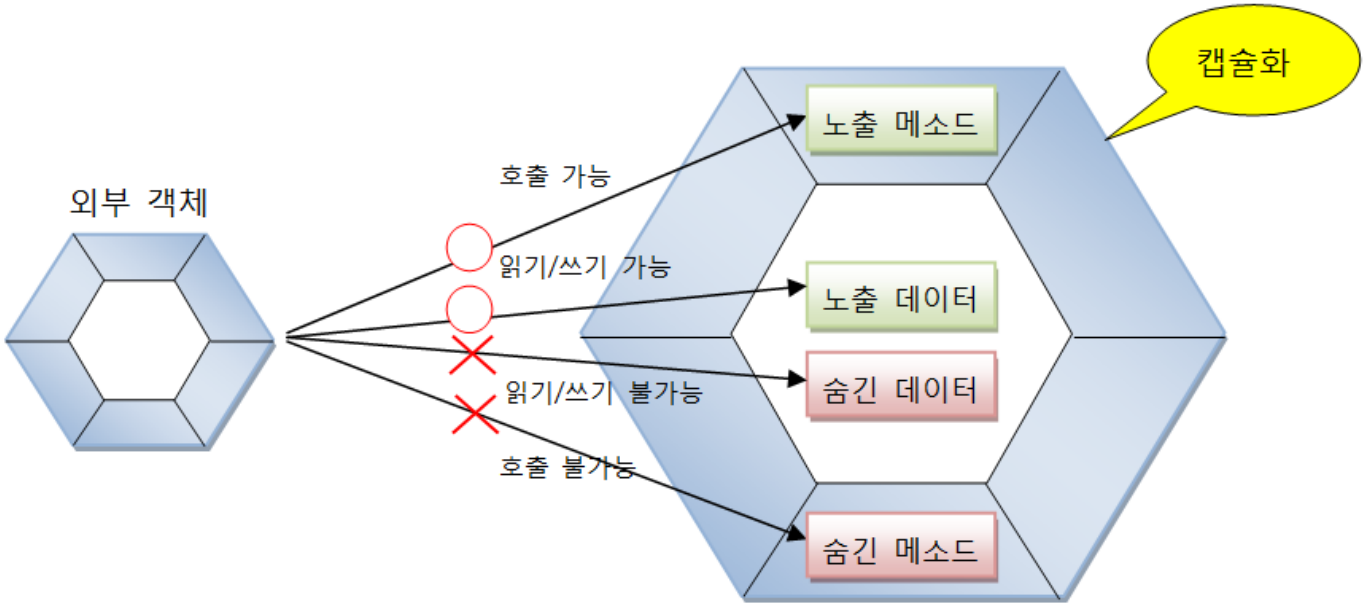
한림대학교 SW중심대학

클래스 다이어그램

- **캡슐화**
  - 객체의 필드, 메소드를 하나로 묶고, 실제 구현 내용을 감추는 것
  - 필드와 메소드를 캡슐화하여 보호하는 이유는 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록



잘못된 경우

```
public class UserInvalid {

    public String id="orange";
    public String password="banana";

}
```

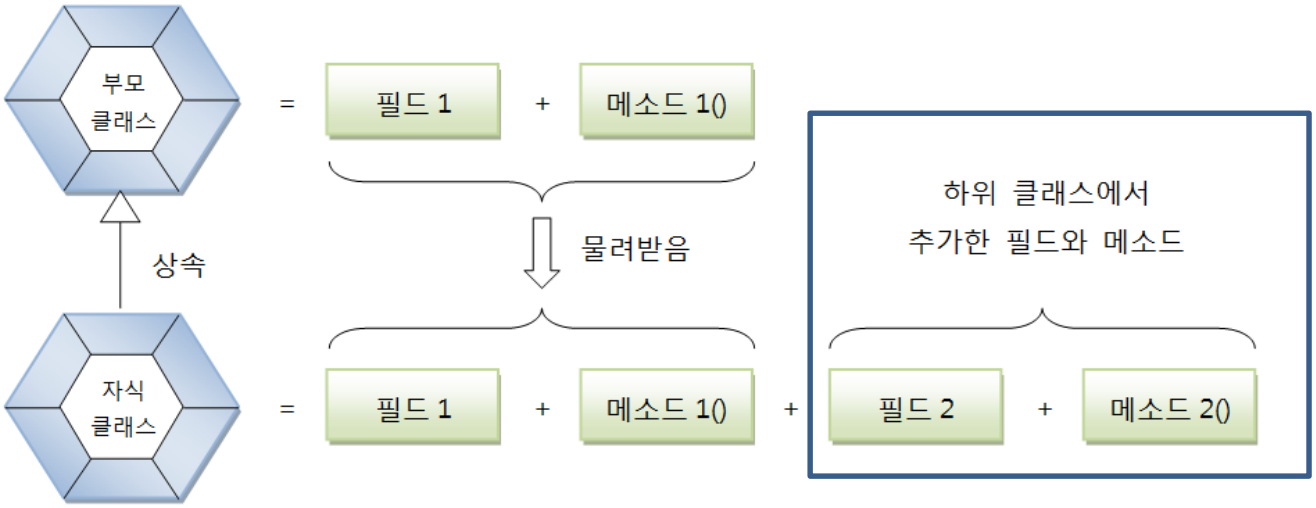
```
public class CapsuleMain {

    public static void main(String[] args) {
        UserInvalid uiv = new UserInvalid();
        uiv.password = "apple";
        System.out.println(uiv.password);
    }

}
```

올바른 경우

```
public class User {
    private String id;
    private String password;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

}
```

```
public class CapsuleMain {

    public static void main(String[] args) {

        UserValid uv = new UserValid();
        uv.setPassword("apple");
        System.out.println(uv.getPassword());

    }
}
```

- 상속
  - 상위(부모) 객체의 필드와 메소드를 하위(자식) 객체에게 물려주는 행위



슈퍼 클래스

```
public class BasicCalculator {

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public int mul(int x, int y) {
        return x*y;
    }

}
```

서브 클래스

```
public class EngineerCalculator extends BasicCalculator{

    public double convertToLog10(double num) {
        return Math.log(num);
    }

}
```

```
public class InheritMain {

    public static void main(String[] args) {
        EngineerCalculator ec = new EngineerCalculator();
        int  result =ec.add(1, 2);
        System.out.println(result);

    }

}
```

- **다형성**
  - 같은 이름의 메소드가 클래스 혹은 객체에 따라 다르게 구현되는 것
  - 메소드 오버라이딩

슈퍼 클래스

```
public class BasicCalculator {

    public int add(int x, int y) {
        System.out.println("BasicCalculator add");
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public int mul(int x, int y) {
        return x*y;
    }
}
```

서브 클래스

```
public class EngineerCalculator extends BasicCalculator{

    public double convertToLog10(double num) {
        return Math.log(num);
    }
    public int add(int x, int y) {
        int res=x+y;
        System.out.println("EngineerCalculator add : "+res);
        return res;
    }

}
```

```
public class PolymorphismMain {
    public static void main(String[] args) {
        EngineerCalculator ec = new EngineerCalculator();
        int  result =ec.add(1, 2);
        System.out.println(result);
        result =ec.sub(1, 2);
        System.out.println(result);
    }
}
```

결과

```
EngineerCalculator add : 3
3
-1
```

클래스의 구성 요소

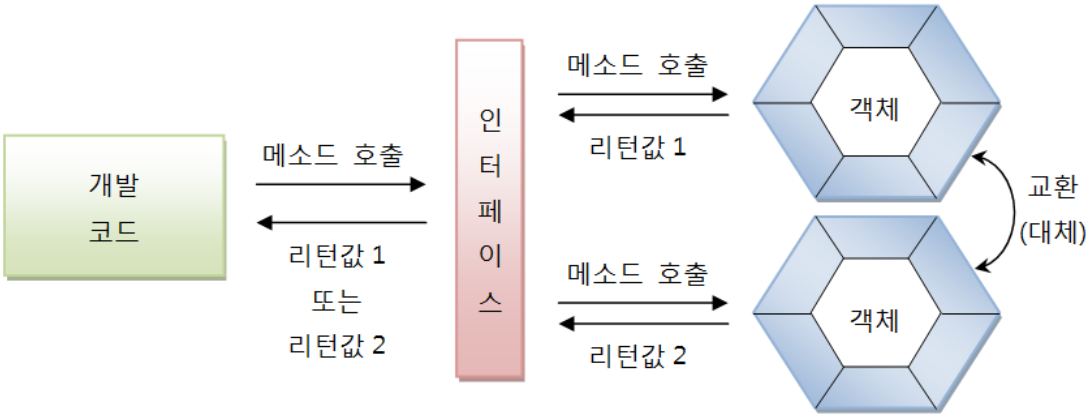
- 클래스는 클래스 이름, 생성자, 속성, 메서드로 구성
  - 클래스class
    - 공통의 속성, 메서드(오퍼레이션), 관계, 의미를 공유하는 객체 집합에 대한 기술
  - 속성attribute
    - 클래스의 구조적 특성에 이름을 붙인 것
    - 구조적 특성에 해당하는 인스턴스가 보유할 수 있는 값의 범위를 기술
    - 영문 소문자로 시작함
  - 메서드method
    - 오퍼레이션이라고도 함
    - 이름, 타입, 매개변수들과 연관된 행위를 호출할 때 제약사항이 요구되는데, 이 제약사항을 명세하는 클래스의 행위적 특징

Abstract class

- 구현 클래스 설계 규격을 만들고자 할 때
  - 구현 클래스가 가져야 할 필드와 메소드를 추상 클래스에 미리 정의
  - 구현 클래스의 공통된 필드와 메소드의 이름 통일할 목적
- 전체 기능중 일부 기능이 달라질수 있을 경우

Interface

- 개발 코드와 객체가 서로 통신하는 접점(형식을 약속)
  - 개발 코드는 인터페이스의 메소드만 알고 있으면 OK



class

```
public class Calculator {

    public String calName;

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public int mul(int x, int y) {
        return x*y;
    }

}
```

Abstract class

```
public abstract class Calculator {

    public String calName;

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public abstract int mul(int x, int y);

}
```

interface

```
public interface Calculator {

    public int add(int x, int y) ;
    public int sub(int x, int y) ;
    public int div(int x, int y);
    public int mul(int x, int y) ;

}
```

객체지향 프로그래밍

```
public interface RemoteControl {

    void turnOn();
    void turnOff();
    void setVolume(int volume);

}
```

```
public class SKRemoteControlImpl implements RemoteControl {

    private int volume;

    public void turnOn() {
        System.out.println("SK turnOn");
    }
    public void turnOff() {
        System.out.println("SK turnOff");
    }
    public void setVolume(int volume) {

        System.out.println("SK volume: " + volume);
    }
}
```

```
public class LGRemoteControlImpl implements RemoteControl {

    private int volume;

    public void turnOn() {
        System.out.println("LG turnOn");
    }
    public void turnOff() {
        System.out.println("LG turnOff");
    }
    public void setVolume(int volume) {

        System.out.println("LG volume: " + volume);
    }
}
```

```
public class RemoteControlExample {
    public static void main(String[ ] args) {
        RemoteControl rc = new LGRemoteControlImpl();
//      RemoteControl rc = new SKRemoteControlImpl();
        rc.turnOn();
        rc.turnOff();
        rc.setVolume(1);
    }
}
```

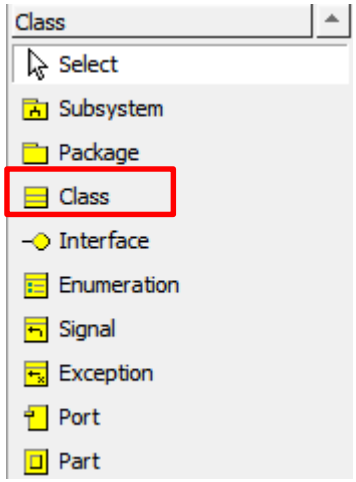
- **객체**
  - 붕어빵 기계 = 클래스 / 붕어빵 = 객체
  - 객체는 클래스의 인스턴스

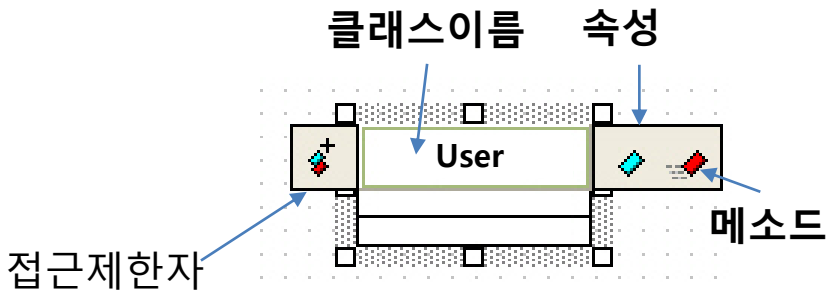
```
public class User {
    private String id;
    private String password;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

}
```

```
public class ObjectMain {

    public static void main(String[] args) {

        User pApple = new User();
        pApple.setPassword("apple");
        System.out.println(pApple.getPassword());

        User pMango = new User();
        pMango.setPassword("mango");
        System.out.println(pMango.getPassword());

    }
}
```

1. 클래스 다이어그램의 구성요소와 표현

- 클래스의 구성 요소의 표현





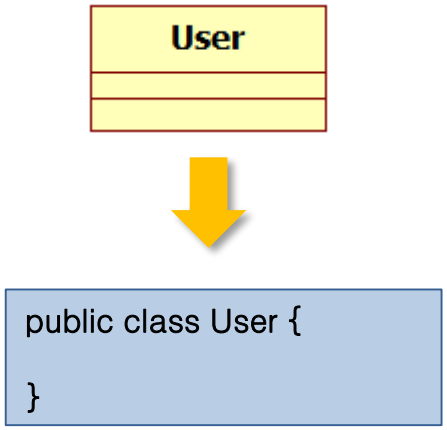
클래스이름
+멤버변수1 +멤버변수2
+메소드1() +메소드2()

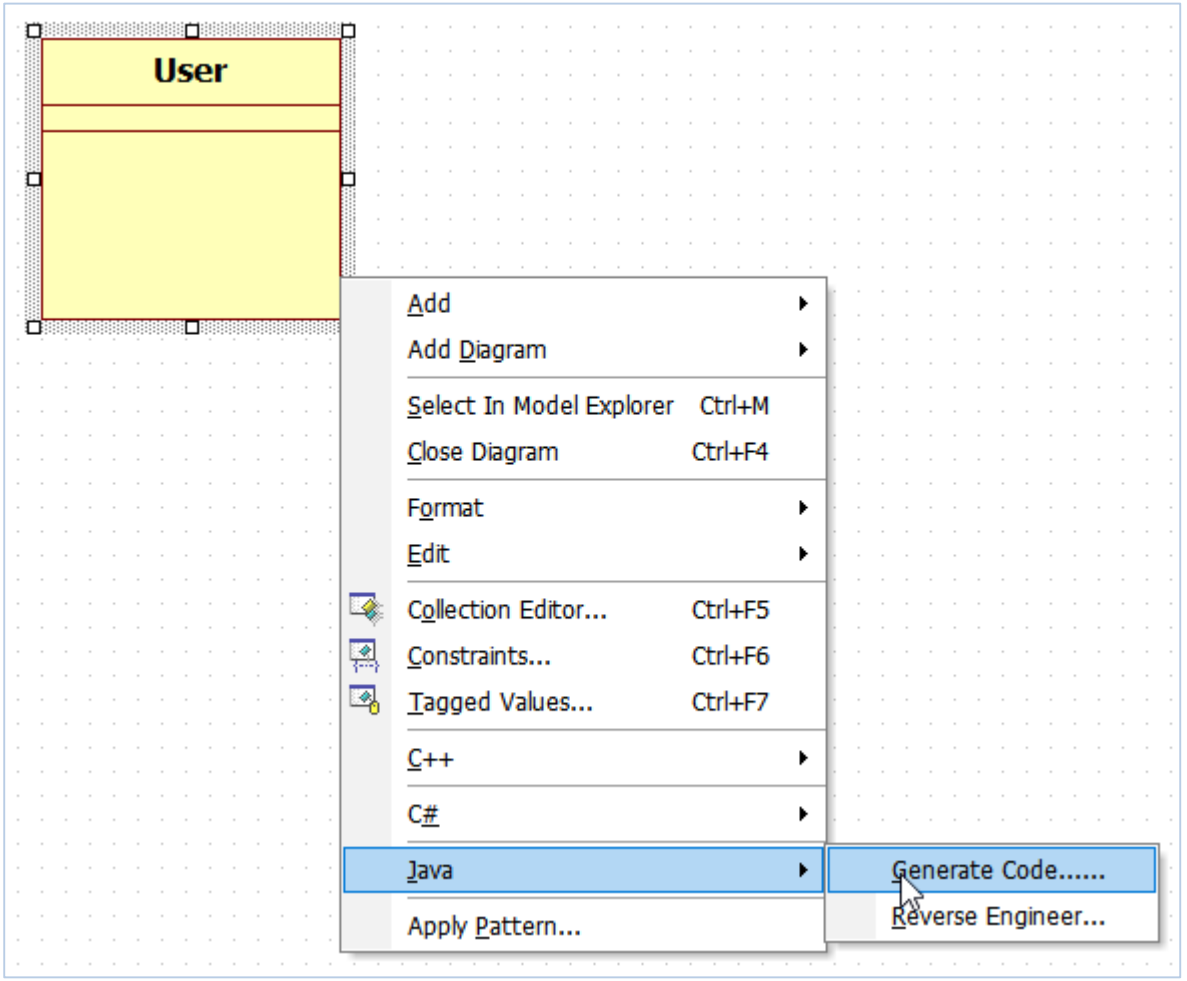
User
-pw: String +id: String
+getPw(): String +setPw(pw: String): void

```
public class User {

    public String id;
    private String pw;

    public String getPw() {
        return pw;
    }
    public void c(String pw) {
        this.pw = pw;
    }

}
```

1. 클래스 다이어그램의 구성요소와 표현

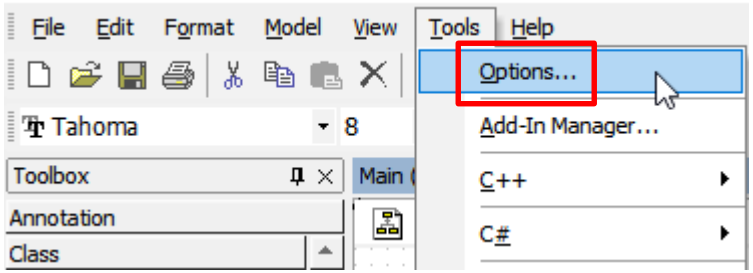
- **클래스 생성 및 자바 코드 생성**

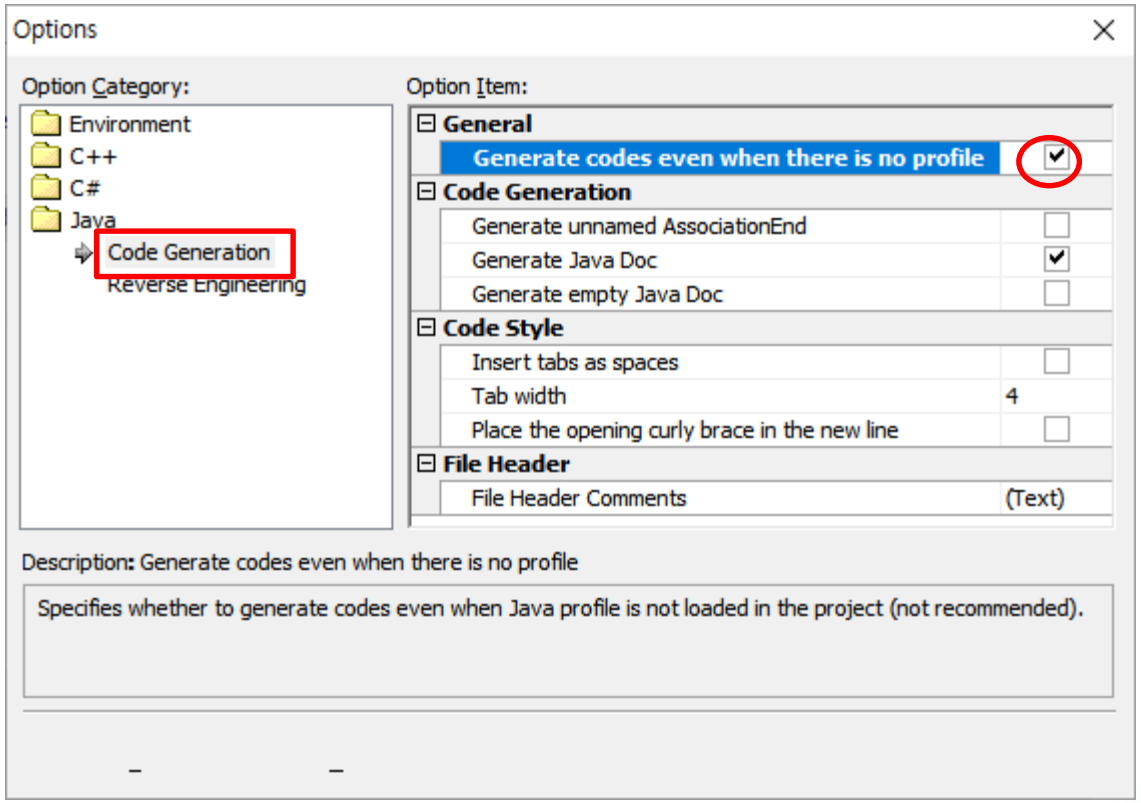




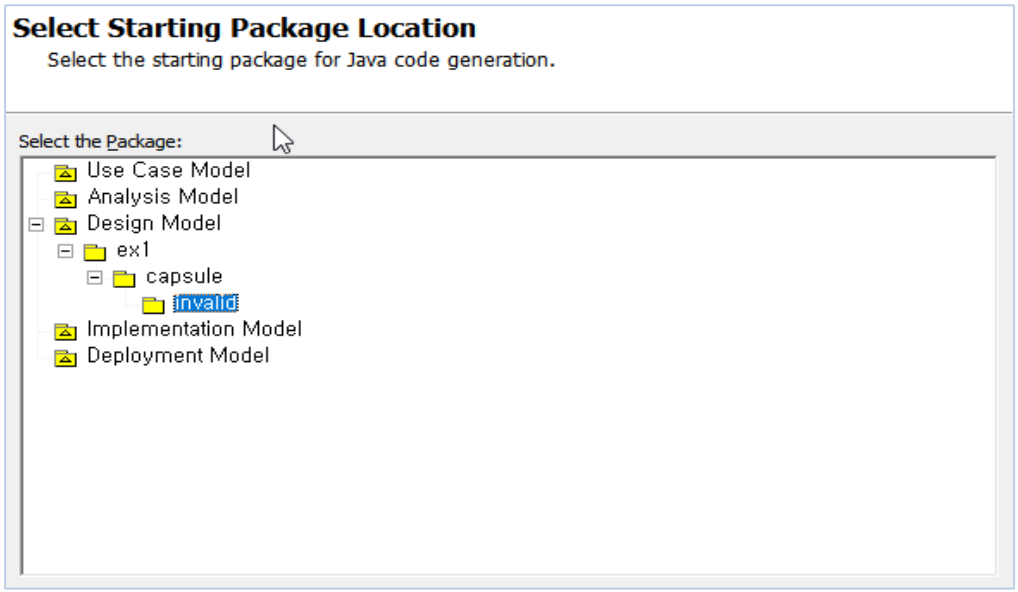
1. 클래스 다이어그램의 구성요소와 표현

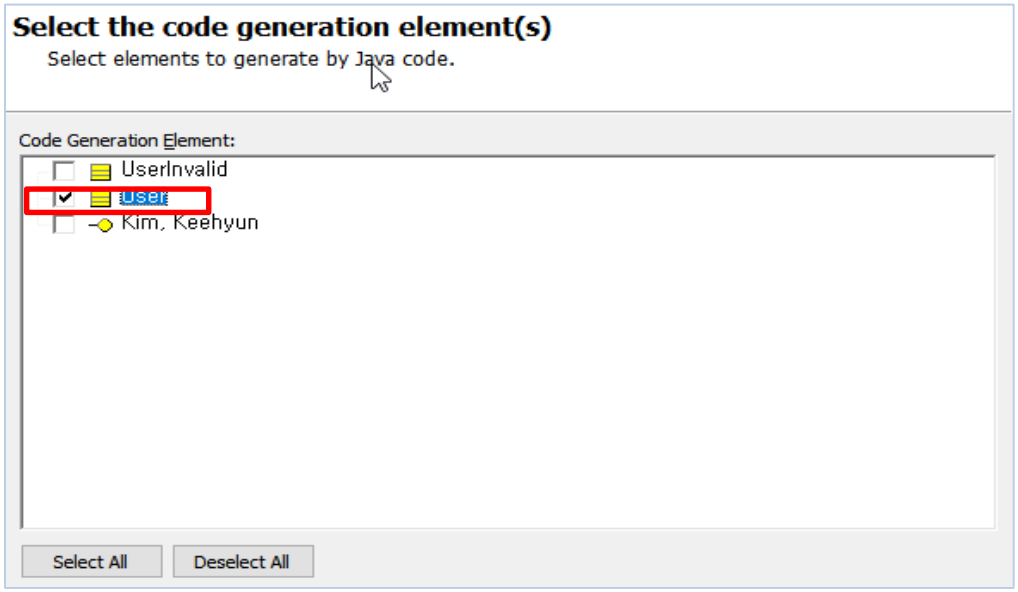
- generate code시 error 발생시



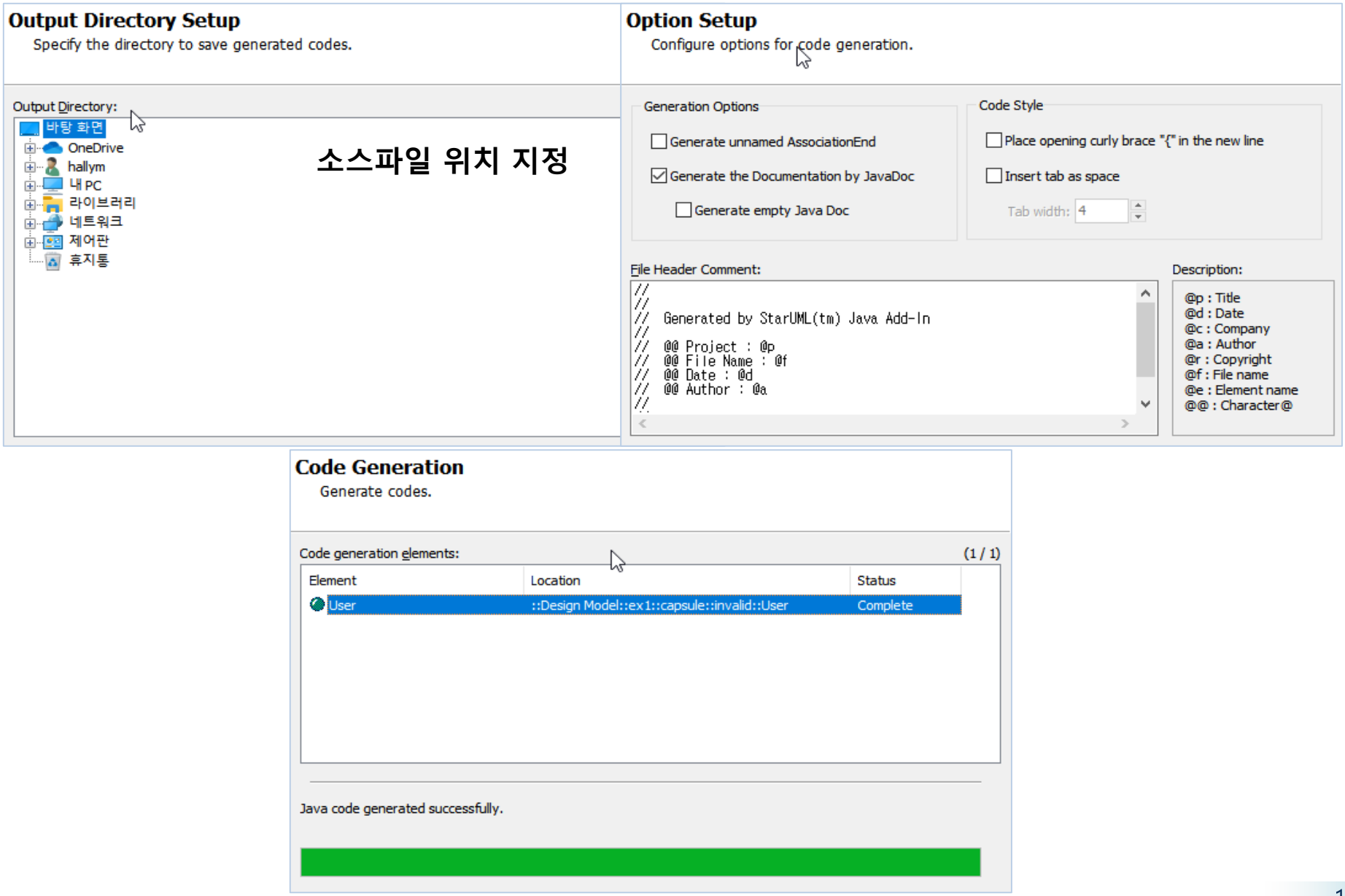


1. 클래스 다이어그램의 구성요소와 표현





1. 클래스 다이어그램의 구성요소와 표현



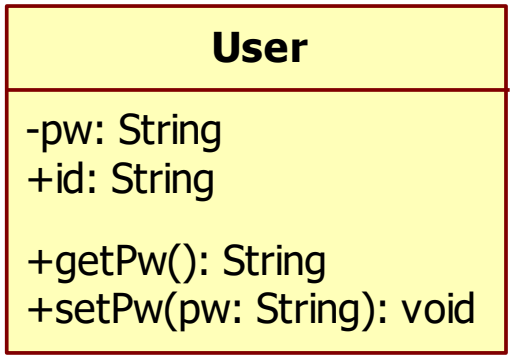
1. 클래스 다이어그램의 구성요소와 표현

```
public class User {

    public String id;
    private String pw;

    public String getPw() {
        return pw;
    }
    public void c(String pw) {
        this.pw = pw;
    }

}
```

내가 만들고 싶은 코드의 형태

```
public class User {

    private String pw;
    public String id;

    public String getPw() {
    }
    public void setPw(String pw) {
    }

}
```

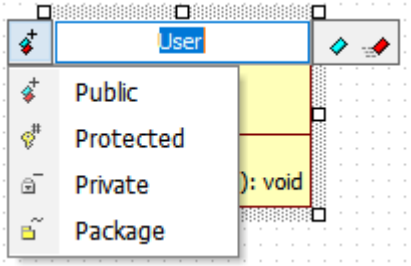
클래스 다이어그램으로 생성된 코드

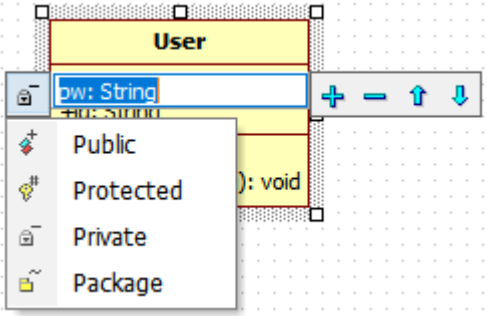


Class diagram

1. 클래스 다이어그램의 구성요소와 표현

- 접근제한자, **속성 , 메소드, 패키지 추가**
  - 접근제한자





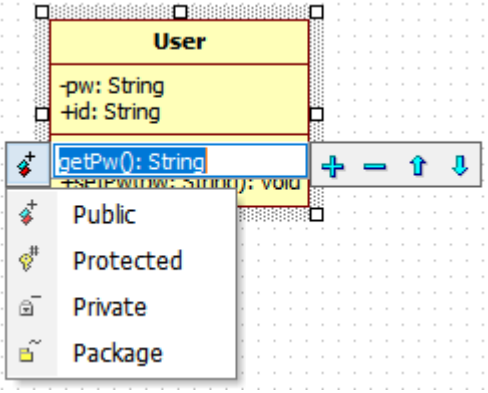
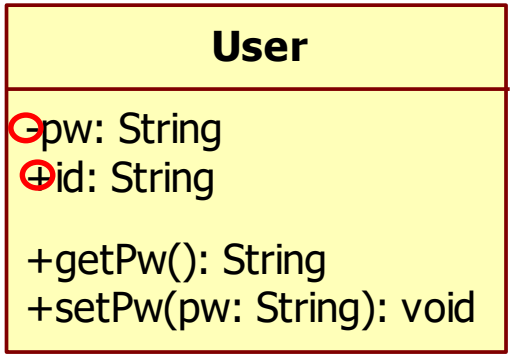


표 4-1 메서드의 종류와 기호

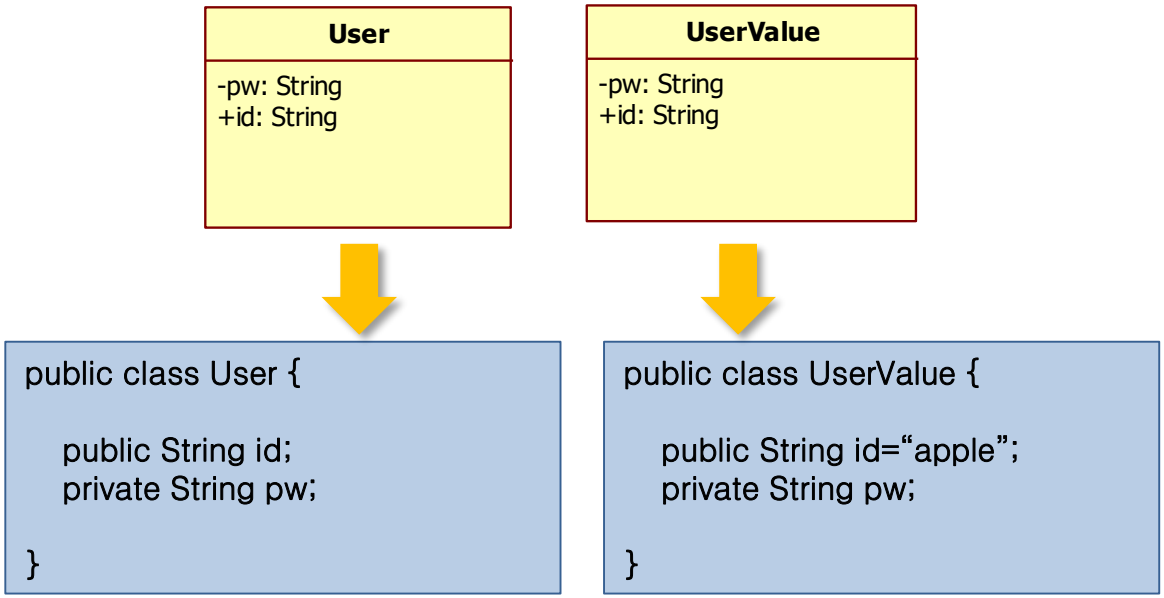
종류	부호	설명
public	+	자신의 속성이나 동작을 외부에 공개하는 접근 제어
private	–	상속된 파생 클래스만 액세스할 수 있는 접근 제어
protected	#	구조체의 멤버 함수만 접근할 수 있으며 외부에서 엑세스할 수 없는 접근 제어

멤버에 접근하는 클래스	멤버의 접근 지정자			
	private	디폴트 접근 지정	protected	public
같은 패키지의 클래스	×	○	○	○
다른 패키지의 클래스	×	×	×	○
접근 가능 영역	클래스 내	동일 패키지 내	동일 패키지와 자식 클래스	모든 클래스



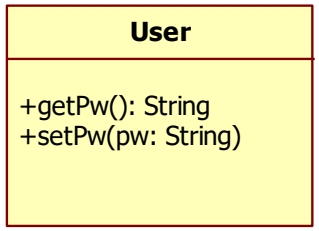
1. 클래스 다이어그램의 구성요소와 표현

- 속성
  - 형식 : **변수명 : 데이터타입 or 변수명 : 데이터타입 =값**



1. 클래스 다이어그램의 구성요소와 표현

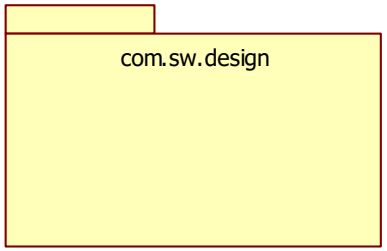
- 메소드
  - 형식 : **메소드명(매개변수명 : 데이터타입):리턴타입**

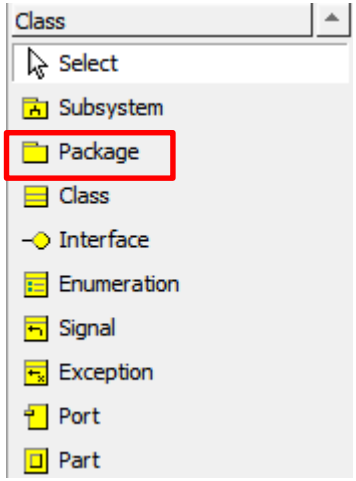


```
public class User {
    public String getPw() {

    }

    public void setPw(String pw) {

    }
}
```

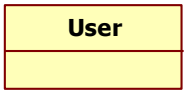
1. 클래스 다이어그램의 구성요소와 표현

- package
  - package 생성

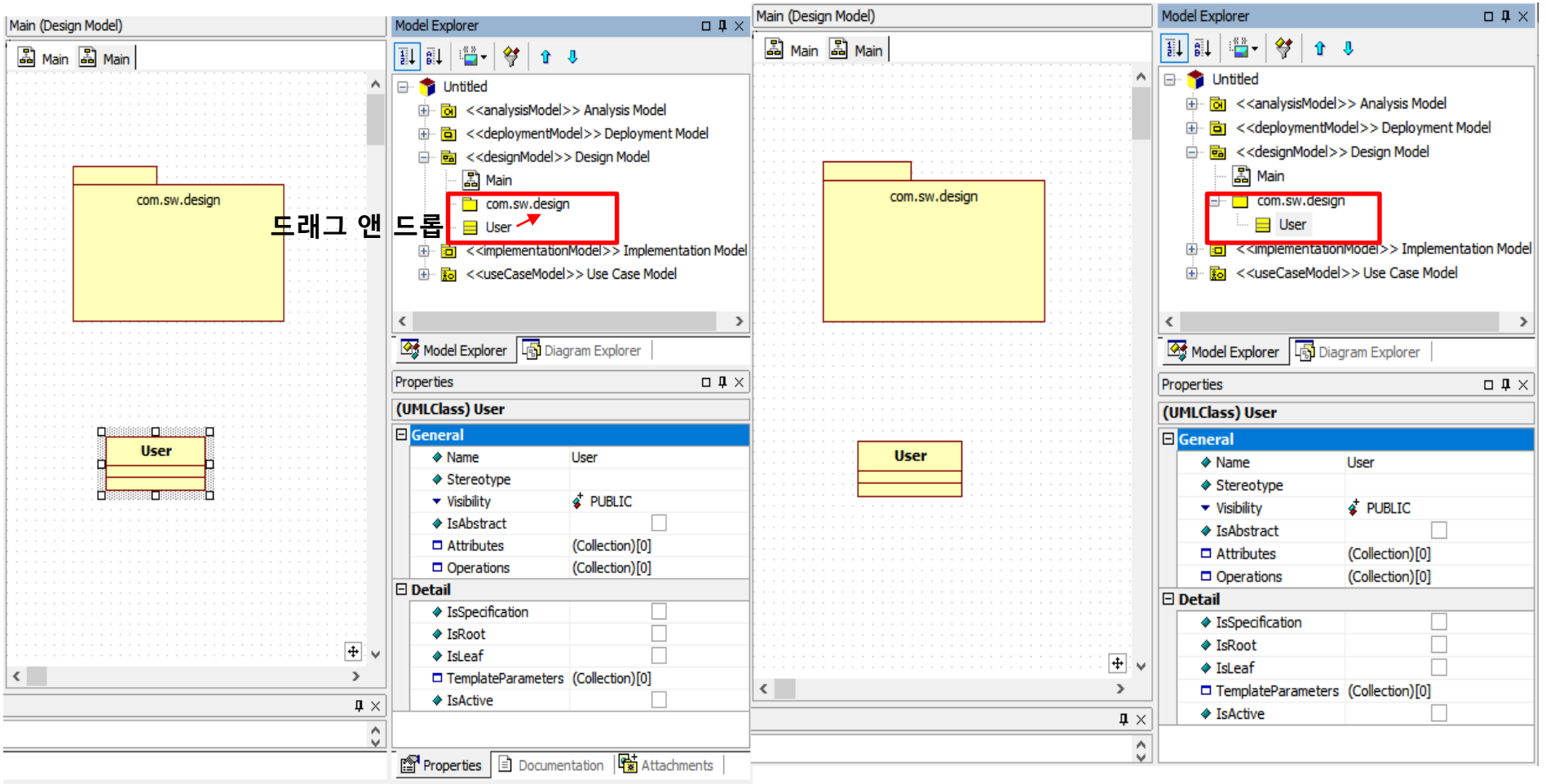




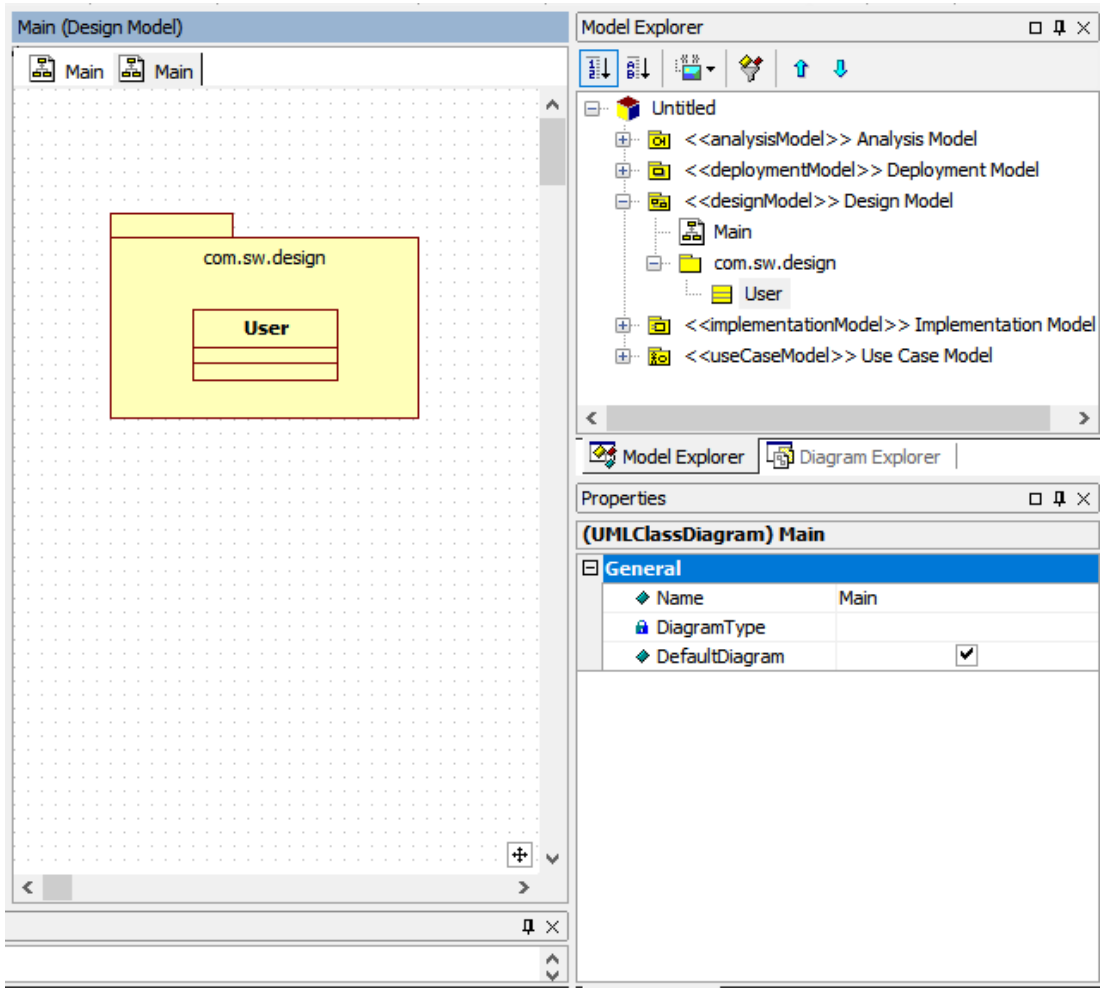
  - class 생성



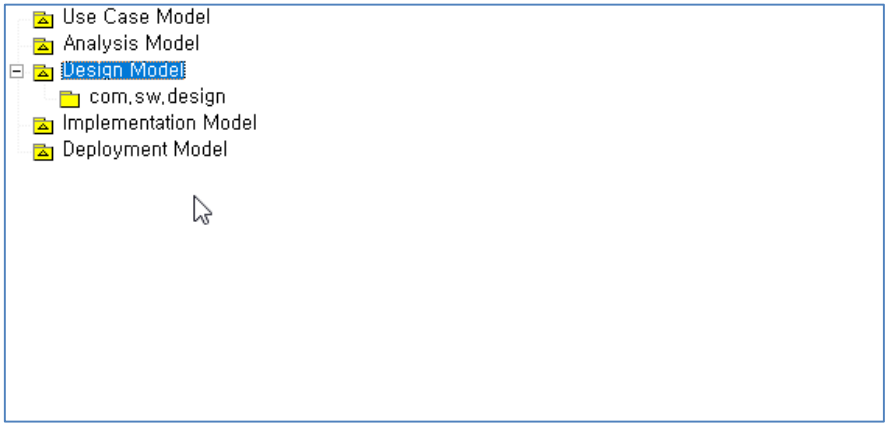
1. 클래스 다이어그램의 구성요소와 표현

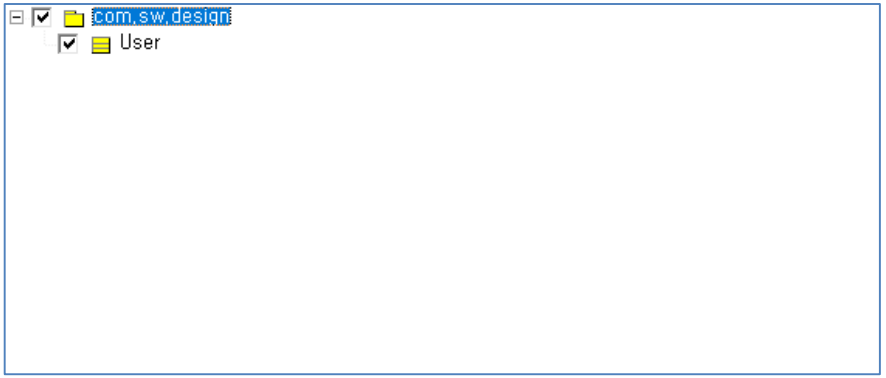


1. 클래스 다이어그램의 구성요소와 표현



- 자바코드 생성






```
package com.sw.design;

public class User {

}
```

실습 (클래스 다이어그램의 구성요소와 표현)

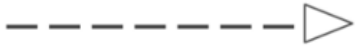
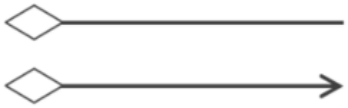
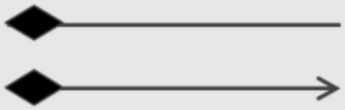
- 클래스 이름 : OrderDto
  - 속성

접근제한자	데이터타입	변수명	초기값
public	String	orderId	100
private	String	userId	212
public	String	productId	454
public	int	price	0
public	Int	orderAmount	2

  - 메소드

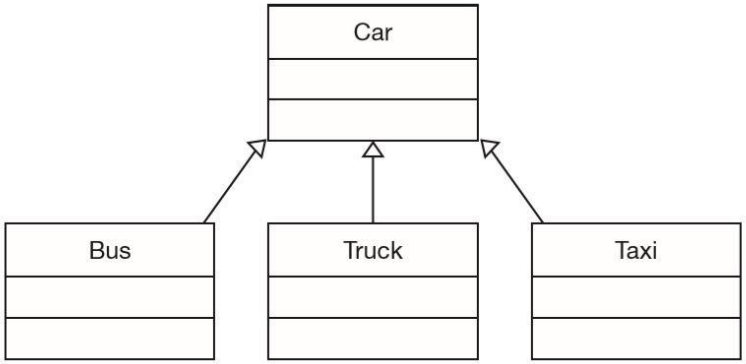
접근제한자	리턴타입	메소드명	매개변수
public	String	getUserId	없음
private	void	setUserId	String id

클래스 사이의 관계

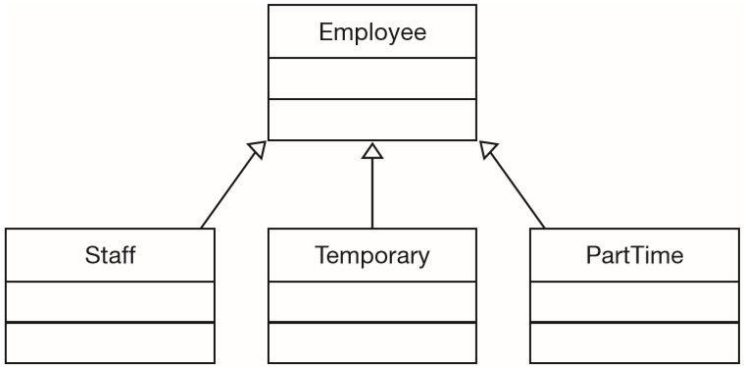
관계	UML 표기
Generalization (일반화)	
Realization (실체화)	
Dependency (의존)	
Association (연관)	
Directed Association (직접연관)	
Aggregation (집합, 집합연관)	
Composition (합성, 복합연관)	

클래스 사이의 관계

- **일반화(Generalization) 관계**
  - 슈퍼클래스와 서브 클래스간의 상속관계를 나타냄

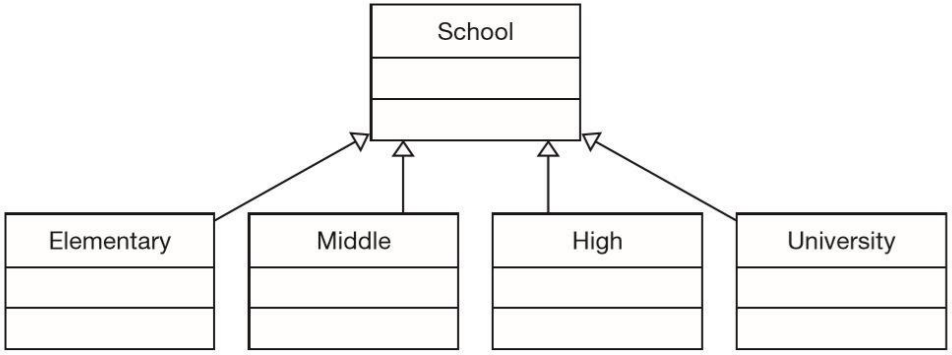


(a) 차와 버스, 트럭, 택시



(c) 사원과 정사원, 계약사원, 아르바이트생

그림 4-15 일반화 관계의 예



(b) 학교와 초등학교, 중학교, 고등학교, 대학교

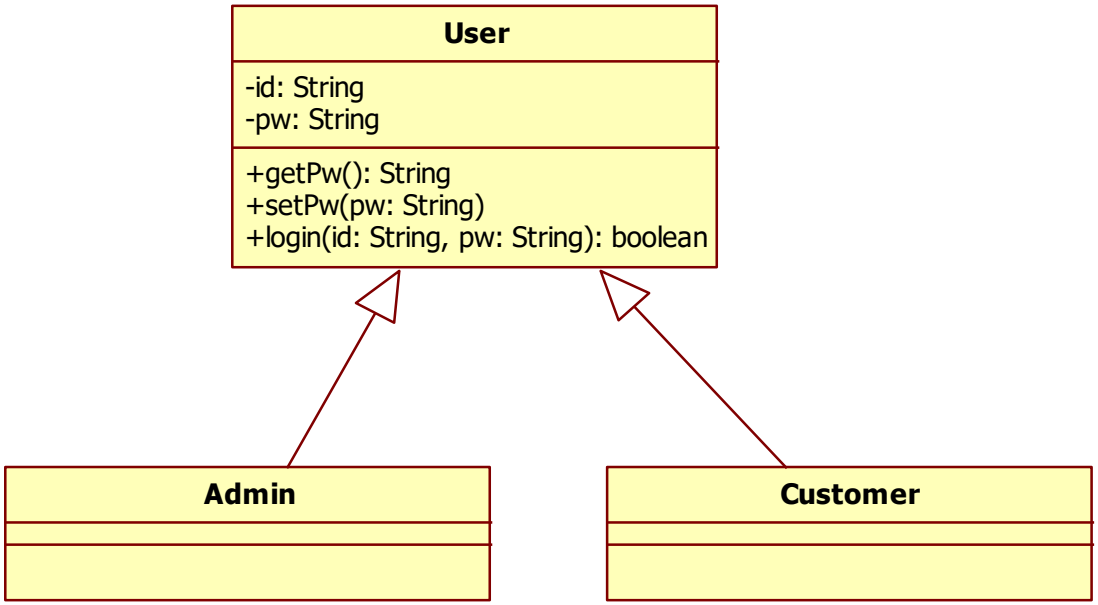
클래스 사이의 관계

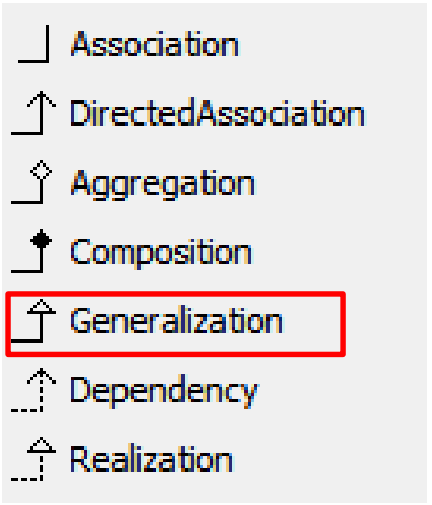
```
public class User {

    private String id;
    private String pw;

    public String getPw() {
        return pw;
    }
    public void setPw(String pw) {
        this.pw = pw;
    }
    public boolean login(String id, String pw) {
        boolean result=false;
        //login 구현
        return result;
    }

}
```

```
public class Admin extends User{

    public boolean login(String id, String pw) {

    }
}
```

```
public class Customer extends User{

    public boolean login(String id, String pw) {

    }
}
```

클래스 사이의 관계


User
-id: String
-pw: String
+getPw(): String
+setPw(pw: String)
+login(id: String, pw: String): boolean
Admin
Customer


Association
DirectedAssociation
Aggregation
Composition
Generalization
Dependency
Realization

클래스 사이의 관계

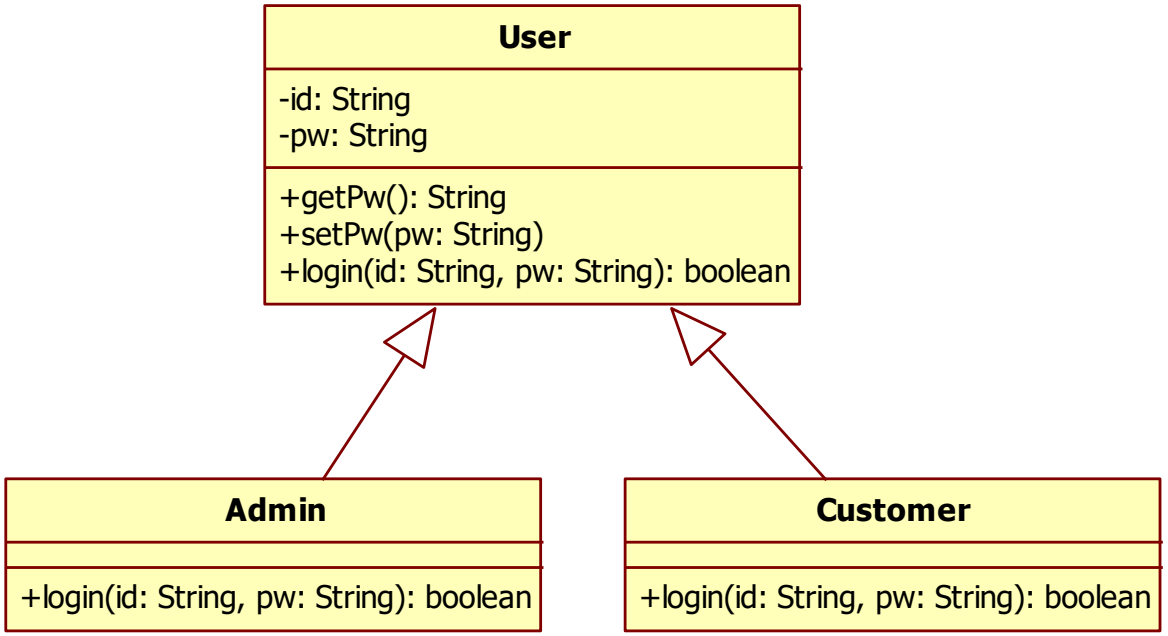
- method overriding

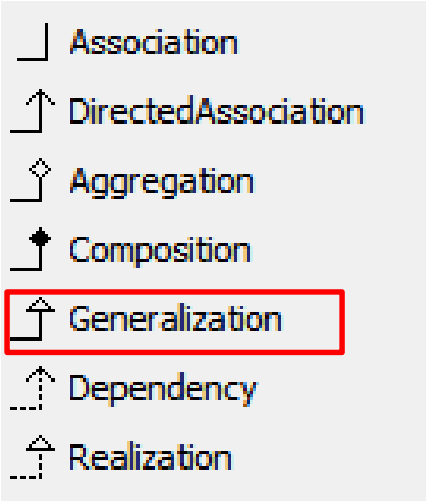
```
public class User {

    private String id;
    private String pw;

    public String getPw() {
        return pw;
    }
    public void setPw(String pw) {
        this.pw = pw;
    }
    public boolean login(String id, String pw) {
        boolean result=false;
        //login 구현
        return result;
    }

}
```

```
public class Admin extends User{

    public boolean login(String id, String pw) {
        boolean result=false;
        //Admin login 구현
        return result;
    }
}
```

```
public class Customer extends User{

    public boolean login(String id, String pw) {
        boolean result=false;
        //Customer login 구현
        return result;
    }

}
```

클래스 사이의 관계


User
-id: String
-pw: String
+getPw(): String
+setPw(pw: String)
+login(id: String, pw: String): boolean
Admin
+login(id: String, pw: String): boolean
Customer
+login(id: String, pw: String): boolean


Association
DirectedAssociation
Aggregation
Composition
Generalization
Dependency
Realization

실습 (Generalization)

- super class : Item
  - 속성

접근제한자	데이터타입	변수명	초기값
public	String	name	없음
public	int	kind	1
public	int	price	0

  - 메소드

접근제한자	리턴타입	메소드명	매개변수
public	int	getKind	없음
private	int	getPrice	없음

- sub class : giftCard
  - 속성

접근제한자	데이터타입	변수명	초기값
private	String	codeNum	없음

  - 메소드

접근제한자	리턴타입	메소드명	매개변수
public	int	getCodeNum	없음

stereo type

- UML에서 제공하는 기본 요소(표현의 한계를 가짐) 이외에 추가적인 확장 요소를 나타내는것
- <<>> 로 표기

표현	뜻
«interface»	인터페이스 클래스
«abstract»	추상화 클래스
«enumeration»	열거형 타입 클래스
«utility»	인스턴스가 없는 static 메서드만 모아둔 클래스
«create»	생성자

인터페이스

class

```
public class Calculator {

    public String calName;

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public int mul(int x, int y) {
        return x*y;
    }

}
```

Abstract class

```
public abstract class Calculator {

    public String calName;

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public abstract int mul(int x, int y);

}
```

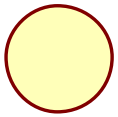
interface

```
public interface Calculator {

    public int add(int x, int y) ;
    public int sub(int x, int y) ;
    public int div(int x, int y);
    public int mul(int x, int y) ;

}
```

인터페이스

- **표현 방법**

1



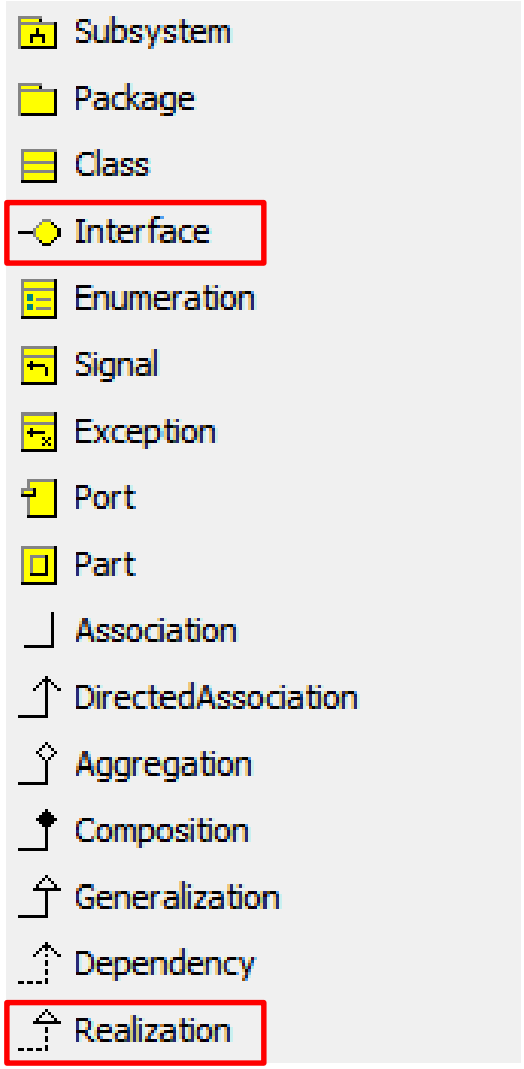
CheckeckLogic

interface icon

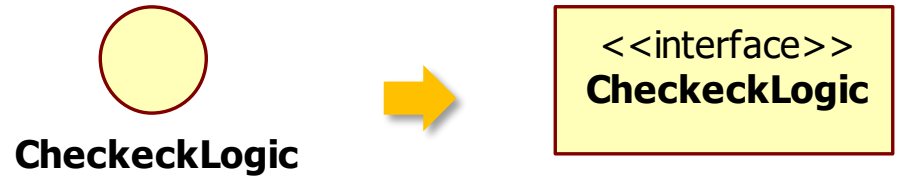
2

<<interface>>
CheckeckLogic

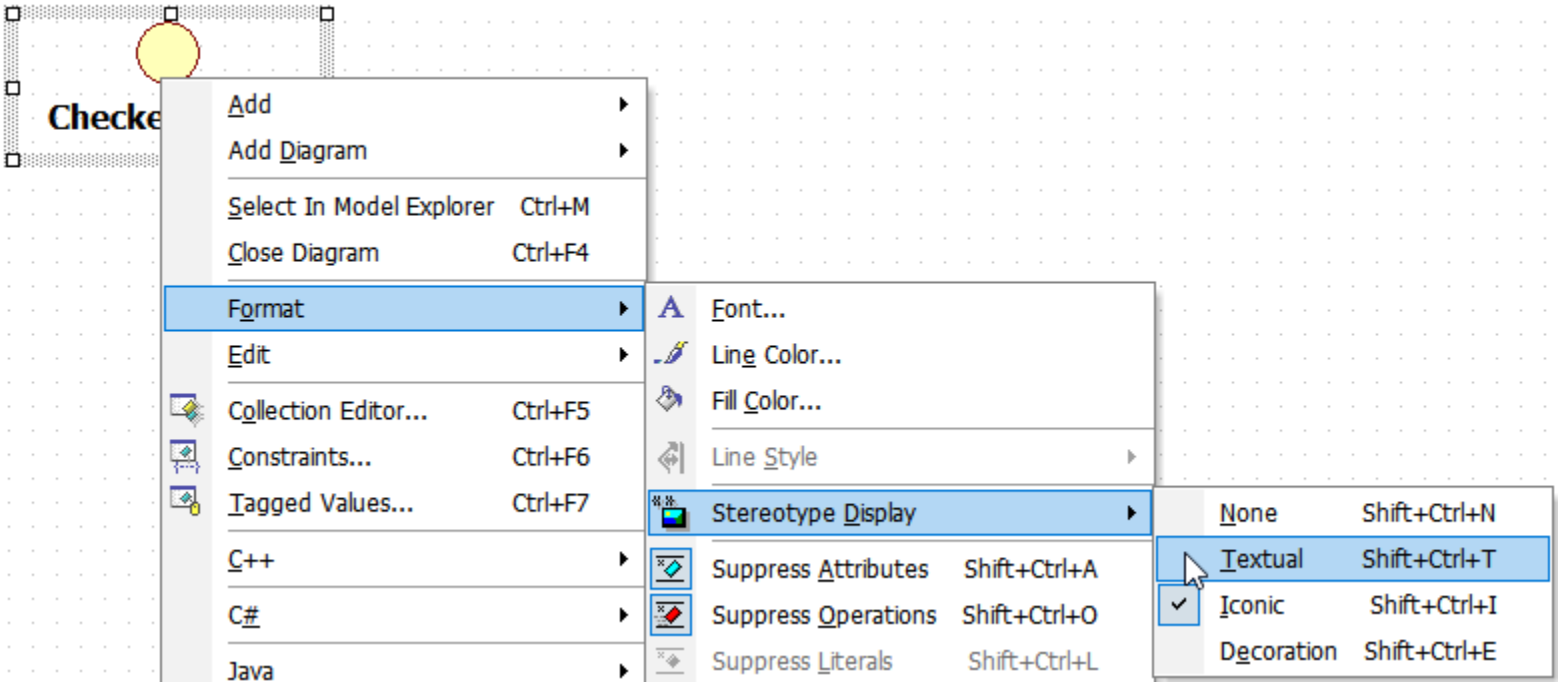
interface strereo



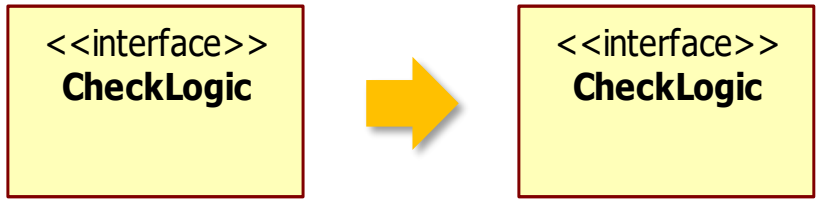
- 아이콘 타입을 스테레오 타입으로 바꾸는 방법

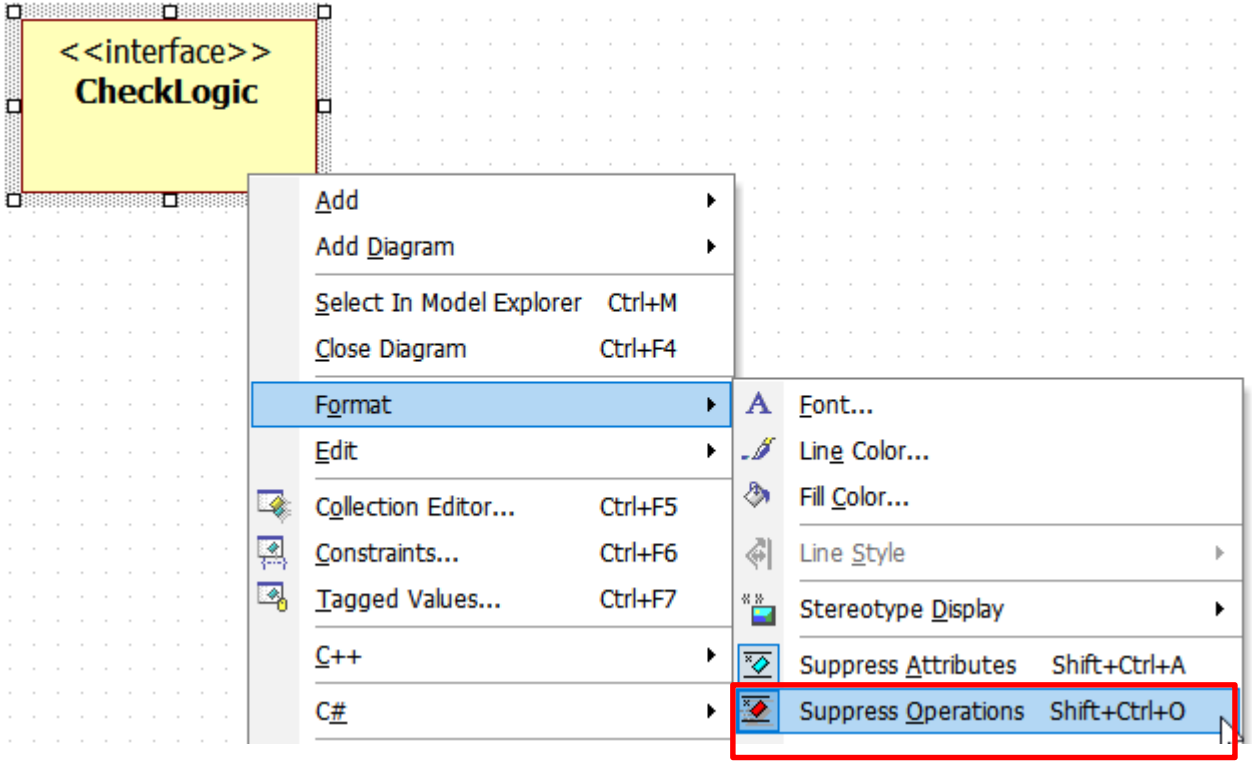


interface icon　　interface strereo

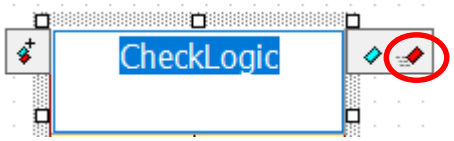


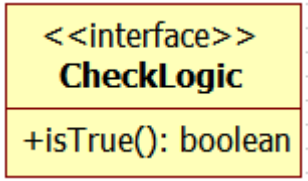
- **메소드 추가 방법**
  - 지금까지의 방법에서 메소드를 추가해도 클래스 다이어그램에서는 보이지 않음



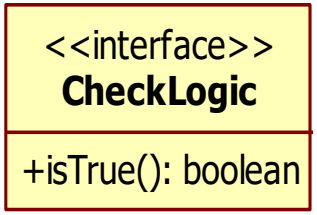


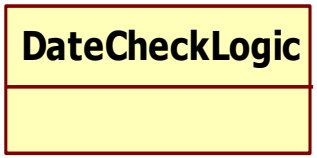
인터페이스





- 구현 클래스 추가

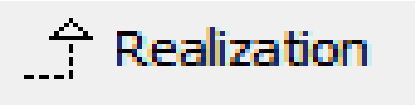


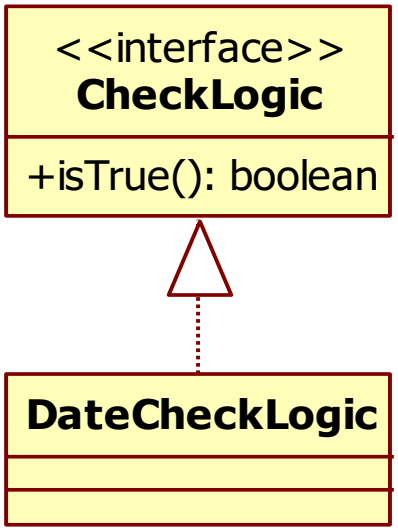


구현클래스명을 인터페이스명+Impl 로 하면 좋음
isTrue() 는 구현클래스에 추가 하지 않음

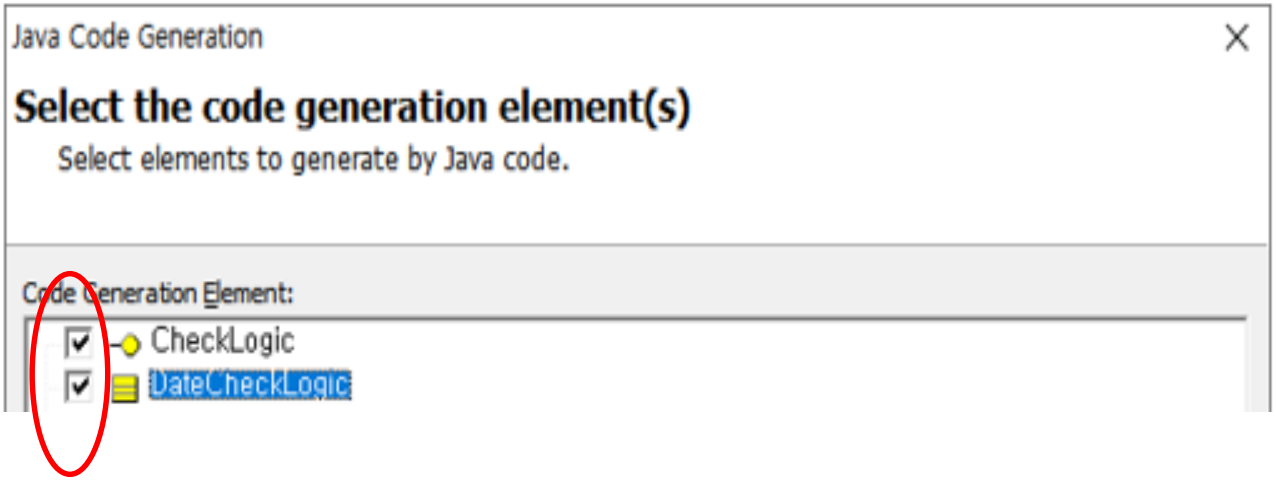
인터페이스

- 관계 추가


Realization


<<interface>>
CheckLogic
+isTrue(): boolean
DateCheckLogic

- 코드 생성



두개 모두 체크

```
public interface CheckLogic {
    public boolean isTrue();
}
```

```
public class DateCheckLogic implements CheckLogic {
    public boolean isTrue();
}
```

주의 !! 제대로 자바코드 생성 안됨

인터페이스

실습

- 인터페이스 이름 : Order
  - 메소드

접근제한자	리턴타입	메소드명	매개변수
public	int	total_price	없음
public	int	getOrderId	없음

- 구현 클래스 이름 : MeatOrderImpl

접근제한자	데이터타입	변수명	초기값
public	String	orderId	100
private	String	userId	212
public	String	productId	454
public	int	price	0
public	Int	orderAmount	2

abstract class

- **Abstract class**

class

```
public class Calculator {

    public String calName;

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public int mul(int x, int y) {
        return x*y;
    }

}
```

Abstract class

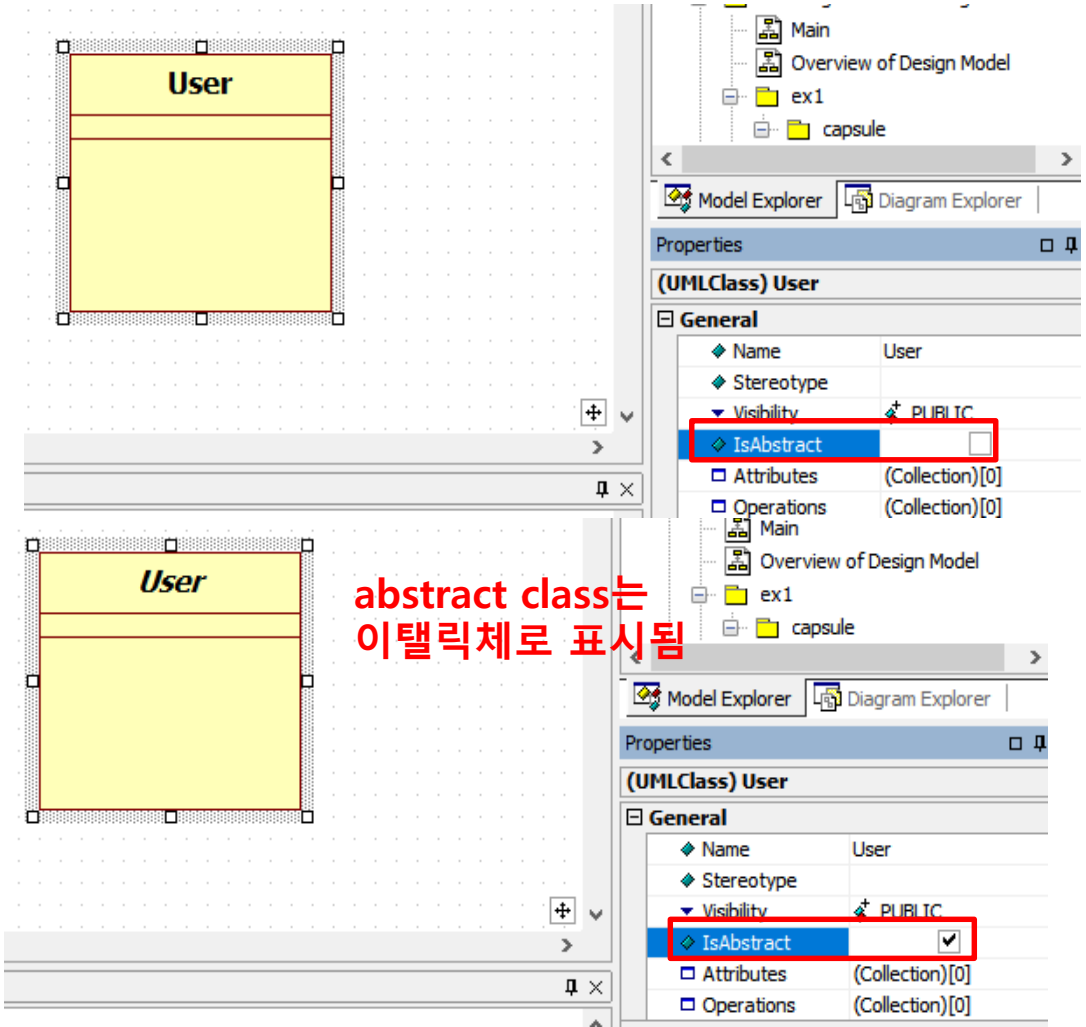
```
public abstract class Calculator {

    public String calName;

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public abstract int mul(int x, int y);

}
```

interface

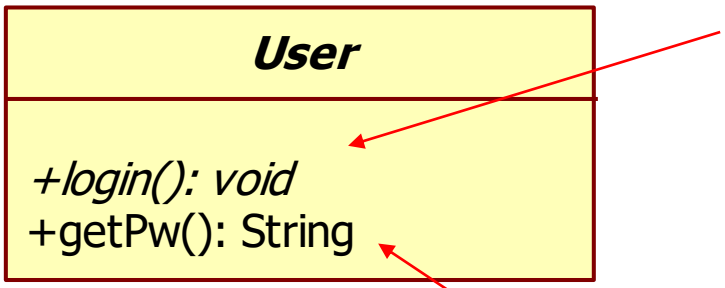
```
public interface Calculator {

    public int add(int x, int y) ;
    public int sub(int x, int y) ;
    public int div(int x, int y);
    public int mul(int x, int y) ;

}
```

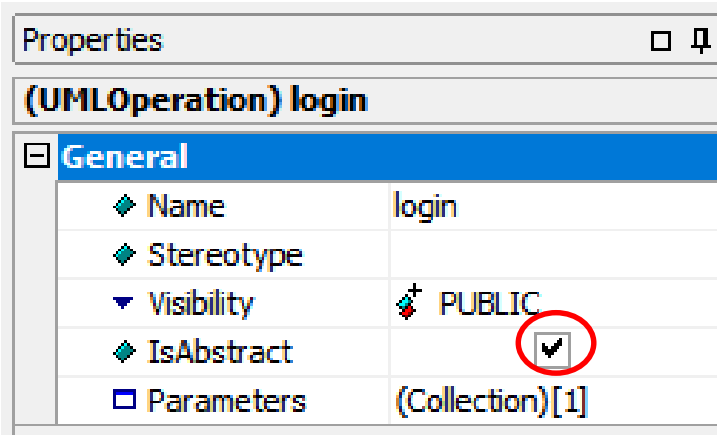
abstract class

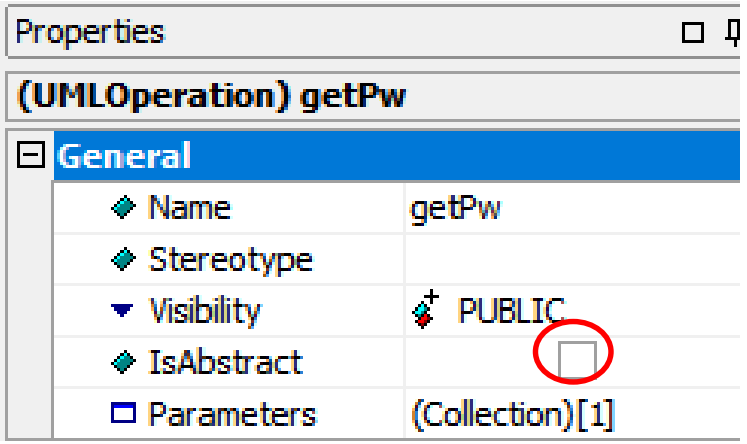
- **abstract class 표현방법**
  - 한눈에 일반 클래스 와 abstract class를 구분하기 어려움



- **abstract method 표현 방법**
  - 메소드 properites에 isAbstract 체크 -> 메소드명이 이탤릭체로 변함

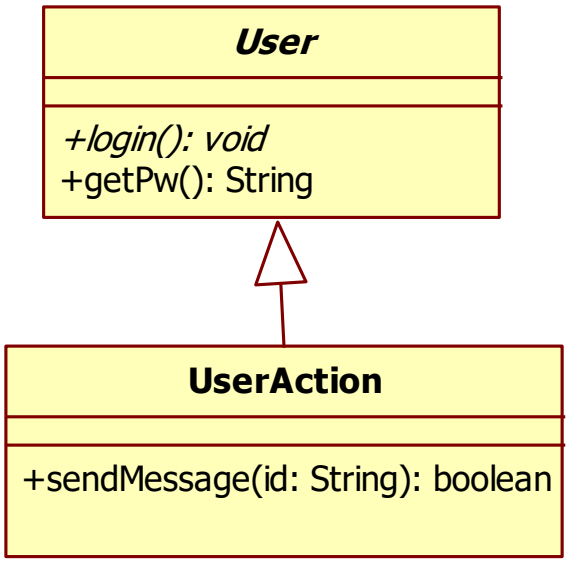


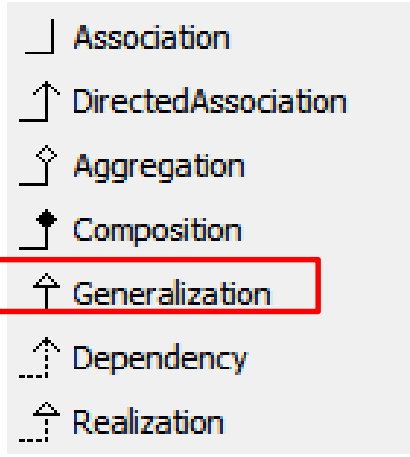




abstract class

- 관계추가






```
public abstract class User {
    public abstract void login();
    public String getPw() {

    }
}
```

```
public class UserAction extends User {
    public boolean sendMessage(String id) {

    }

    public void login() {

    }
}
```

실습 (abstract class)

- abstract class : giftCard
  - 속성

접근제한자	데이터타입	변수명	초기값
public	String	name	없음
public	int	kind	1
public	int	price	0
private	String	codeNum	없음

  - 메소드

접근제한자	리턴타입	메소드명	매개변수
public	int	getKind	없음
private	int	getPrice	없음
public	int	getCodeNum	없음

abstract method

- sub class : HpGiftCard
  - 속성

접근제한자	데이터타입	변수명	초기값
private	int	balance	없음

  - 메소드

접근제한자	리턴타입	메소드명	매개변수
public	int	getBalance	없음

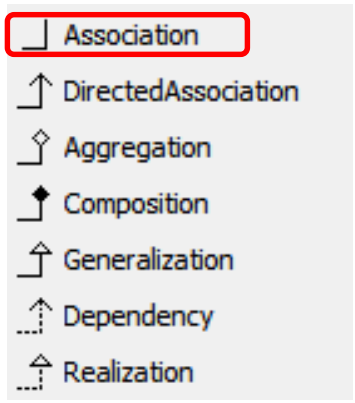
클래스 사이의 관계

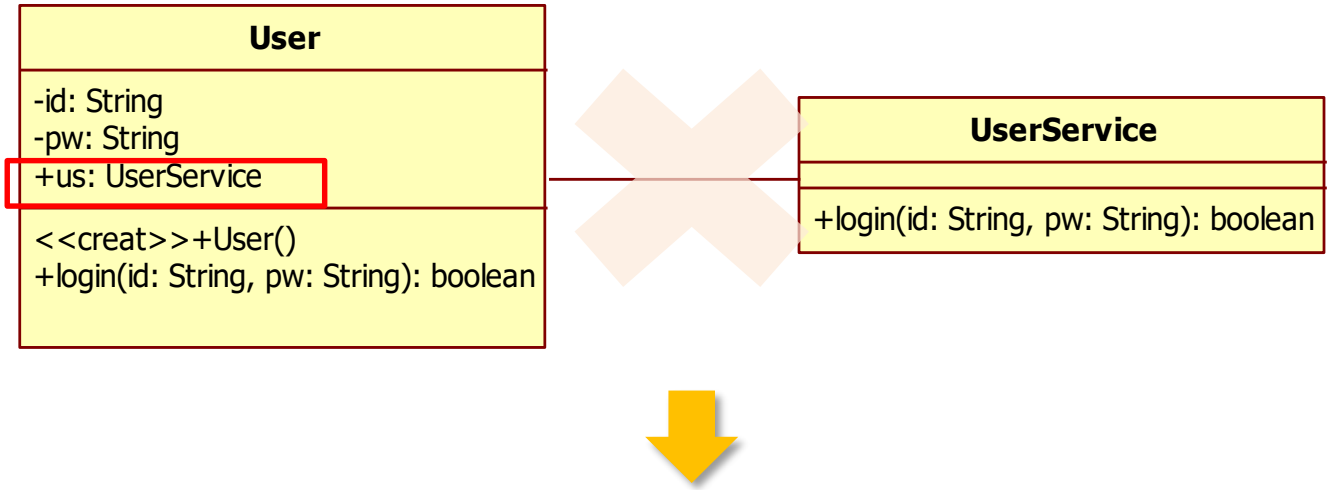
- **Association**
  - 한 객체가 다른 객체와 연결되어 있음을 나타낼때
    - 속성으로 다른 다른 클래스 변수를 가지고 있을때
    - 참조의 관계가 모호함

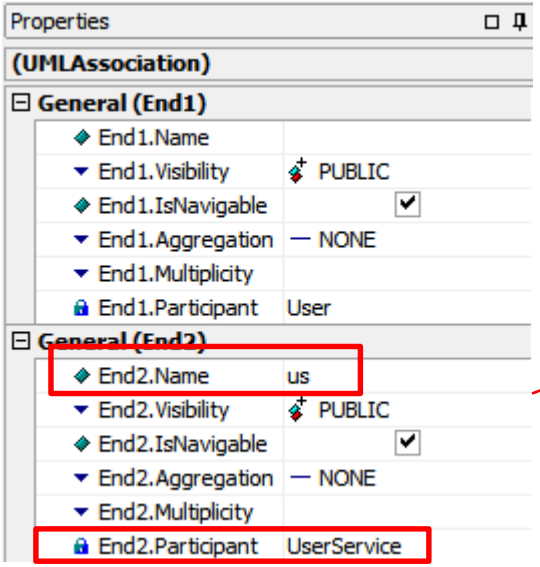
```
public class User {

    private String id;
    private String pw;

    public UserService us;
    public User(){
        us = new UserService();
    }

    public boolean login(String id, String pw) {
        boolean result=false;

        result=us.login(id, pw);
        return result;
    }
}
```

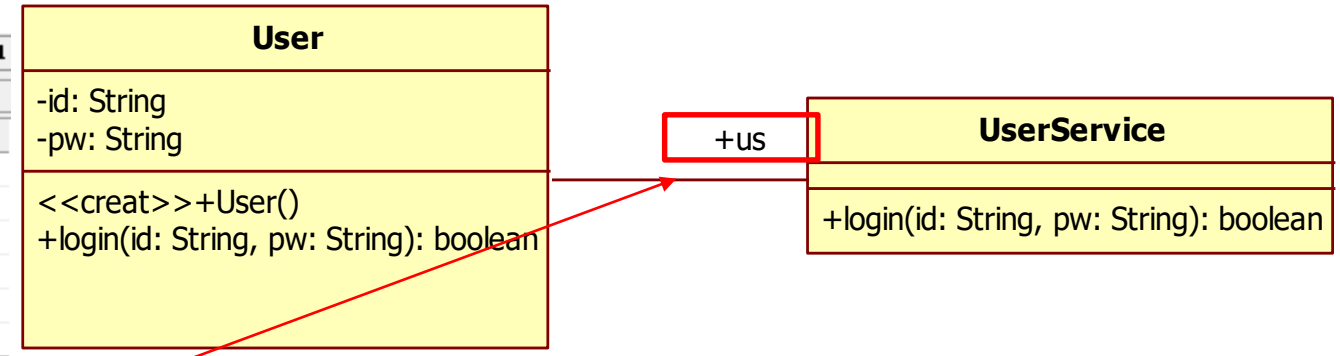
```
public class UserService {
    public boolean login(String id, String pw) {
        boolean result=false;
        //ToDo : implementation
        return result;
    }
}
```

클래스 사이의 관계









association 클릭 후 우측하단 properties 에서 변경

클래스 사이의 관계

- 생성된 코드

```
public class User {
    private String id;
    private String pw;

    public UserService us;
    public void User() {

    }

    public boolean login(String id, String pw) {



    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {

    }
}
```

클래스 다이어그램으로 자동 생성된 코드

```
public class User {

    private String id;
    private String pw;

    public UserService us;
    public User(){
        us = new UserService();
    }

    public boolean login(String id, String pw) {
        boolean result=false;

        result=us.login(id, pw);
        return result;
    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {
        boolean result=false;
        //ToDo : implementation
        return result;
    }
}
```

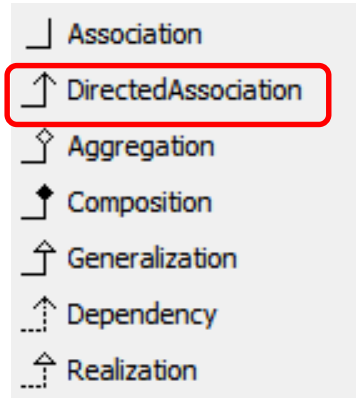
내가 만들고자 하는 코드

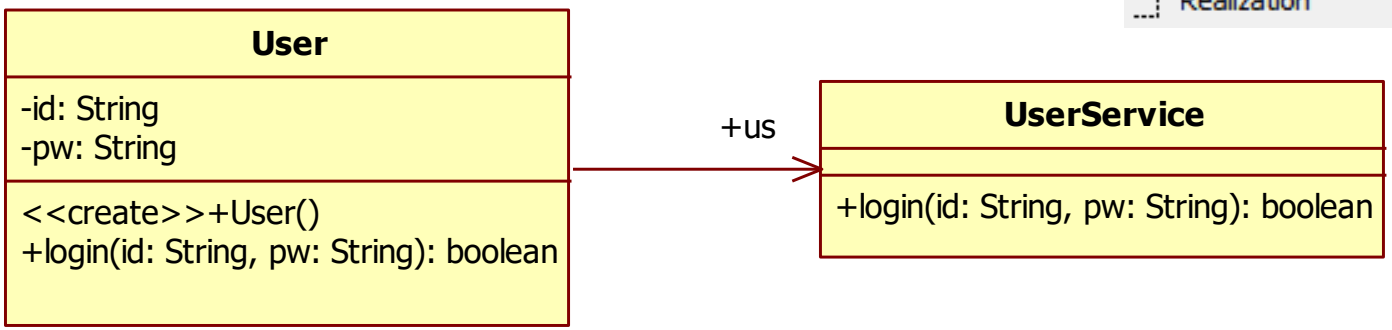
클래스 사이의 관계

- **DirectedAssociation**
  - Association의 의미에 참조하는쪽과 당하는 쪽의 방향성이 추가됨

```
public class User {

    private String id;
    private String pw;

    public UserService us;
    public User(){
        us = new UserService();
    }

    public boolean login(String id, String pw) {
        boolean result=false;

        result=us.login(id, pw);
        return result;
    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {
        boolean result=false;
        //ToDo : implementation
        return result;
    }
}
```





클래스 사이의 관계

- 생성된 코드

```
public class User {
    private String id;
    private String pw;

    public UserService us;
    public void User() {

    }

    public boolean login(String id, String pw) {



    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {

    }
}
```

클래스 다이어그램으로 자동 생성된 코드

```
public class User {

    private String id;
    private String pw;

    public UserService us;
    public User(){
        us = new UserService();
    }

    public boolean login(String id, String pw) {
        boolean result=false;

        result=us.login(id, pw);
        return result;
    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {
        boolean result=false;
        //ToDo : implementation
        return result;
    }
}
```

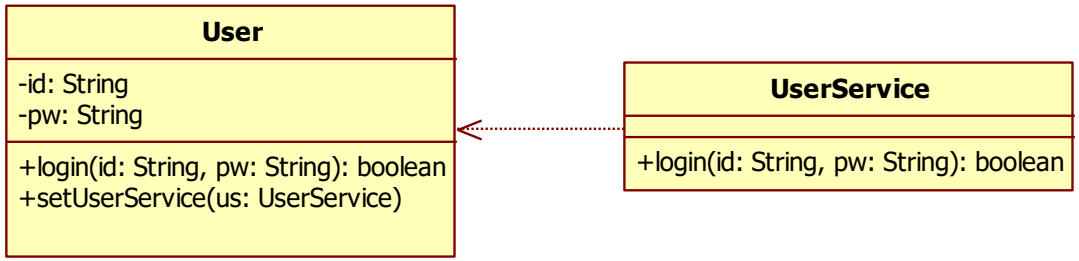
내가 만들고자 하는 코드

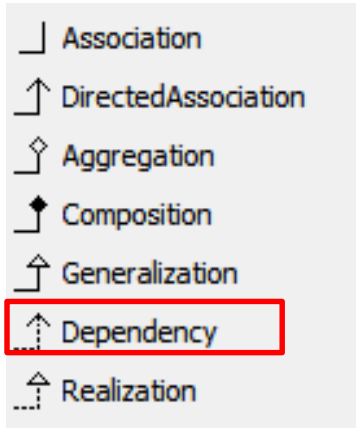
클래스 사이의 관계

- **dependency**
  - 일반적으로 한 클래스가 다른 클래스를 사용하는 경우
    - 연산의 인자(참조값)로 사용될 때
    - 메서드 내부의 지역객체로 참조될 때

```
public class User {

    private String id;
    private String pw;

    public boolean login(String id, String pw) {
        boolean result=false;
        UserService us = new UserService();
        result=us.login(id, pw);
        return result;
    }

    public void setUserService(UserService us) {

    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {
        boolean result=false;
        //ToDo : implementation
        return result;
    }
}
```





클래스 사이의 관계

- 생성된 코드

```
public class User {

    private String id;
    private String pw;

    public boolean login(String id, String pw) {

    }



    public void setUserService(UserService us) {

    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {

    }
}
```

클래스 다이어그램으로 자동 생성된 코드

```
public class User {

    private String id;
    private String pw;

    public boolean login(String id, String pw) {
        boolean result=false;
        UserService us = new UserService();
        result=us.login(id, pw);
        return result;
    }

    public void setUserService(UserService us) {

    }
}
```

```
public class UserService {
    public boolean login(String id, String pw) {
        boolean result=false;
        //ToDo : implementation
        return result;
    }
}
```

내가 만들고자 하는 코드

실습 (클래스 사이의 관계)

- 다음을 클래스 다이어그램으로 표현하세요

```
public class User {

    public Schedule getSchedule() {
        return new Schedule();
    }
    public void useSchedule(Schedule shedule) {

    }
}
```

```
import java.util.Date;

public class Schedule {
    Date date;
    String title;
}
```

- 다음을 클래스 다이어그램으로 표현하세요

```
public class Grade {

    public Subject subject;
    private String grade;

}
```

```
public class Subject {

    public String name;

}
```