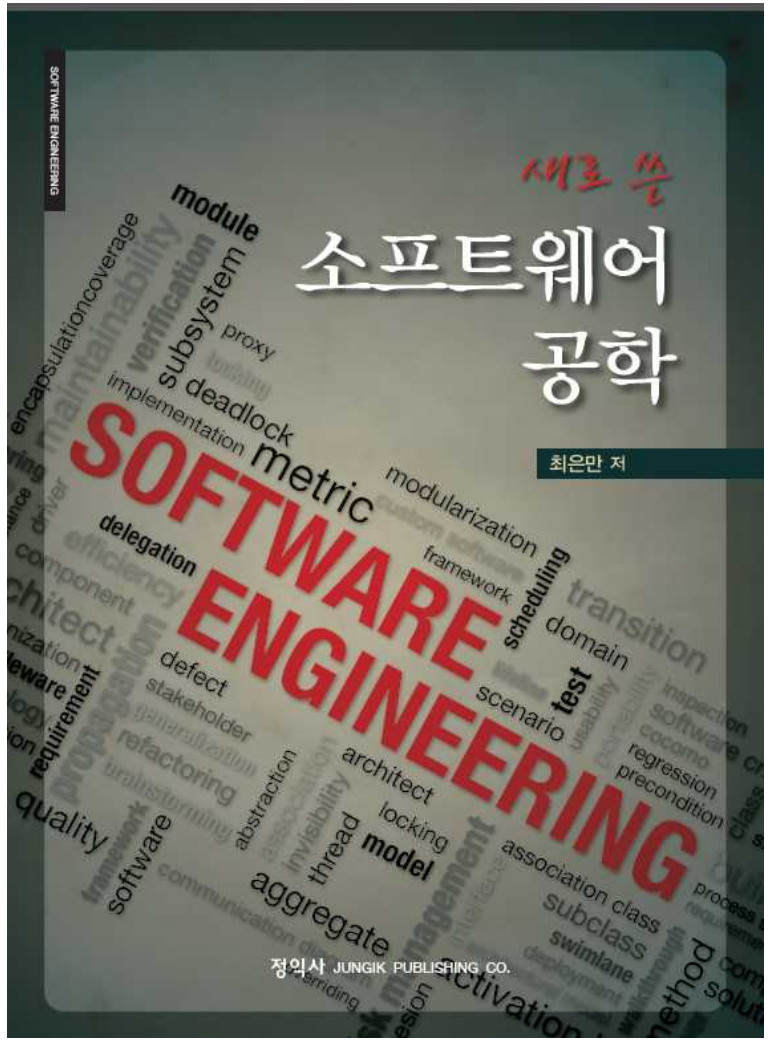
소프트웨어 공학 개론

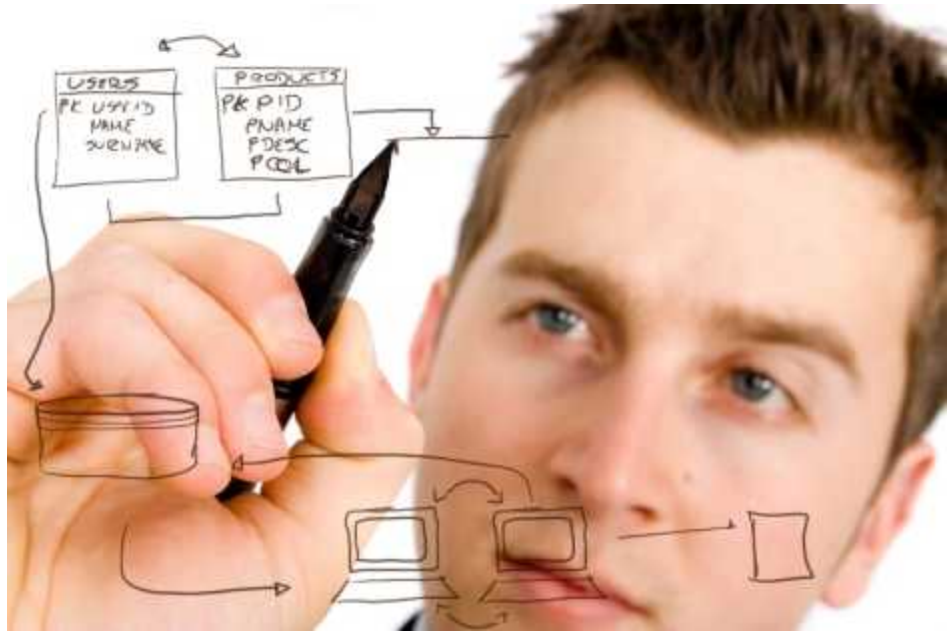


강의 10: 디자인 패턴

최은만
동국대학교 컴퓨터공학과

설계 작업은?

- 소프트웨어 설계는 어려운 일
  - 문제를 잘 분할하고
  - 유연하고 잘 모듈화 된 좋은 디자인이 되어야 함
- 설계는 시행 착오(trial and error)의 결과
  - 시행 착오로 얻은 지식
- 성공적인 설계가 존재
  - 두 설계가 똑 같은 일은 없음
  - 반복되는 특성



디자인 패턴

- 디자인 패턴이란?
  - 소프트웨어 설계에서 자주 발생하는 문제에 대한 일반적이고 반복적인 해결책을 말한다.
  - 여러 가지 상황에 적용될 수 있는 일종의 템플릿

- 디자인 패턴 구성 요소
  - 패턴의 이름과 구분
  - 문제 및 배경 – 패턴이 사용되는 분야 또는 배경
  - 솔루션 – 패턴을 이루는 요소들, 관계, 협동과정
  - 사례 – 적용 사례
  - 결과 – 패턴의 이점, 영향
  - 샘플 코드 – 예제 코드

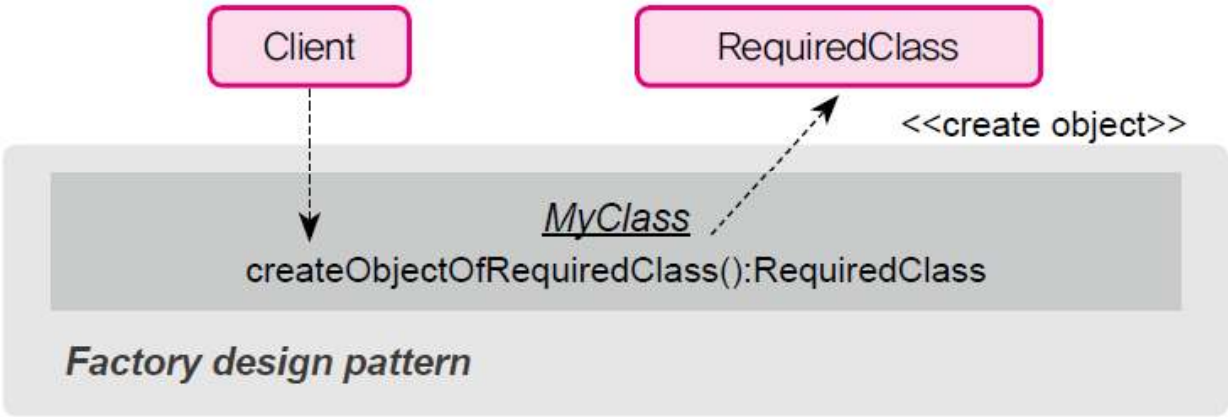
디자인 패턴

- 패턴의 분류
  - Gamma의 23개 패턴

		목적에 의한 분류		
		생성유형	구조적	행위적
범위	클래스	팩토리 메소드	어뎁터(클래스)	인터프리터 템플릿 메소드
	객체	추상 팩토리 싱글톤 프로토타입 빌더	어뎁터(객체) 브리지 컴포지트 데코레이터 퍼싸드 플라이웨이트 프록시	책임 체인 커맨드 반복자 중재자 메멘토 옵서버 상태 전략 비지터

팩토리 메소드 패턴

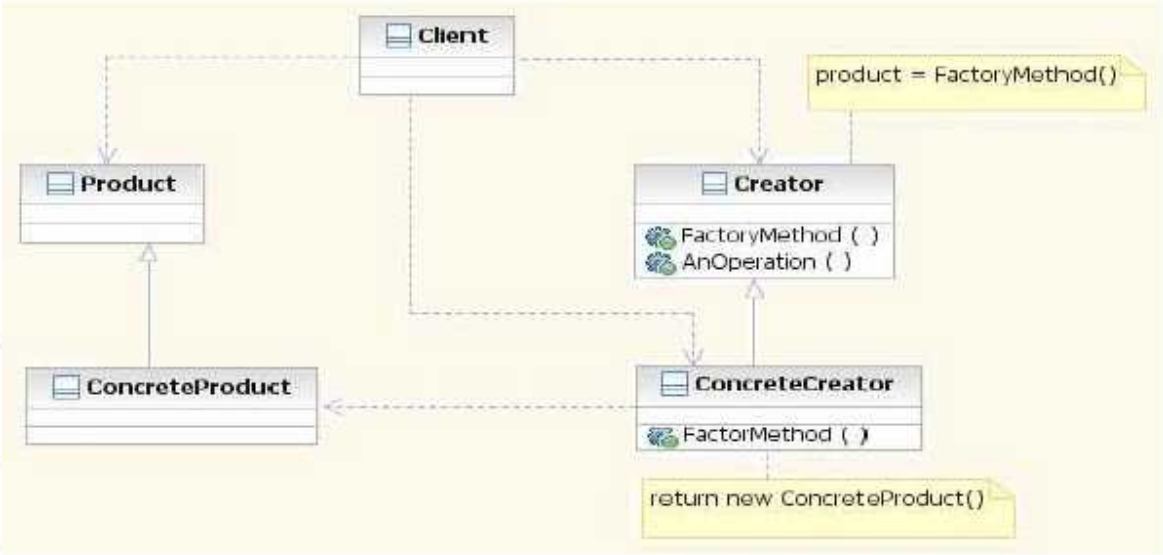
- 객체 생성을 위한 인터페이스의 정의
- 위임(delegation) 형태
  - RequiredClass의 생성을 팩토리 메소드를 가진 MyClass에게 위임
  - 팩토리 메소드는 createObjectOfrequiredClass()



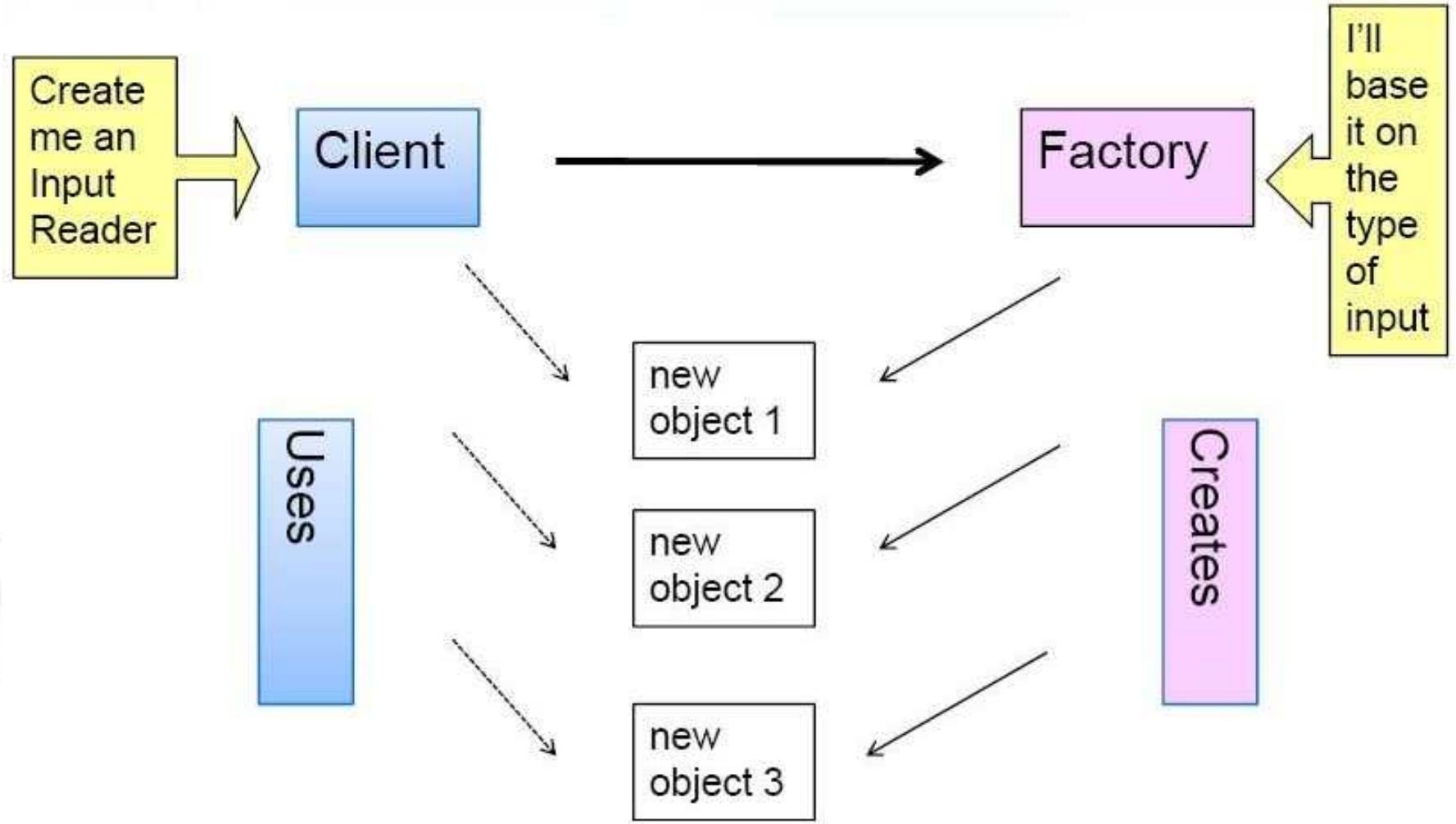
- 사용 이유
  - 베이스 클래스에 속하는 객체 중 하나가 필요한데 그 아래에 있는 자식 객체 중 어떤 것이 필요한지 실행 시간까지 알 수 없을 때

팩토리 메소드 패턴

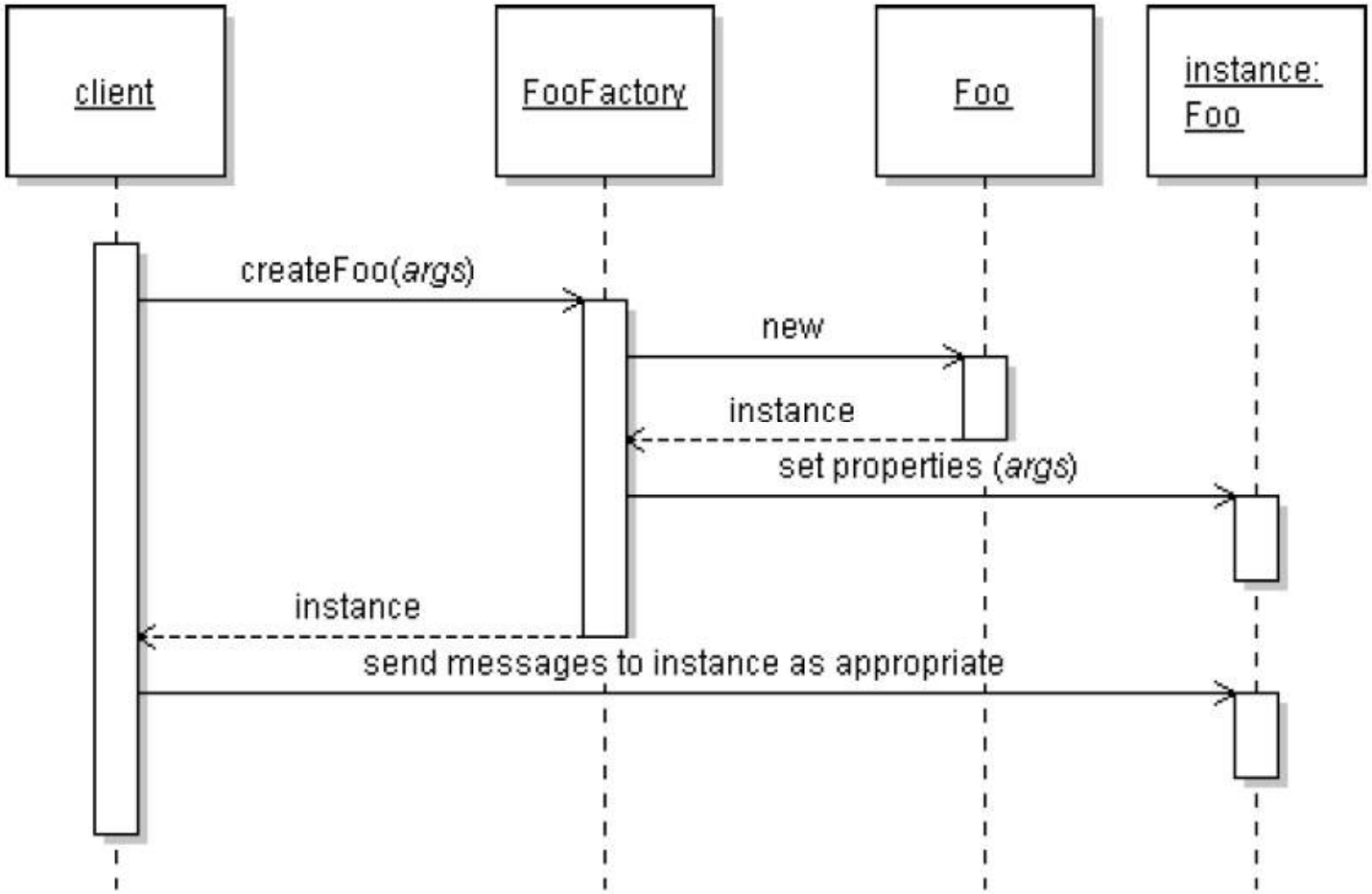
- 팩토리 메소드
  - 다른 클래스의 인스턴스를 쉽게 생성하고 리턴하는 임무를 가진 메소드
  - 생성자를 부르는 대신 팩토리 메소드를 사용하여 객체를 셋업
  - 구축 정보를 사용 정보에서 분리(응집을 높이고 결합을 약하게 하기 위하여)하여 객체의 생성과 관리를 쉽게
  - 서브 클래스의 인스턴스화를 지연하는 효과



사용과 생성을 분리

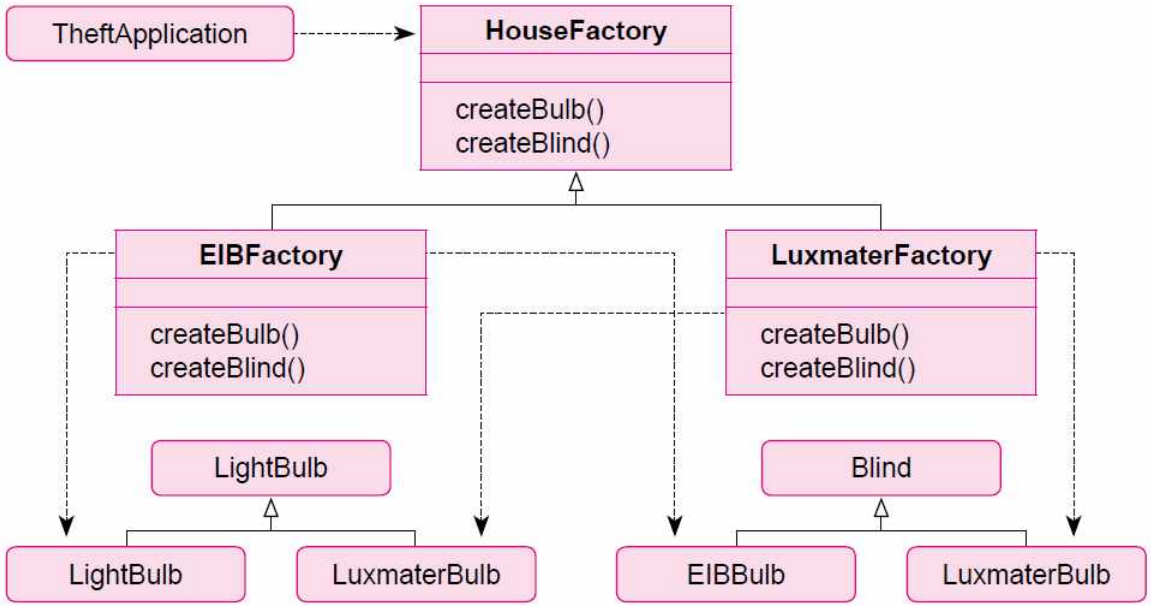

Create me an Input Reader
Client
Factory
I'll base it on the type of input
new object 1
new object 2
new object 3
Uses
Creates

팩토리 순서 다이어그램


client
FooFactory
Foo
instance:
Foo
createFoo(args)
new
instance
set properties (args)
instance
send messages to instance as appropriate

추상 팩토리 패턴

- 클래스의 집합의 종류를 지정하여 관련된 객체의 집합을 생성할 수 있게 하는 패턴
- 추상 팩토리 패턴이 필요한 경우
  - 하나의 업체에서 가전제품을 생산하기 위한 하드웨어를 제공함에도 불구하고 호환성이 취약하다. 여러 생산자가 만든 장치들을 혼합하여 사용할 때 취합하기 어렵다.
- 추상 팩토리 적용 사례



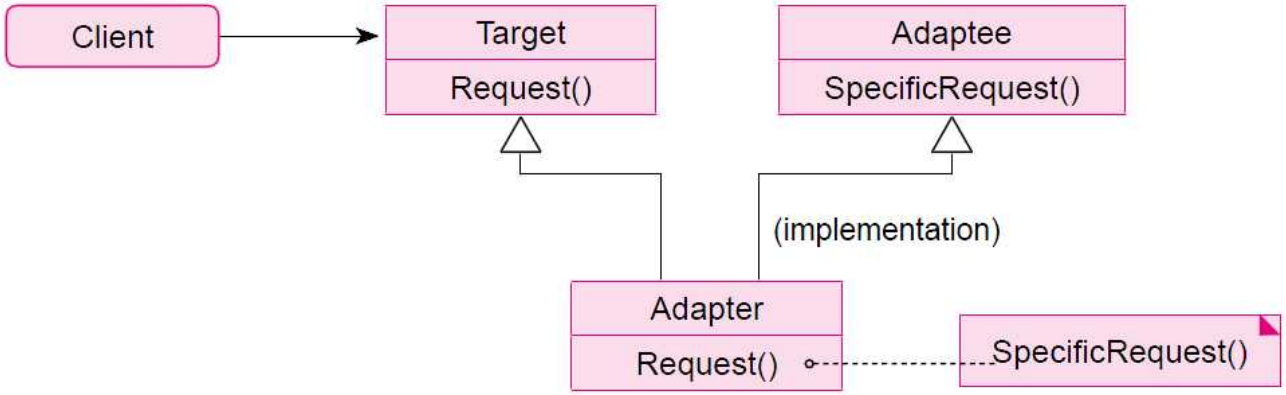
어댑터 패턴

- 이미 개발된 클래스, 즉 레거시 시스템의 인터페이스를 다른 클래스의 요구에 맞게 **인터페이스를 변환**해주는 것

- 이미 만들어져 있는 클래스를 사용하고 싶지만 인터페이스가 원하는 방식과 일치하지 않을 때 사용

- 패턴 적용 방법
  - 위임을 이용
  - 상속을 이용

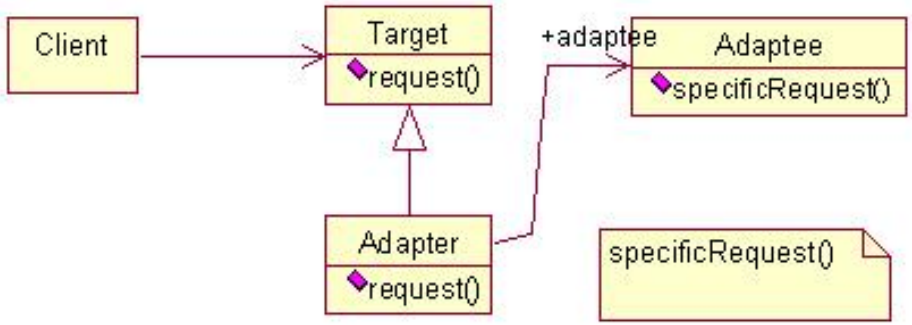
<상속 이용 예>



어댑터 패턴

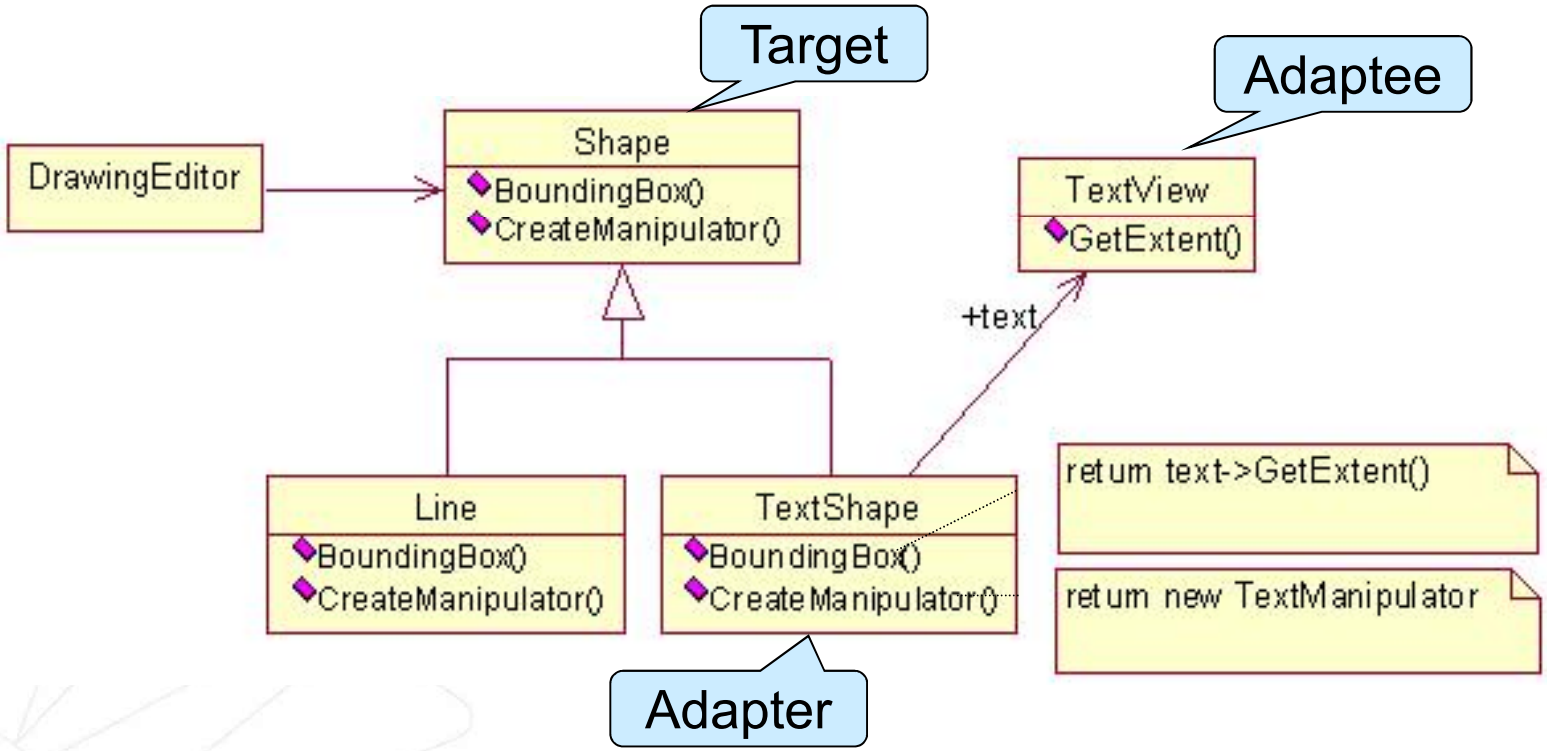
- 특정 클래스의 인터페이스를 클라이언트가 원하는 인터페이스로 변환

- 적용
  - 기존에 존재하는 클래스를 재사용하고 싶지만, 원하는 인터페이스와 맞지 않을 때
  - 미리 정해져 있지 않은 클래스들과 상호 작용할 수 있는 재사용 가능 클래스를 만들려는 경우



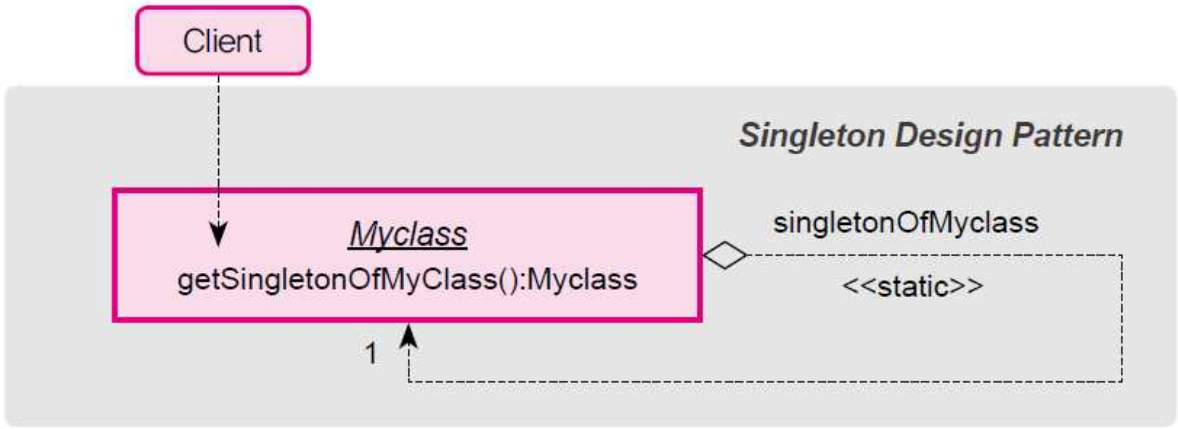
어댑터 패턴 사례

- 그래픽 에디터 프로그램
  - 그래픽 객체로 Line 과 Text 가 존재
  - TextShape 구현이 복잡한 관계로 기존에 존재하는 TextView 클래스 사용



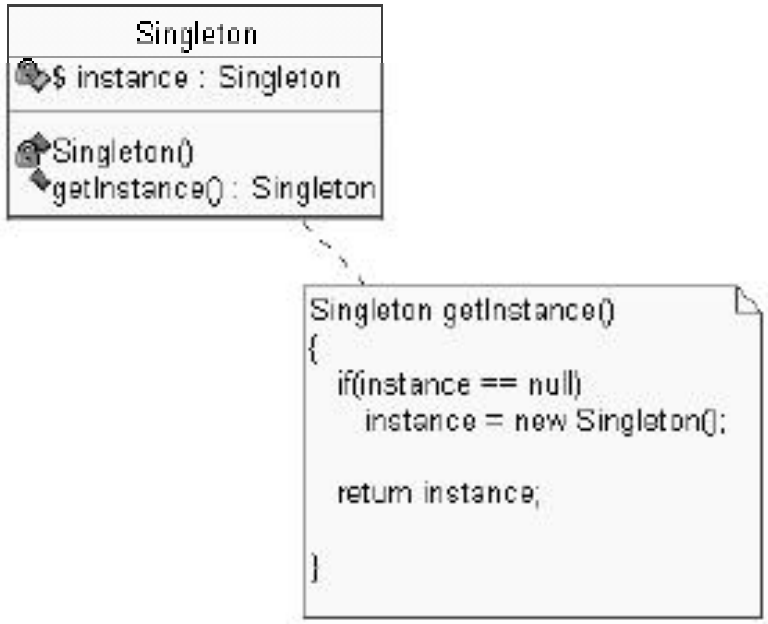
싱글톤 패턴

- 시스템에서 단 하나의 인스턴스만을 갖도록 할 필요가 있는 경우 사용
  - 프린터 스풀러
  - 파일 시스템 등
- 패턴 사용 요령
  - 클래스의 유일한 인스턴스를 넣을 위치를 정한다.
  - 생성자의 접근 수정함수를 private로 지정
    - 싱글톤을 얻기 위한 클래스 S에 public static 접근 메소드를 포함한다.
  - 싱글톤을 얻는 방법은 생성자에게 위임



싱글톤 패턴

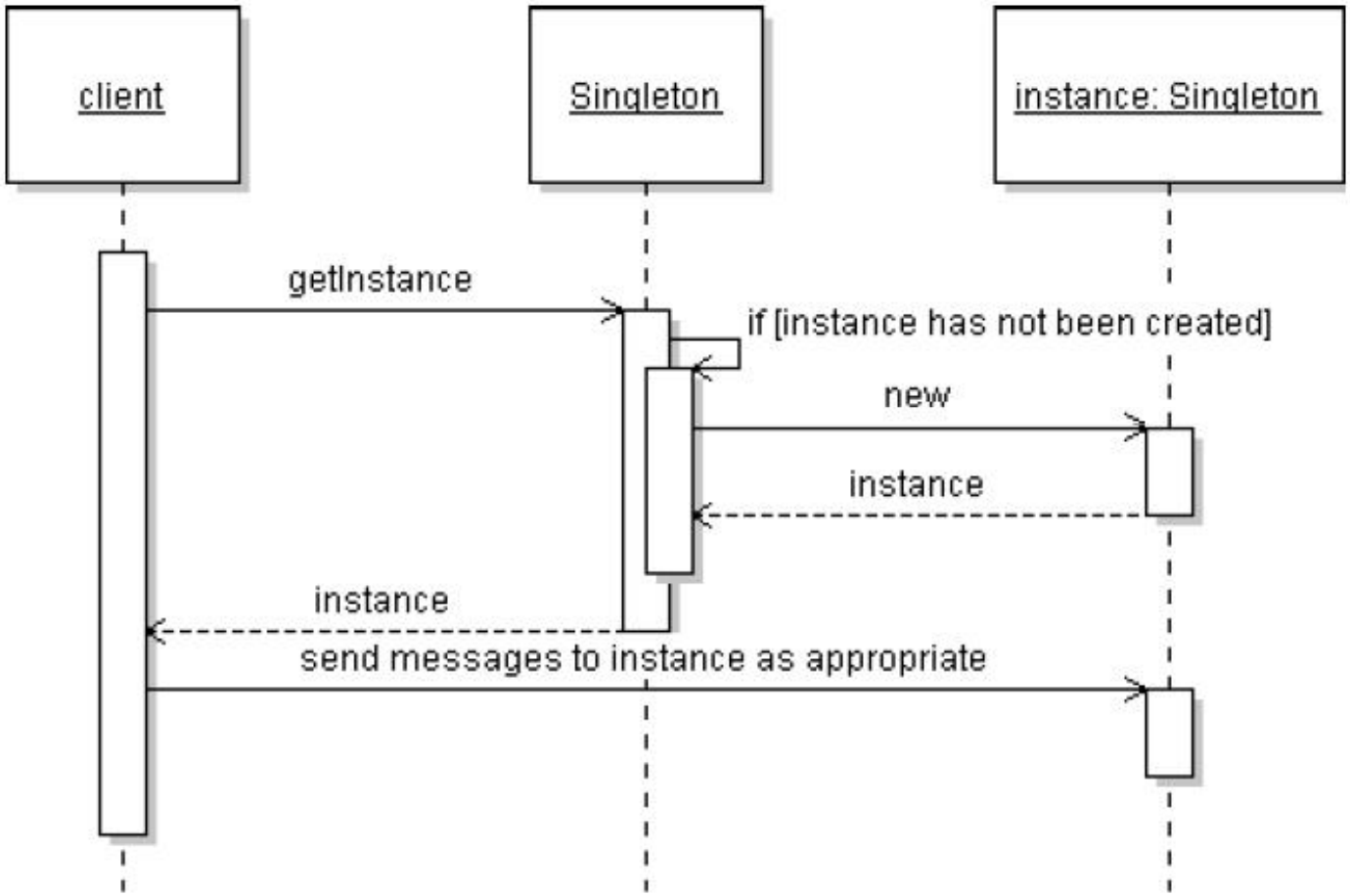
- 싱글톤: 그 타입의 유일한 객체
  - 많아야 하나의 인스턴스를 가짐을 보장
  - 인스턴스에 대하여 어디서든 접근할 수 있게 하여야
  - 프로그래머가 인스턴스를 없애버리는(또는 더 생성할) 관리 책무는 빼앗음
  - 사용자에게 유일한 인스턴스를 접근할 수 있는 메소드를 제공



싱글톤 패턴의 구현

- 생성자를 밖에서 부르지 못하도록 private으로 만든다.
- 클래스 안에 클래스의 인스턴스를 static private 으로 선언
- 단일 인스턴스를 접근할 수 있는 public getInstance()나 유사 메소드를 둔다.
  - 이 메소드는 다중 스레드로도 실행될 수 있기 때문에 보호되어야 하고 동기화 되어야

싱글톤 시퀀스 다이어그램



싱글톤 구현 예 #1

- 난수를 만들어 내는 RandomGenerator를 싱글톤으로 만들어 보자

```
public class RandomGenerator {
        private static RandomGenerator gen = new
                RandomGenerator();
        public static RandomGenerator getInstance() {
                return gen;
        }
        private RandomGenerator() {}
        ...
}
```

이 프로그림의 문제점은?

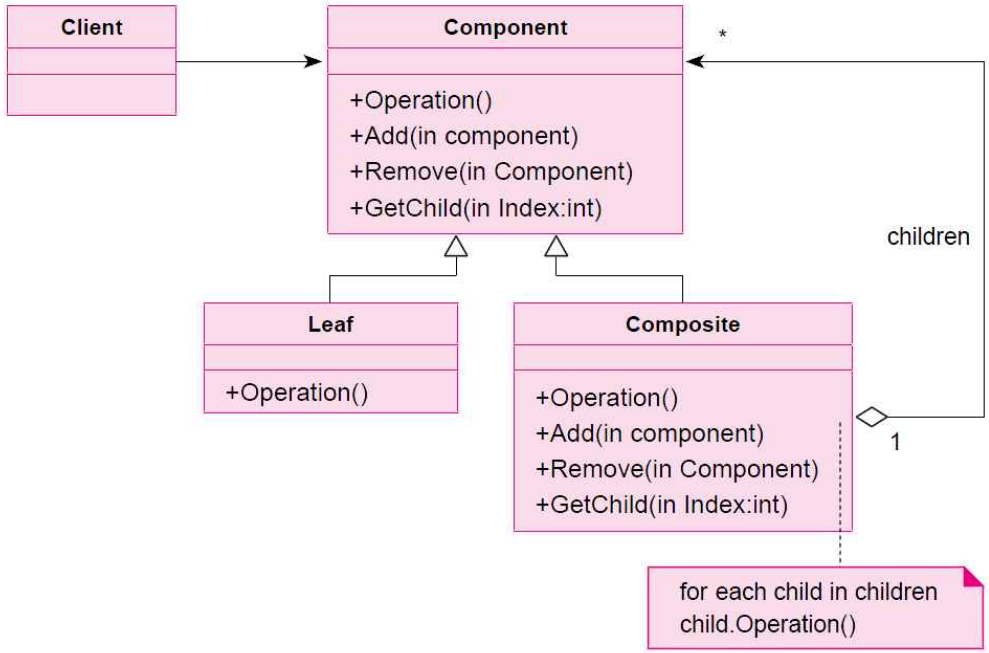
싱글톤 구현 예 #2

- 필요할 때까지는 객체를 만들지 않는다.

```
public class RandomGenerator {
      private static RandomGenerator gen = null;
      public static RandomGenerator getInstance() {
            if (gen == null) {
              gen = new RandomGenerator();
            }
            return gen;
      }
      ...
}
```

컴포지트 패턴

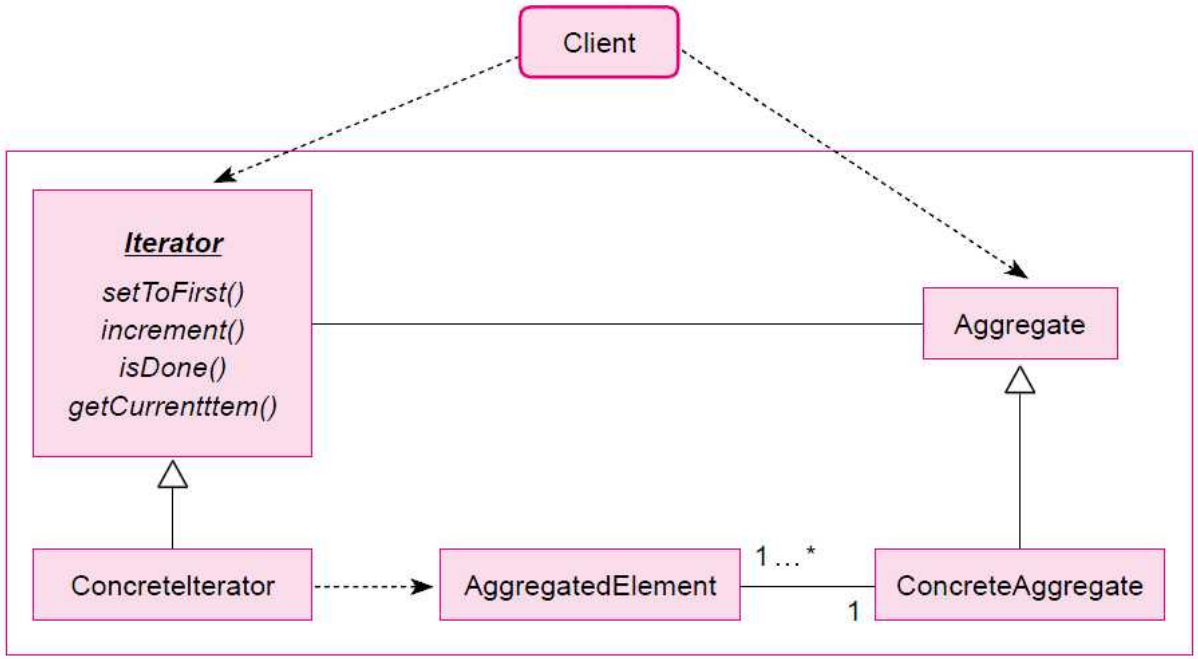
- 객체 집합 속에 또 다시 객체 집합을 갖는 경우 사용
- 집합 속에 포함될 객체와 집합을 가지고 있는 객체, 이들 모두가 자기 자신과 동일한 타입(메소드와 데이터)의 객체 리스트를 가질 수 있도록
- 기본 클래스와 이를 포함하는 컨테이너 클래스를 구분하지 않고 처리하는 재귀적 합성을 이용할 수 있다.



반복자 패턴

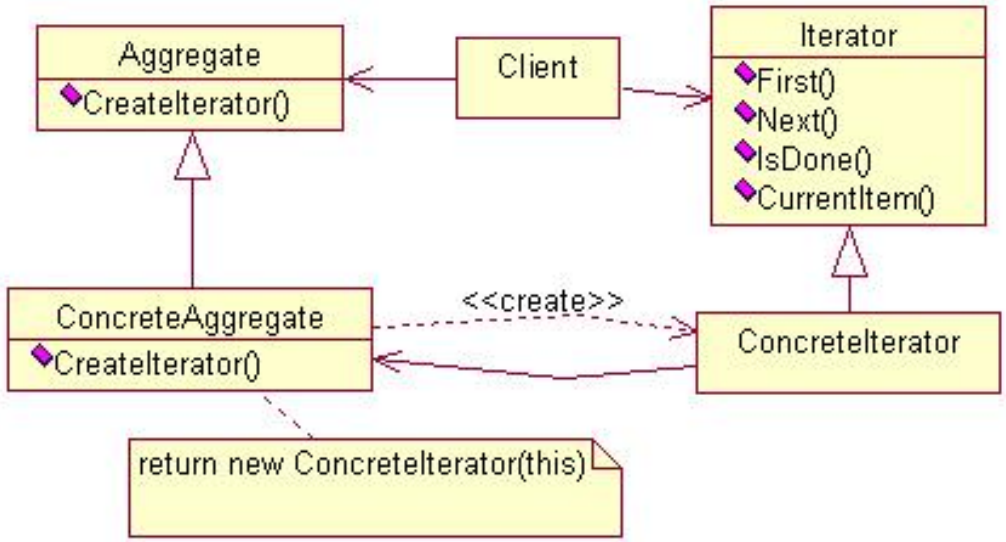
- 시스템의 유사한 객체를 다룰 때, 동일한 인터페이스를 이용하여 접근할 수 있도록 만드는 패턴

- 처리하려는 자료구조가 다른 형태로 바뀌더라도 클라이언트가 영향을 받지 않음



반복자 패턴

- 반복자
  - 집합에 포함된 객체를 검사하여 반복하는 일을 할 수 있도록 표준 방법을 제공하는 객체
- 장점:
  - 클라이언트 자세한 표현 방법을 몰라도 됨
  - 접근 인터페이스의 단순화

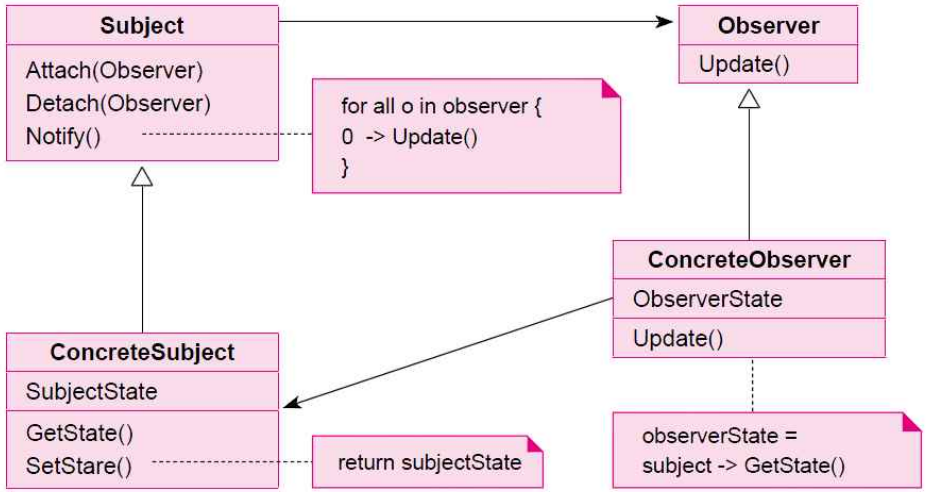


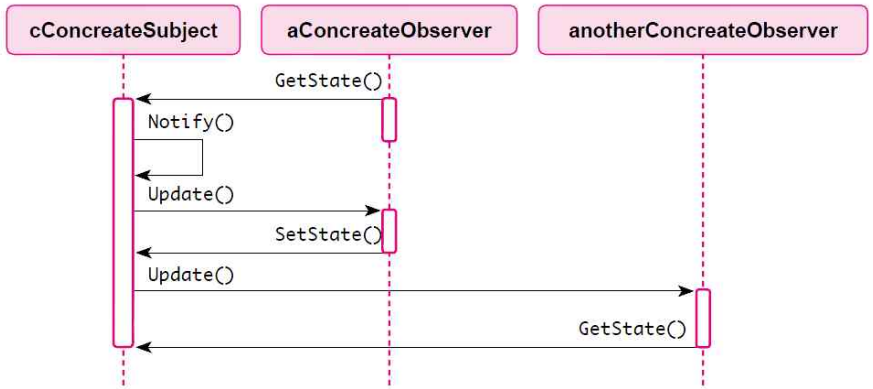
반복자의 구현

```
class List {
  public:
        int size() {…}
        boolean isEmpty() {…}
        ListElement* get(int index) {…}
}
public class ListIterator {
        int currentIndex;
  public:
        boolean hasNext() {…}
        ListElement* first() {…}
        ListElement* next() {…}
        ListElement* current() {…}
}
```

옵서버 패턴

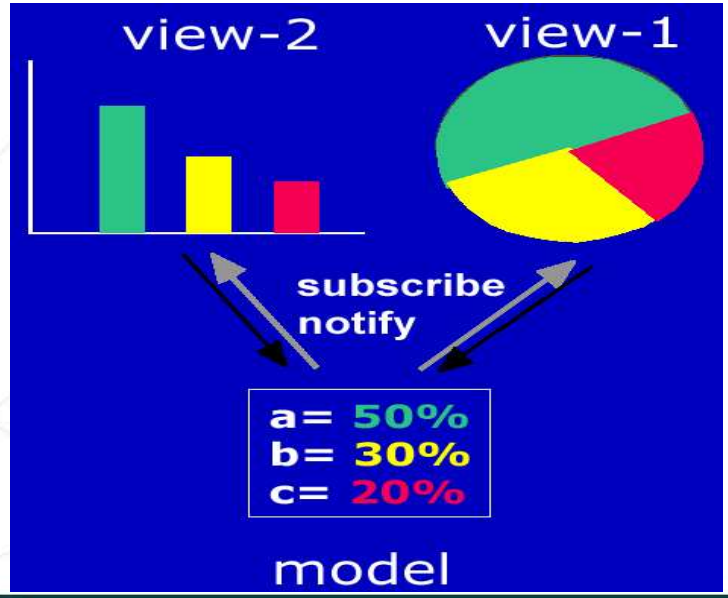
- 1대 다의 객체 의존관계를 정의한 것
- 한 객체가 상태를 변화시켰을 때 의존 관계에 있는 다른 객체들에게 자동적으로 통지하고 변경
- 객체 하나를 변경하였을 때, 다른 객체에 통보하여 갱신하기 위한 옵저버 객체를 둔다.





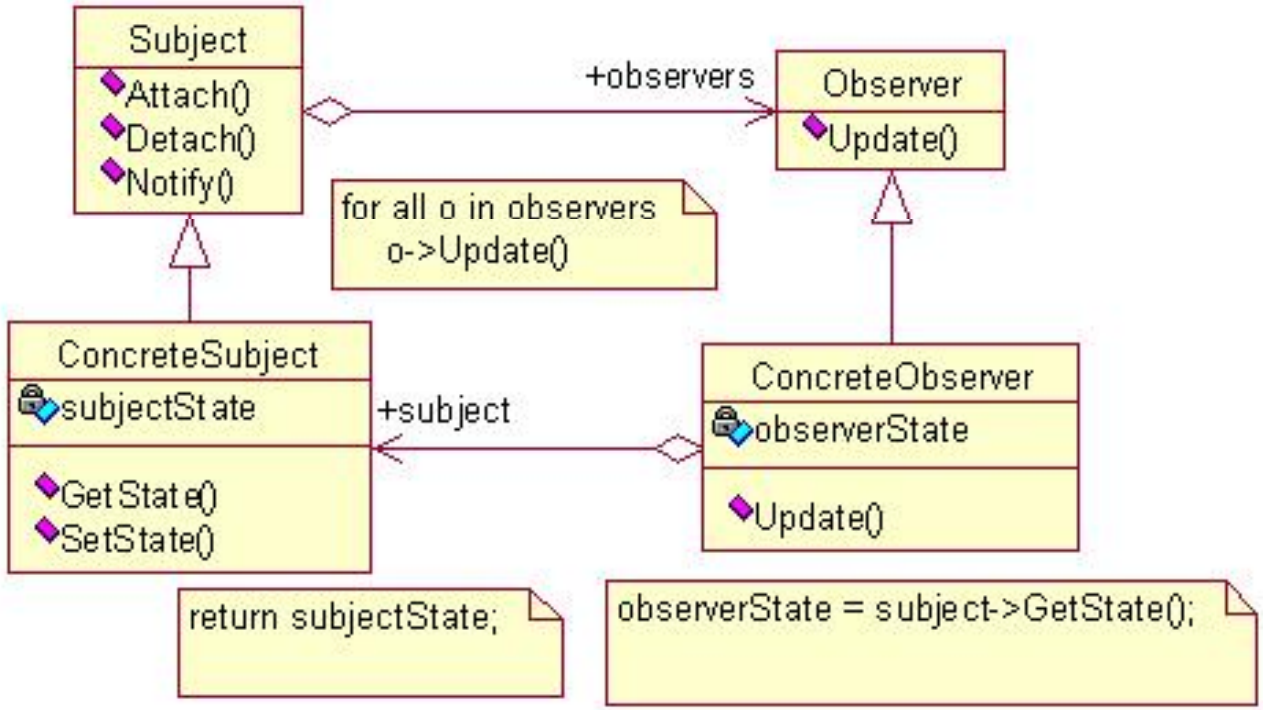
옵서버 패턴

- 뷰(옵서버)가 모델(데이터)안의 값에 영향을 받는 경우
- 두 가지 방법의 절충
  - 모델이 변경되면 모든 뷰에게 알려주는 방법(push)
  - 모델의 값이 변경되었는지 뷰가 계속 알아보는 방법(polling)
- 서로 독립성을 유지하며 효율적으로 협력하는 방법



옵서버 패턴

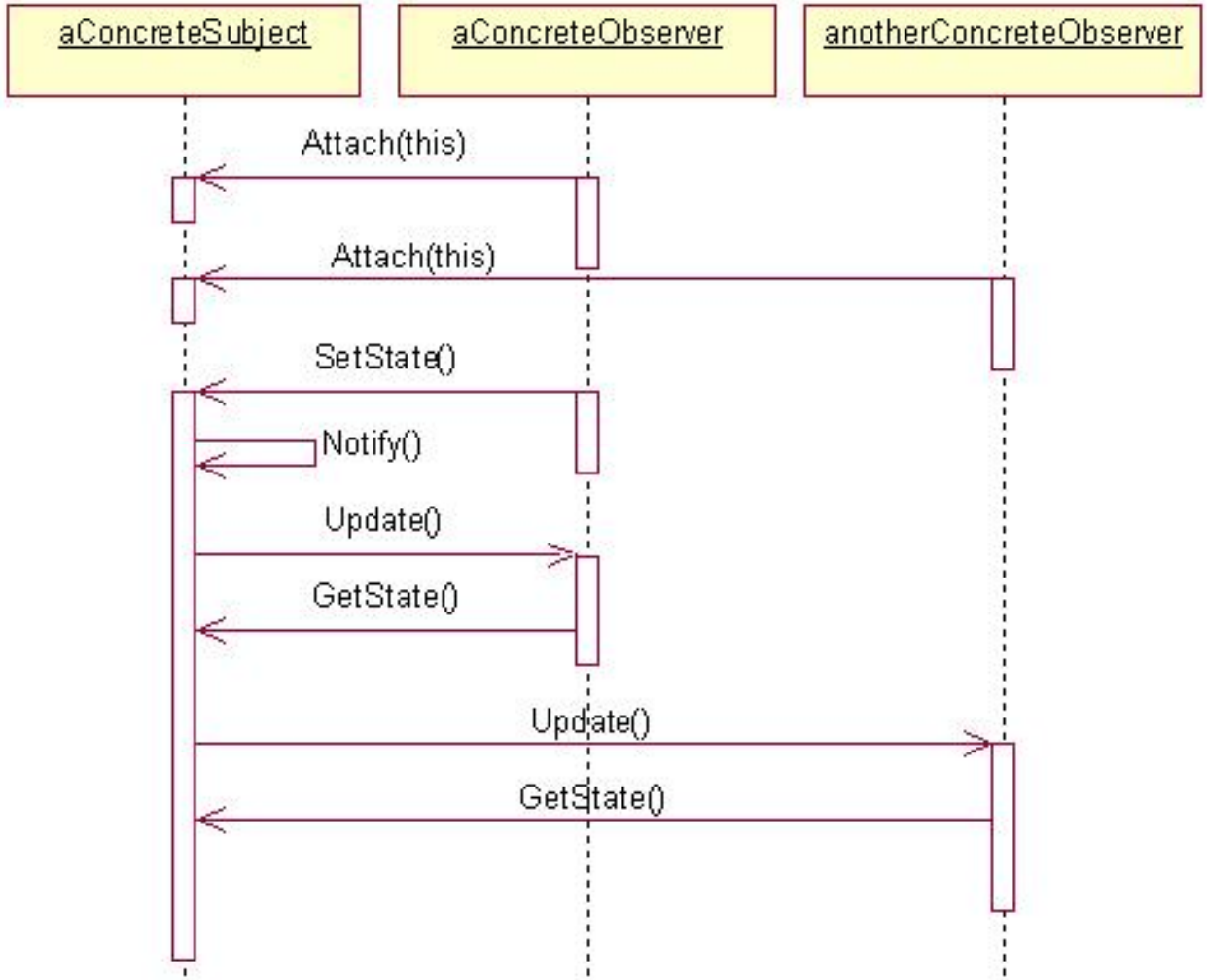
- Subject의 값이 변하면 옵서버에게 통지
- 통지 받은 옵서버는 Subject의 값을 접근하여 받아옴



옵서버 패턴

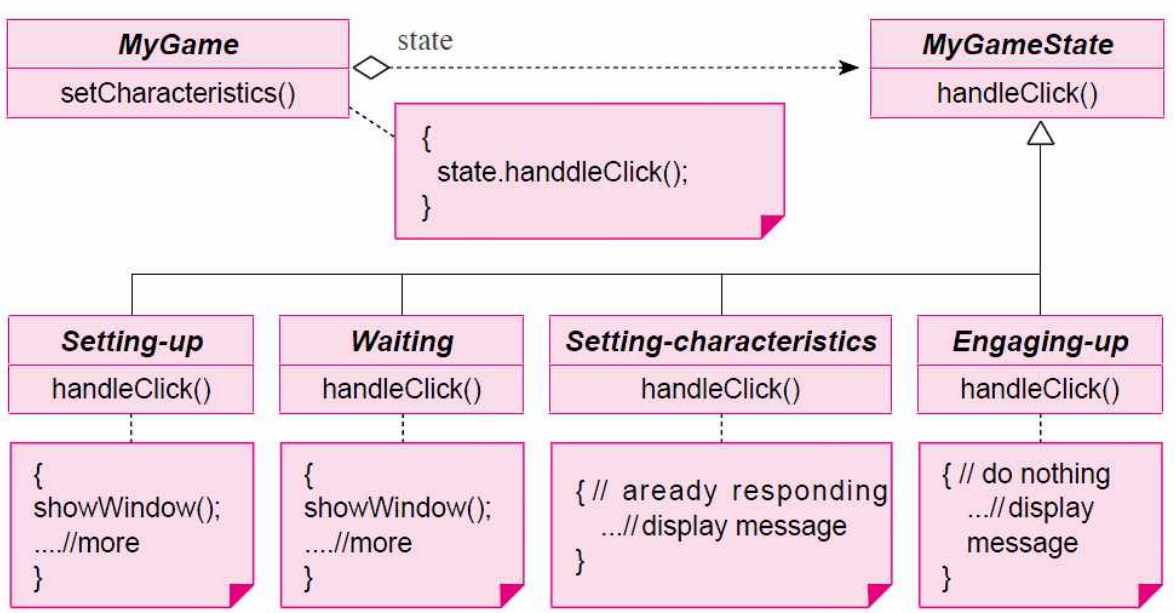
- Subject 객체와 옵서버 객체 간의 coupling을 줄일 수 있음
  - Subject 객체는 단지 Observer 객체 list를 가지고 있다는 정도만 알면 된다.
- Subject class와 Observer class가 서로 독립적으로 변경 및 확장 가능
- 한 객체에 의하여 영향 받는 객체를 쉽게 정리할 수 있음

옵서버 패턴


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
Attach(this)
Attach(this)
SetState()
Notify()
Update()
GetState()
Update()
GetState()

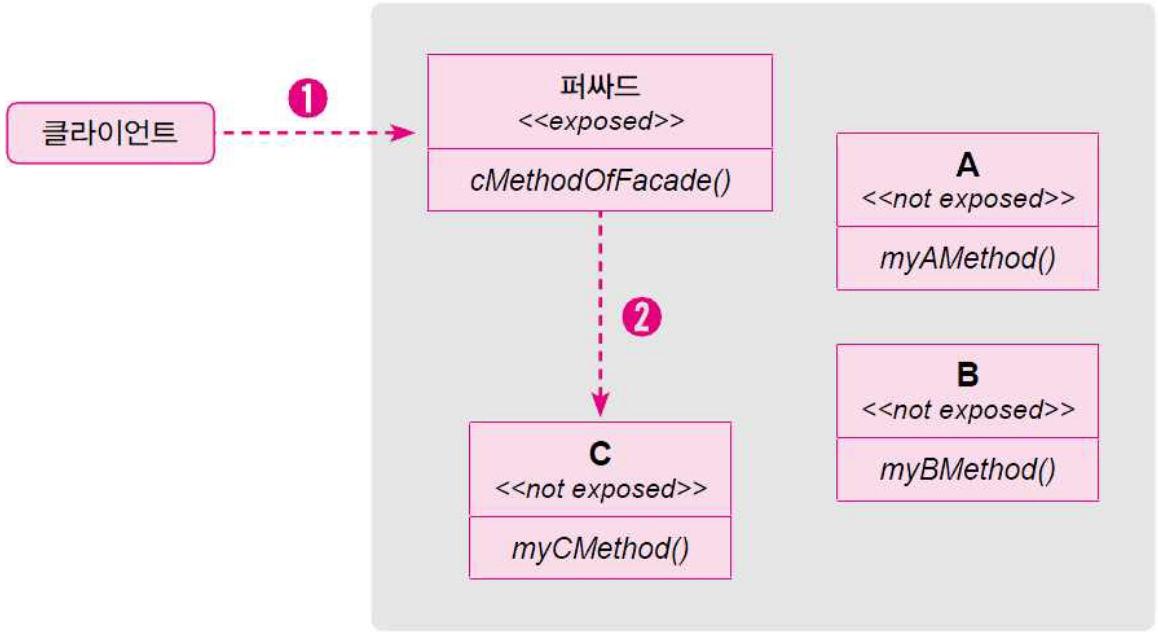
상태 패턴

- 이벤트 의존 애플리케이션에 적합한 패턴
- 시스템이 동작하면서 발생되는 이벤트에 따라 변경이 이루어 질 수 있도록 설계
- 롤플레잉 게임 예제
  - MyGameState 타입의 state를 가지고 있다.
  - State 는 다형성이 적용됨. 상태 객체를 이용



퍼싸드 패턴

- 서브시스템의 내부가 복잡하여 클라이언트 코드가 사용하기 힘들 때 사용
  - 간단한 인터페이스만 알아도 서브시스템 주요 기능을 사용
  - 복잡한 것을 단순하게 보여줌
  - 내부 시스템을 몰라도 사용 가능



디자인 패턴의 선택

- 디자인 패턴이 주어진 문제를 어떻게 해결하고 있는지 스터디
  - 먼저 디자인 패턴을 잘 숙지
- 주어진 상황에 제일 잘 맞는 패턴이 무엇인지 숙고
  - 생성
  - 구조
  - 행위
- 시스템의 변경, 발전, 재사용 어느 측면이 유력한지 고려하여 적용

Questions?