**HW3-1**

HW3-1 에서는 두가지 디자인을 비교해보고 더 나은 디자인에 대하여 논의하고자 합니다.

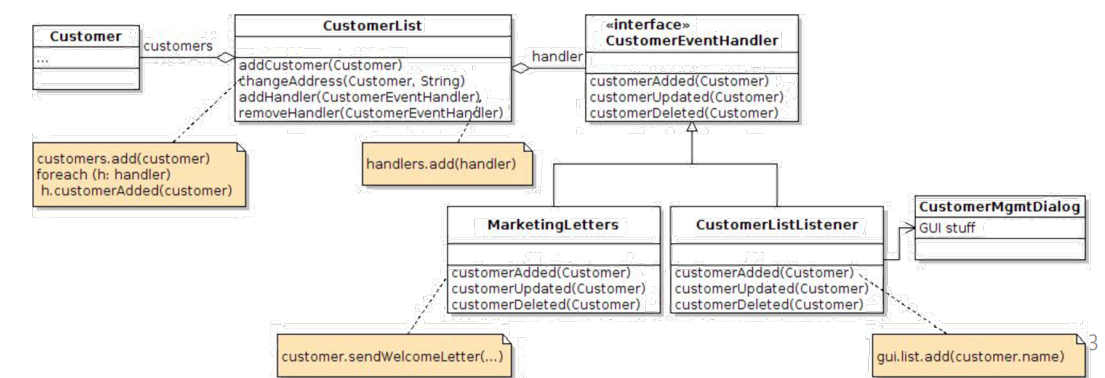
우선 제시된 2가지 클래스 다이어그램을 소개하겠습니다. 우선 각 다이어그램들을 분석해보겠습니다.

**1번 Diagram**



우선 첫번째 다이어그램에선 명확한 디자인 패턴을 찾을 수 없었습니다. 하지만 이 다이어그램 자체는, 당장 나타나있는 기능들을 구현하기에 Simple한 다이어그램 구조입니다. Simple is best라는 말이 있듯이, 간단하고, 가독성 좋은 구조는 타인으로 하여금 그 의도를 명확히 하고 이해하기 쉽게 만들어줍니다. 하지만 문제점 역시 존재합니다. 우선 이 다이어그램에서 OOP 원칙인 SOLID에서 SRP( Single Responsibility Principle) 원칙을 위반하고 있습니다. 위 다이어그램 중 CustomerList 클래스는 gui, 그리고 marketing이라는 두가지 기능을 지니고 있습니다. 이는 단일 책임 원칙을 위반하는데, 이 원칙은 하나의 클래스가 한 개의 책임만 지게하여 인과관계를 명확히하는 원칙입니다. 이렇게될 경우, 구조가 커질수록 한 클래스가 여러 기능에 책임을 가지게되어 내부에서 강한 결합이 발생할 수 있습니다. 이런 경우, 유지보수를 할 때 복잡성이 생기게됩니다. 예를들어 CustomerList가 책임을 클래스들에 새로운 기능을 추가할 경우, 이는 전부 CumtoerList가 책임을 가지게 되어 복잡해집니다. 따라서 위 다이어그램처럼 Simple한 다이어그램이 현 상황에서는 괜찮아보일지 몰라도, 추후의 확장성 등 장기적인 시점에서 고려한다면 좋지못한 디자인이라고 생각합니다.

**2번 diagram**



이번엔 두번째 다이어그램을 살펴보겠습니다. 이 다이어그램은 디자인 패턴 중 메소드 패턴을 사용하여 설계되었습니다. 템플릿 메소드는 행동 패턴의 일종으로써 특정 작업을 수행하는 부분을 서브 클래스로 캡슐화하여 특정 기능을 수정할 떄 전체를 수정하지 않고 해당 특정 기능을 담당하는 캡슐화된 클래스만 수정하여 전체적인 구조를 유지시킬 수 있는 패턴입니다. 또한 구현 중 서브 클래스들 간 중복된 코드가 있는 경우에는 상속을 통하여 이를 감지하고, 미리 알아내어 불필요한 구현을 방지할 수 있습니다. 메소드 패턴의 큰 틀은 상위의 추상화 클래스의 메소드들을 하위 클래스에서 구체화하는 것입니다. 위 다이어그램에서 MarketLetters와 CustomerListListener와 같은 기능들이 CustomerEventhandler 라는 추상화 계층에 의존하고 있는 것을 볼 수 있습니다. 위 다이어그램의 경우, 어떠한 일을 하던 고객에게서 생긴 Event를 Handle해야한다는 점은 바뀌지 않는 기능입니다. 하지만 그 하위에 있는 MarketingLetters, CustomerListner 와 같은 메소드들은 상황에 따라 변경, 삭제될 수 도 있습니다. 따라서 여기서 템플릿 메소드를 사용하는 것은 DIP를 지키는 좋은 방식이락 할 수 있습니다. DIP는 확장에는 열려있고 변경에는 닫혀있는 구조에 대한 원칙으로, CustomerEvnetHandler가 여러 하위 클래스들을 통하여 확장되는 것에는 열려있게 설계하는 것에 더해, CustomerEventHandler 자체는 변경에 닫혀있어 DIP를 잘 지키고 있어 좋은 디자인입니다.

**결론**

저의 의견으로는 2번째 다이어그램이 더 좋은 디자인이라고 생각합니다. 소프트웨어를 디자인 할떄, 업데이트, 수정 등의 변경은 자주, 많이 발생하게 됩니다. 따라서 모든 설계들은 확장성이 중요한 요소로서 동작하고 이는 OOP의 원칙인 SOLID 에서도 확인해볼 수 있는 중요한 사항입니다.

단기적으로 당장 눈앞의 기능들만을 고려했을 때 비교적 단순하고 알기 쉬운 첫번째 다이어그램이 분명 좋은 메리트를 가지고 있습니다.

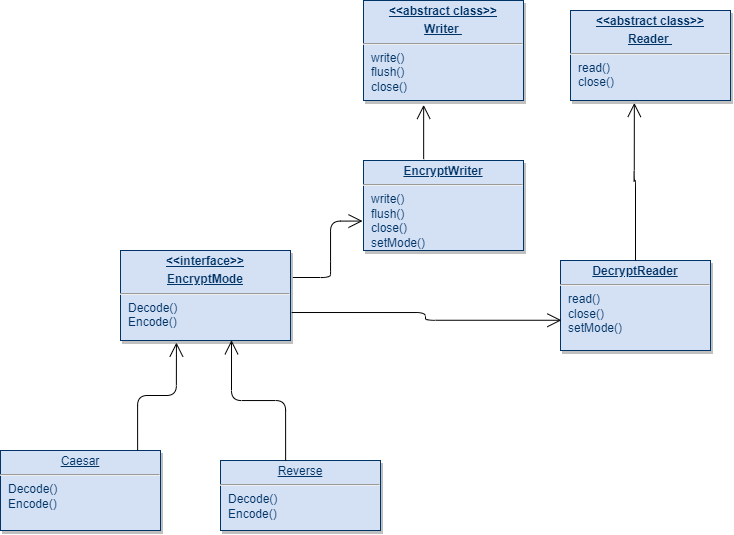
하지만 위의 다이어그램들이 당장 보여지는 기능에서 변경되고 확장되는 등 real-world situation으로 가져온다면, 두번째 다이어그램이 더 좋은 디자인이라고 생각합니다.

**HW 3-2**

HW3-2에서는 구현한 코드들에 대한 다이어그램과 간단한 설명을 작성하였습니다.

P2의 구조가 P1의 구조에서 전략 패턴을 추가한 버전으로 작성 되었기에, 둘의 다이어그램은 P2의 다이어그램으로 통일하여 설명하겠습니다.

**P1 & P2 diagran**



Strategy패턴으로 EncryptMode 인터페이스가 Caesar 혹은 Reverse 두가지 암호화 전략 중 선택 할 수 있게 구현하였습니다.

Decorator 패턴으로 EncryptWriter 사용해 Writer의 write 메소드를 꾸미고 EncryptWriter를 사용해 writer을 꾸몄습니다. 이를 사용하여 EncryWriter객체에서 setMode를 통해 암호화 전략을 선택하고, write 사용시 기존 write를 decorate하여 암호화 기능을 추가합니다.

그리고 DecryptReader를 사용하여 Reader의 read 메소드를 꾸몄습니다. 동일하게 setMode로 암호화전략을 선택 후, reade 사용 시 기존 read를 decorate하게됩니다.

테스트 코드의 경우 P1과 P2모두 Encrpy\_test, Decrypt\_test, all\_test 로 구성되어 있습니다.

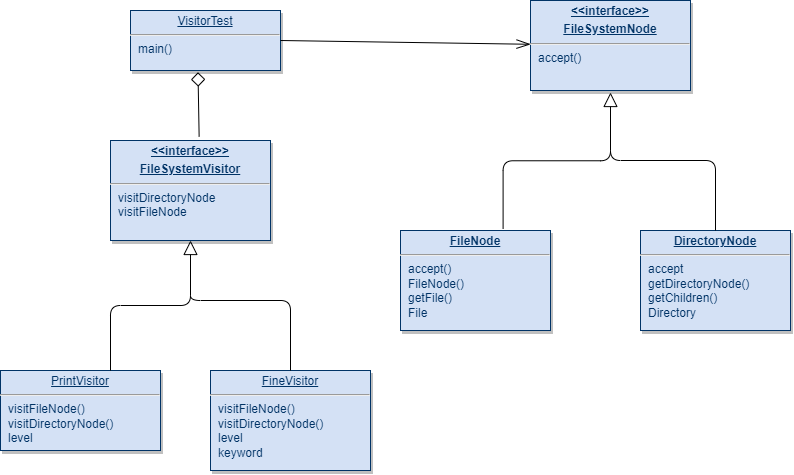
Encrpy\_test, Decrypt\_test는 각각 간단한 1개의 예시로 Encrypt, Decrypt 시에 올바른 결과가 나오나 check하는 테스트입니다. All\_test의 경우 P1 과 P2에서 차이가 있는데 P1의 경우

반복문에 따라 1~1000의 랜덤한 길이를 가지고 랜덤한 char을 추가하여 해당 문자열이

예시로 Encrypt, Decrypt을 수행한 뒤, 결과 값이 올바르게 초기 입력값으로 나오나 테스트 합니다.

P2의 경우 P1의 테스트 코드에 더해 홀수번째 반복문은 Reverse전략, 그리고 짝수번째 반복문은 Caesar 전략을 가지고 Encrypt, Decrypt을 수행한 뒤, 결과 값이 올바르게 초기 입력값으로 나오나 테스트 합니다.

**P3 Diagram**



FileSystemNode는 인터페이스로써, 파일 시스템에 있는 노드인 File 노드와 Directory 노드를 표현합니다. 그리고 VisitTest 로 이 두가지 클래스를 파일 노드와 디렉토리 노드를 담는 객체로써 사용하고, 이들을 통해 2가지 기능을 수행합니다. 이 기능들로는 printVisitor,findvisitor가 있습니다. 여기서 새로 구현한 findvisitor은 keyword를 받게되고, 이 keyword를 포함하는 파일, 디렉토리 명을 탐색합니다. 이번코드의 경우 내부 함수에서 System.out.Print 의 출력함수를 통해 파일명을 출력하는 구조이기에, 별도의 junit 테스트가 아닌, 주어진 VisitorTest에 keyword를 할당한뒤, 나온 출력값 중 제가 입력한 keyword가 포함되어있는지 출력값중에 확인하는 것으로 테스트 하였습니다.

**Refernece**

<https://ko.wikipedia.org/wiki/SOLID_(%EA%B0%9D%EC%B2%B4_%EC%A7%80%ED%96%A5_%EC%84%A4%EA%B3%84)#:~:text=SOLID%20%EC%9B%90%EC%B9%99%EB%93%A4%EC%9D%80%20%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4%20%EC%9E%91%EC%97%85,%EC%A0%84%EB%B0%98%EC%A0%81%20%EC%A0%84%EB%9E%B5%EC%9D%98%20%EC%9D%BC%EB%B6%80%EB%8B%A4>.

<https://ko.wikipedia.org/wiki/%ED%85%9C%ED%94%8C%EB%A6%BF_%EB%A9%94%EC%86%8C%EB%93%9C_%ED%8C%A8%ED%84%B4>