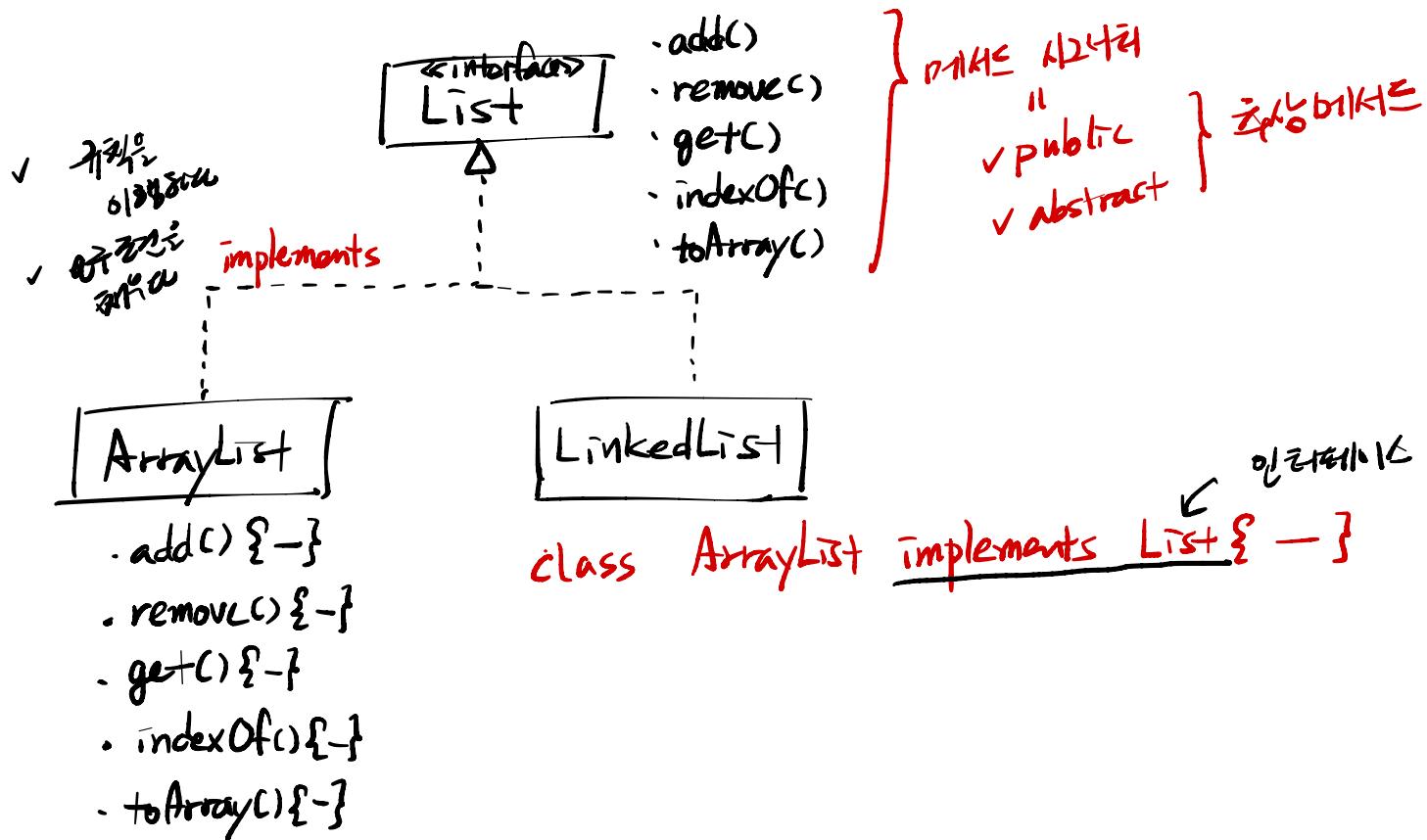
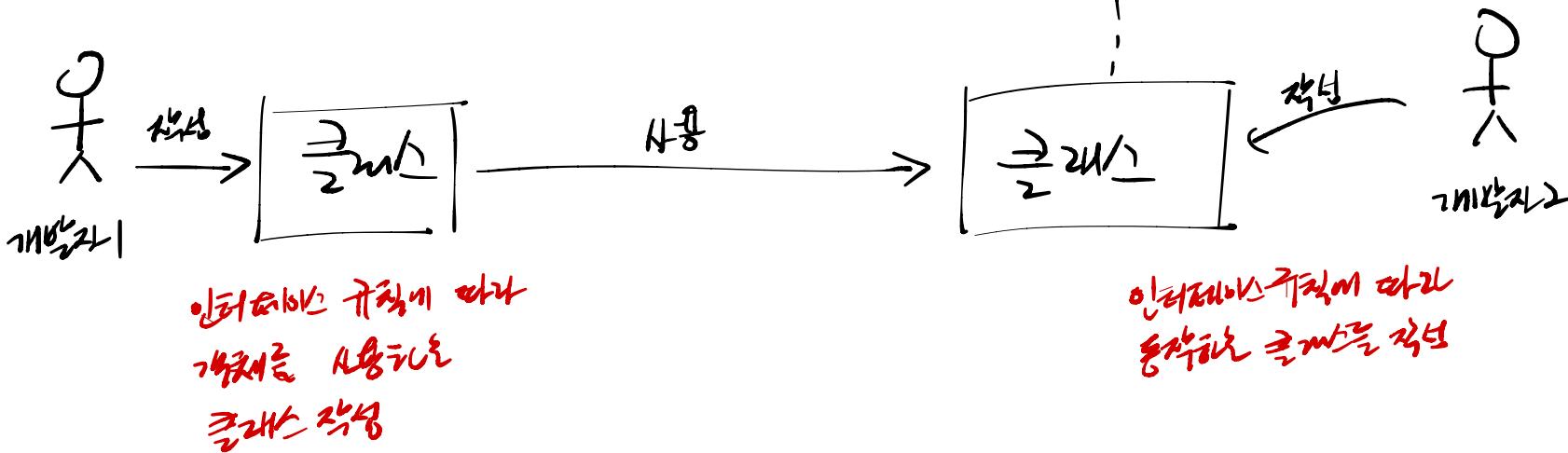
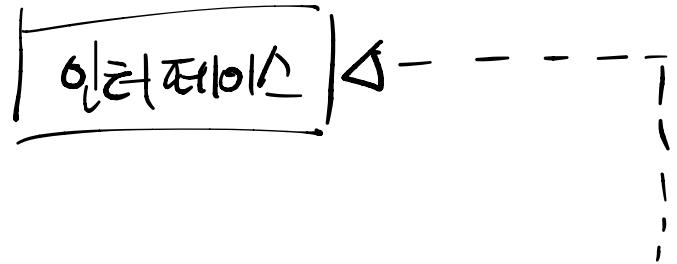


17. 인터페이스를 이용한 구조화된 접근



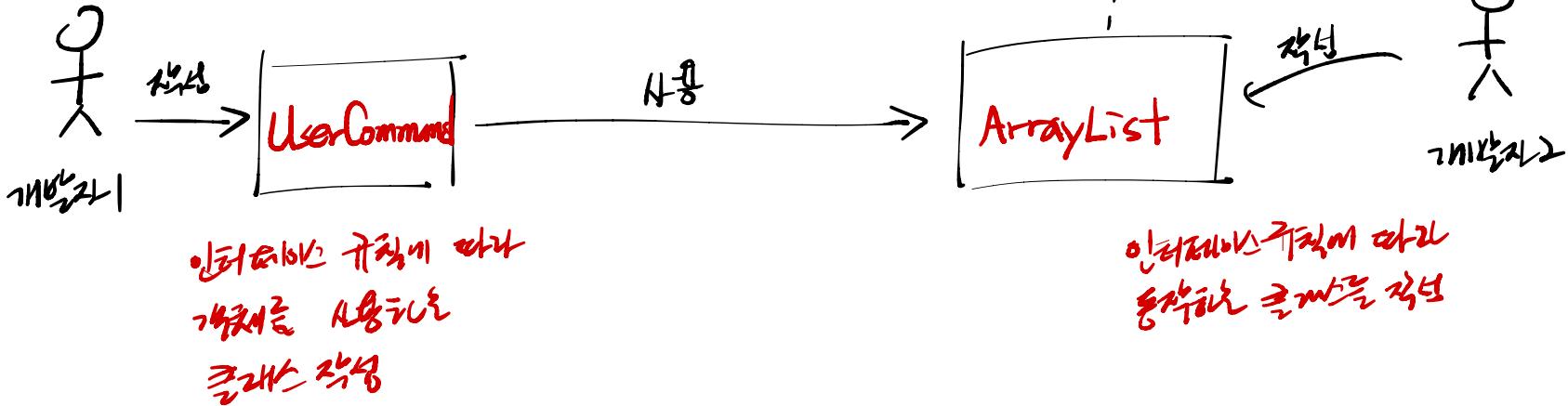
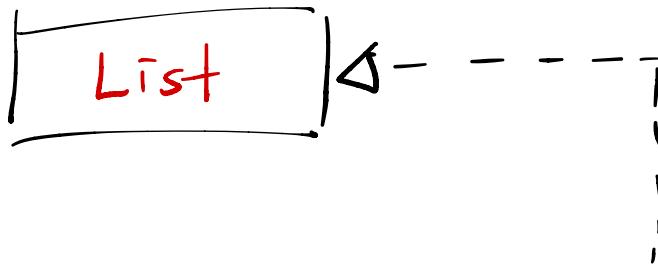
* 인터페이스

가장 중요한 것은 강의에 듣고
① 교과교재는 일관성있게 만족할 수 있다.
② 교재가 쉽다.

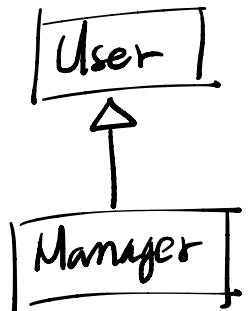


* 인터페이스

인터페이스는 구현체 없이
제공하는 기능의 만족도가 높다.



* instanceof vs getClass()



equals() {
 }
 ==
 }

`equals(Object obj)` {

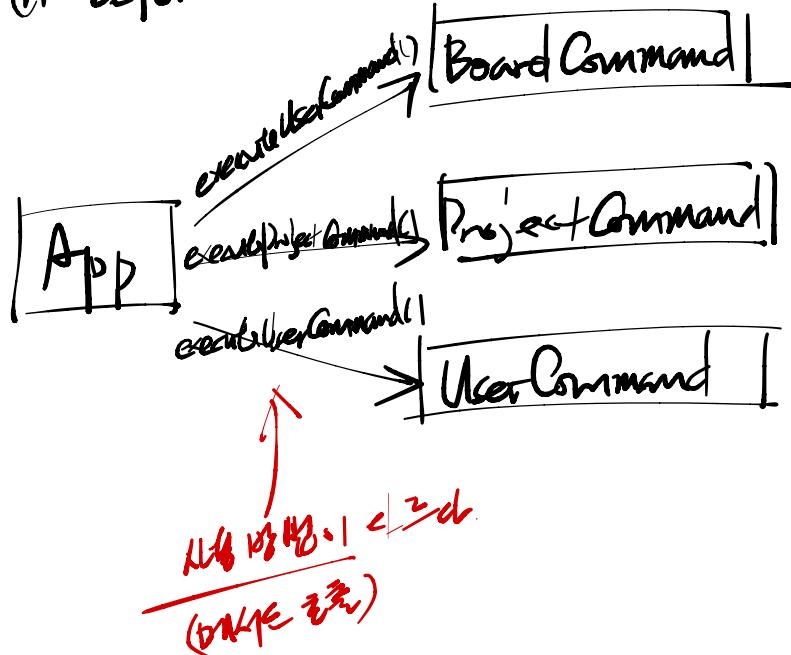
```
if (this.getClass() != obj.getClass()){\n    return false;\n}
```

if(!(obj instanceof User)) {
 return false;
}

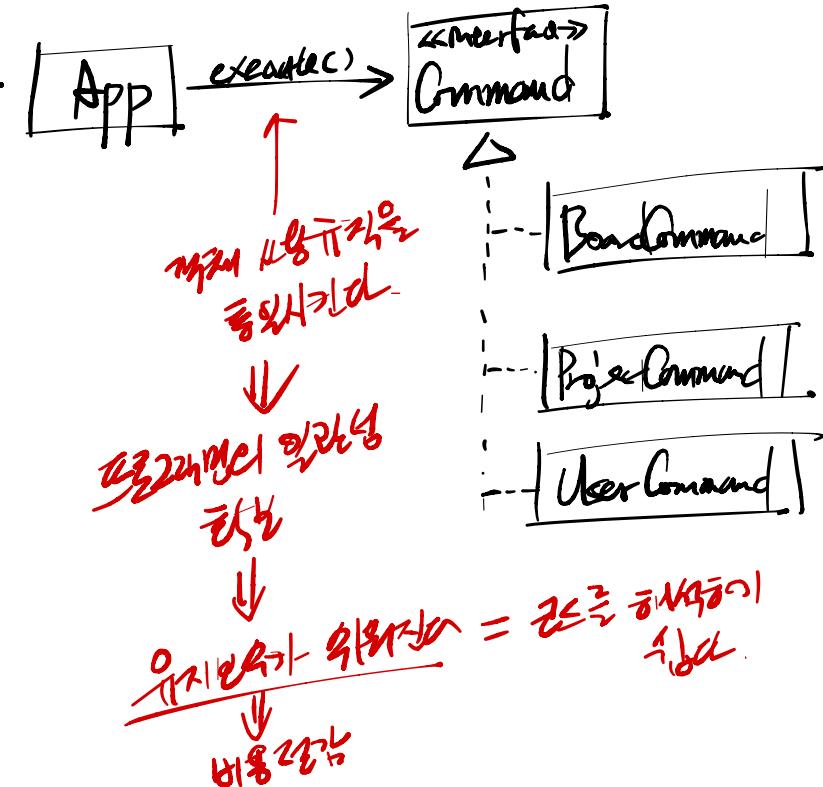
| s | Worl 1 | World 2 |
|----------------|--------|---------|
| ul.equals(str) | F | F |
| ul.equals(ul) | T | T |
| ul.equals(m) | F | T |

* 121 능력과 122 번의 학습자료

① before

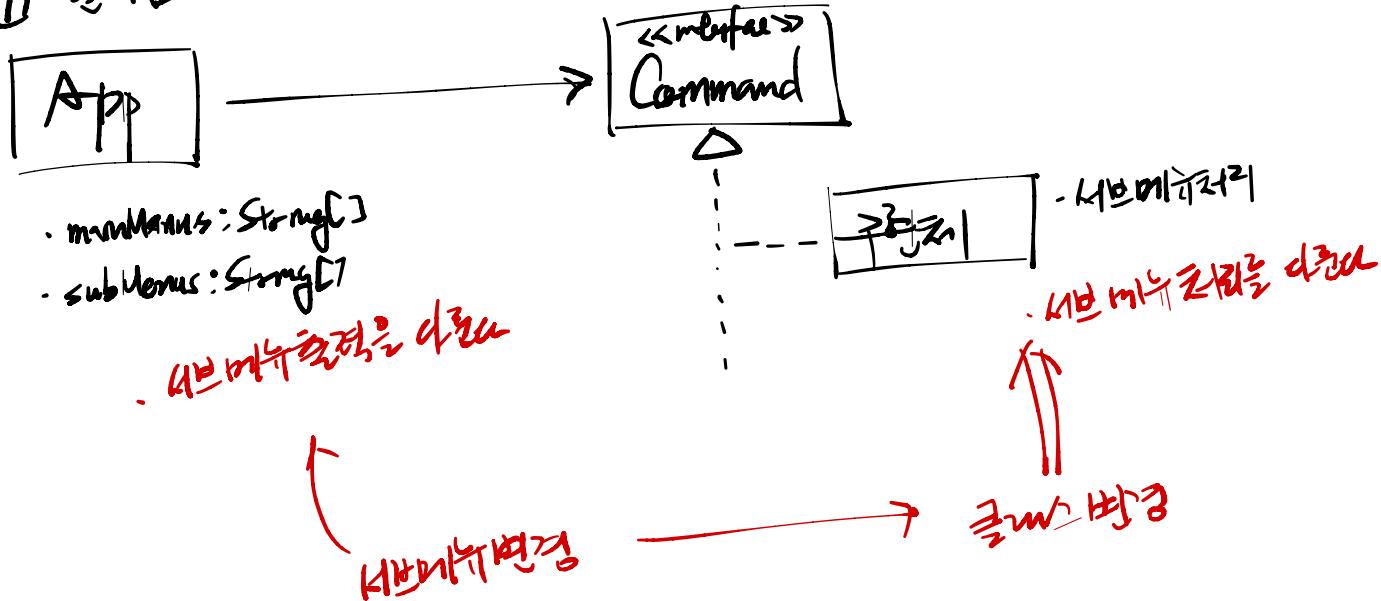


② after



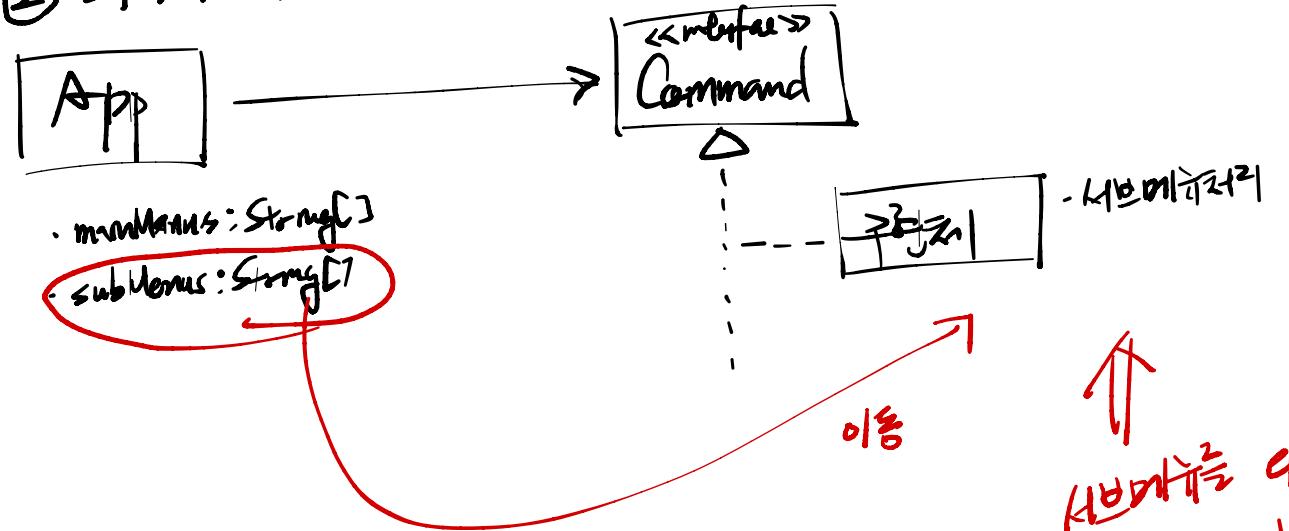
18. 리퍼팅

卷之三



18. 디자인 패턴

② 명령 : GRASP의 High Cohesion & Low Coupling



이동

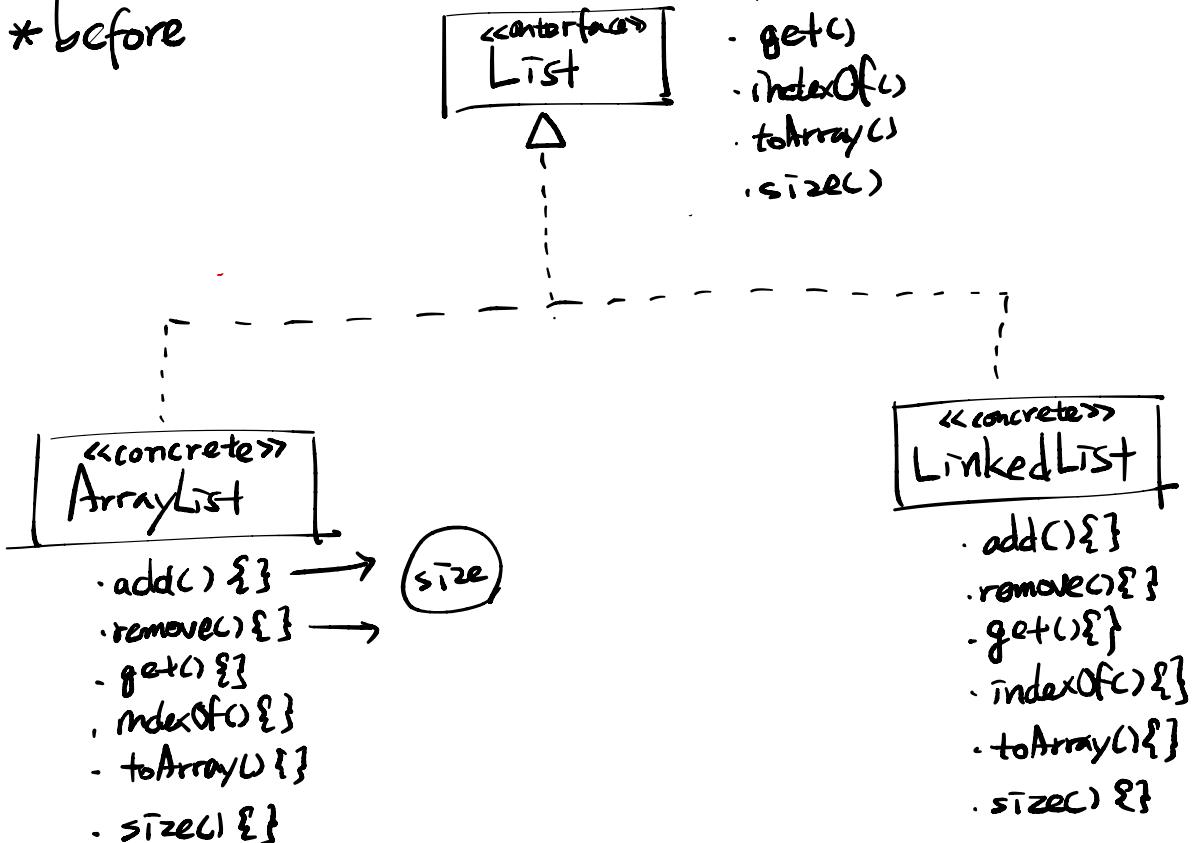
- 메뉴판(메뉴판)

↑
메뉴판을 다루는 역할
Command 구현체로
변경됨.

||
유저에게 응답한다

19. 상속의 Generalization - ①

* before



19. 상속의 Generalization - ①

* after

①

상속을 통한 재사용
공통 기능은 대상으로
설정해주는 편이 좋음
하지만

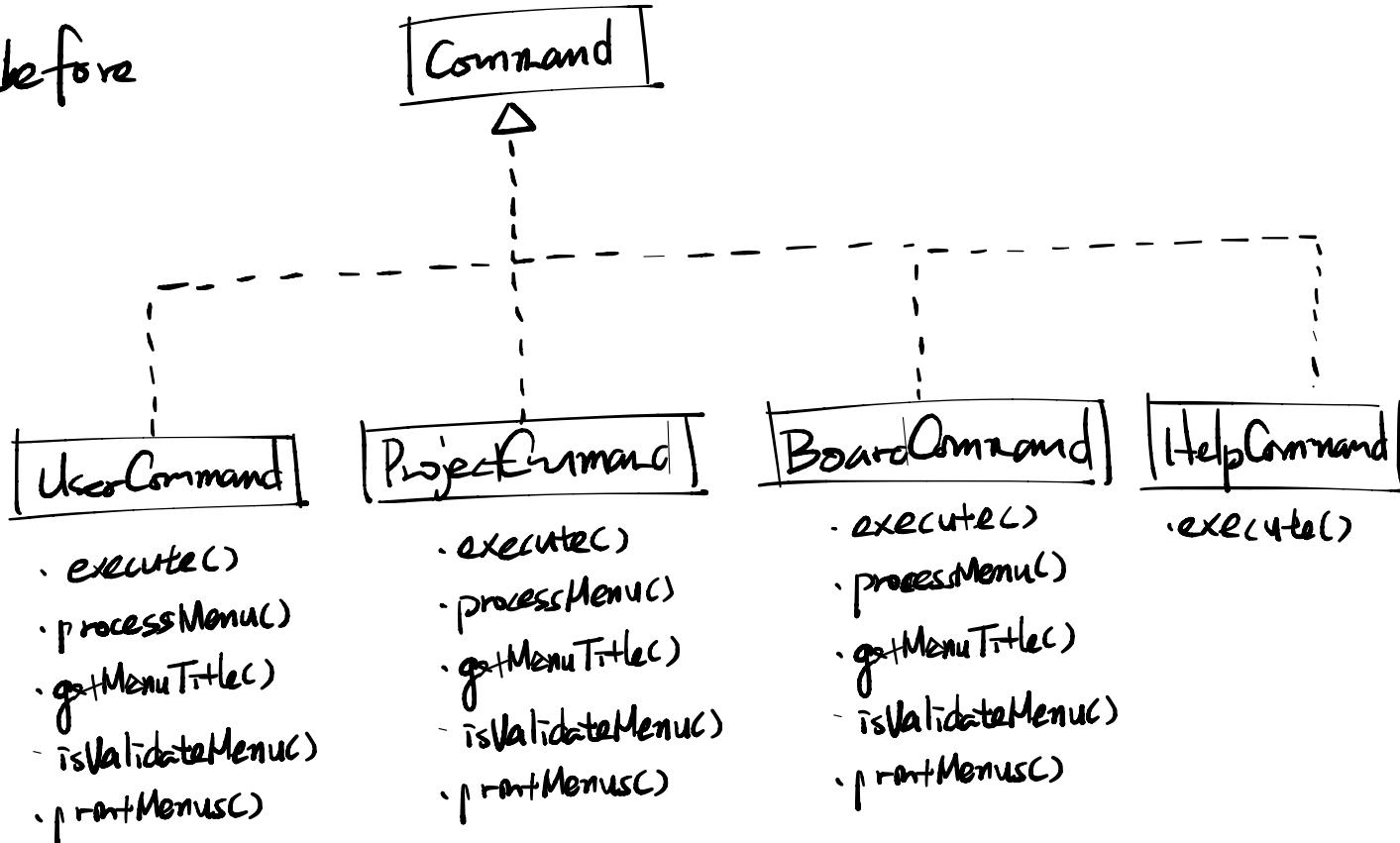
②

인스턴스 사용하기
조작할 때 험상하기!



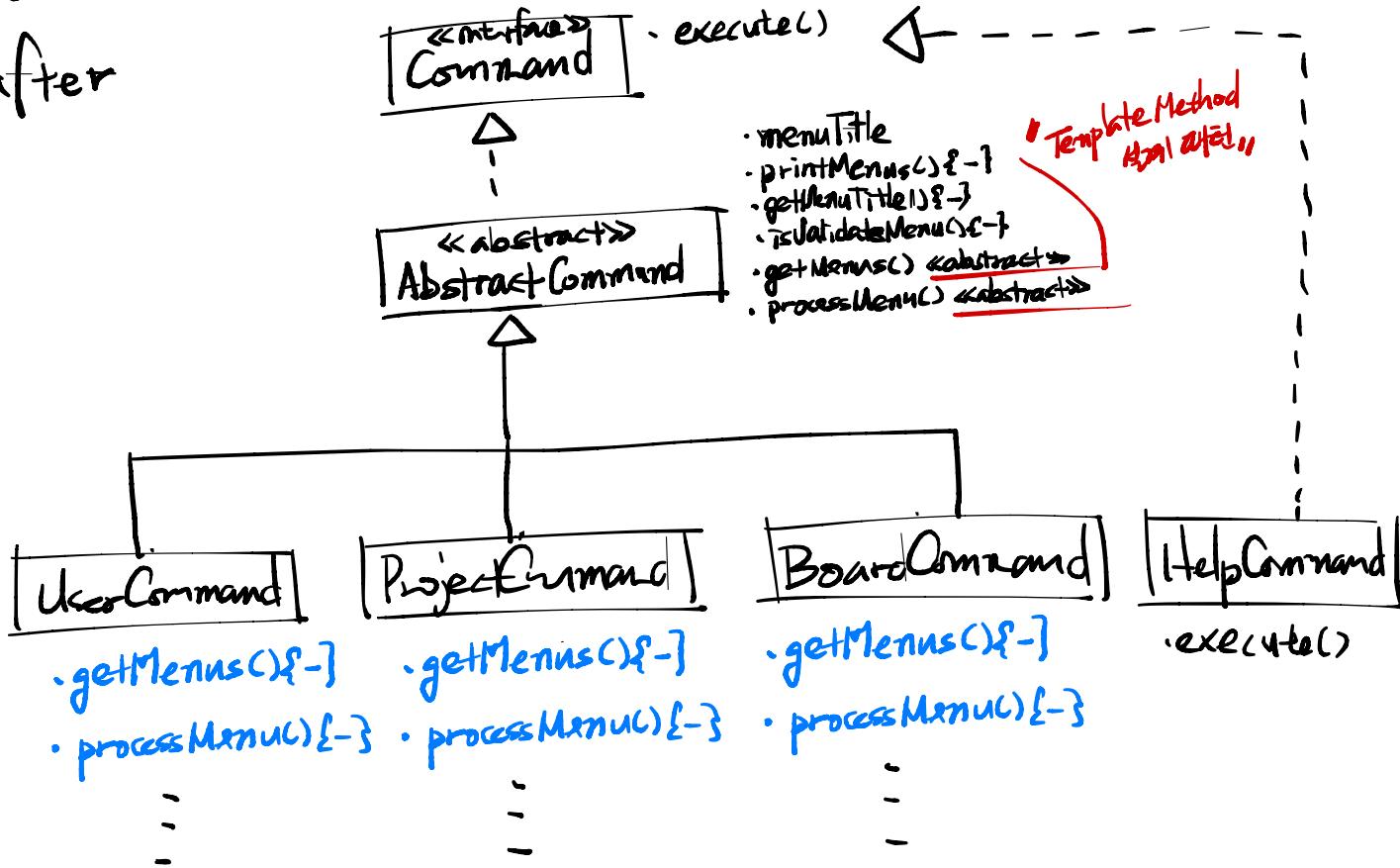
19. 상^k으로 Generalization - ②

* before

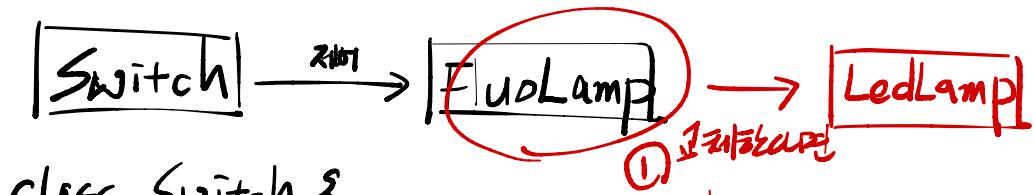


19. 一般化 Generalization - ②

* after



20. SOLID의 DIP와 GIRASP의 Low Coupling



class Switch {

Fluolamp light;
}
=
② 반드시 연결해야 한다.

- ③ Switch 클래스
Fluolamp 클래스와
상호작용 되어야
④ "강한 단점 속성" => 해결?

20. SOLID의 DIP와 GIRASP의 Low Coupling

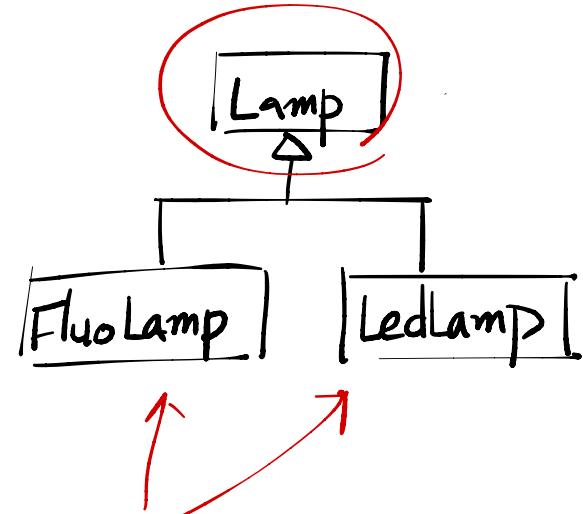


class Switch {

Lamp light;
 }
 =
 ② 아름다운 예술을 활용하여
 여러 종류의 형상을
 제작할 수 있다

③
Lamp
만들지
④

스위치로 다른 제품을 제작할 수 있어야 하는가?



① 두 클래스의 부모를 공유하는
→ 같은 작업으로 봄으면

20. SOLID의 DIP와 GRASP의 Low Coupling



class Switch {

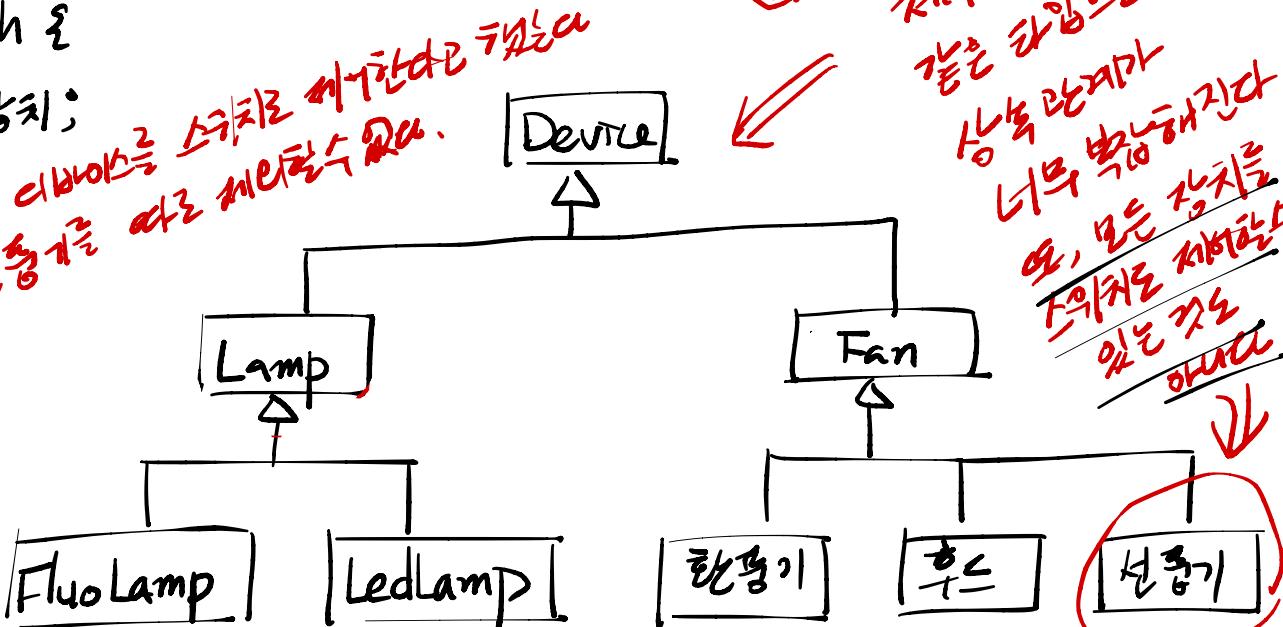
Device 장치;

}

② 모든 클래스를 스위치로 연결하는게 아니라
선택적으로 연결하는게 맞다.

① 여러 장치를 스위치로
연결하는건 고급화로 무관한데
같은 태깅으로 묶어보면
더욱 편리할 것이다
그리고 선택적으로
연결하는게 맞다
여기서도 차이점을
보여주는 것인가
아니면 예제에만
적용되는 것인가?

③
상속은 "캡슐화"의
유지보수 예로,
유연성 부족.

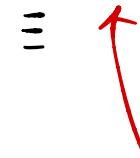


20. SOLID의 DIP와 GIRASP의 Low Coupling



class Switch {

 Switchable 광고;



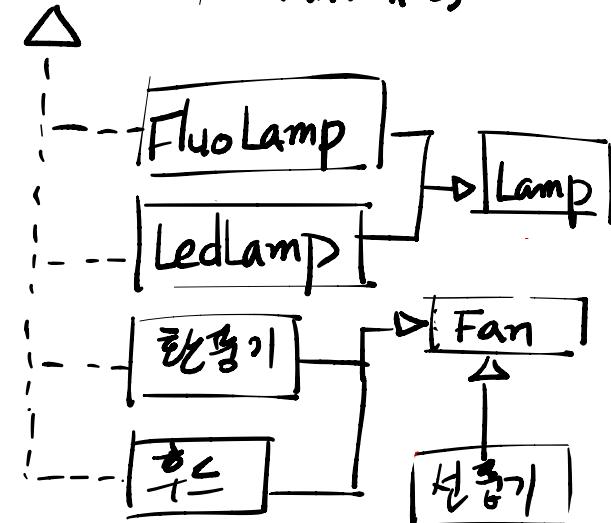
② 어떤 타입이든
Switchable 인터페이스
를 상속하는 것을 가능하게
만들기 가능

Switchable 인터페이스

- turnOn()
- turnOff()

① 품목마다
구현해야 하는
수행부

같은 추상이 아니
구현체는
다양하므로
제작할 수 있다



↳ ② 품목마다 더 유연하다 “약관화” = 느슨한 연결
Low Coupling

* SOLID - Dependency Inversion Principle (DIP)

↳ 의존 객체를 확장 만들지 않고

외부에서 주입 받는 방식.



class Switch {
 Switchable 광고;

① 노드한 광고로
전환한 후

FluoLamp light1 = new FluoLamp();

Switch switch = new Switch(light1);

② Switch가 의존 객체를
생성하는 것 아니고
외부에서 주입 받는
방식으로 전환하면

- ③
- ✓ 광고가 된다
 - ✓ 광고가 된다.

↳ 의존 객체를
간단히 만들어 주입하는
스위치의 용도를 헤아릴 수 있다.

이 유연해진다

* DI

SOLID

Dependency
Inversion
Principle

(의존성 역전 원칙)

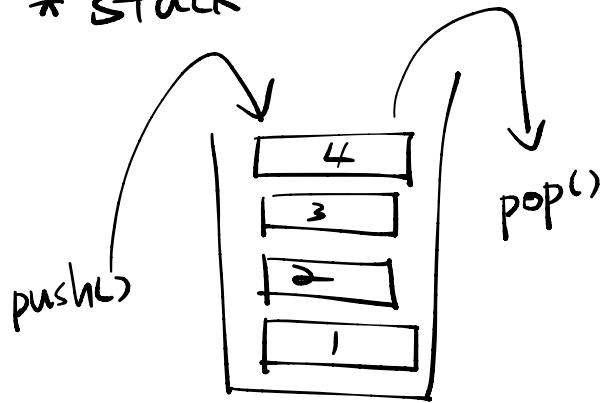
(제어권역)

Inversion of Control (IoC)

- ① Dependency Injection
② Listener = event handler

21. 자료구조 - Stack 와 Queue

* stack

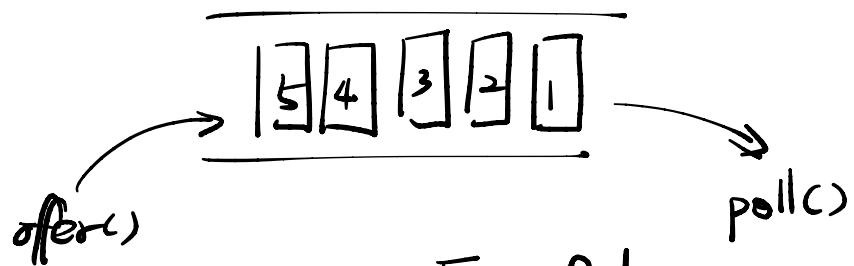


First In Last Out
(FILO)

"
Last In First Out

(LIFO) ① 뒤로나오기 ② 앞으로나오기
③ 예상외나오기

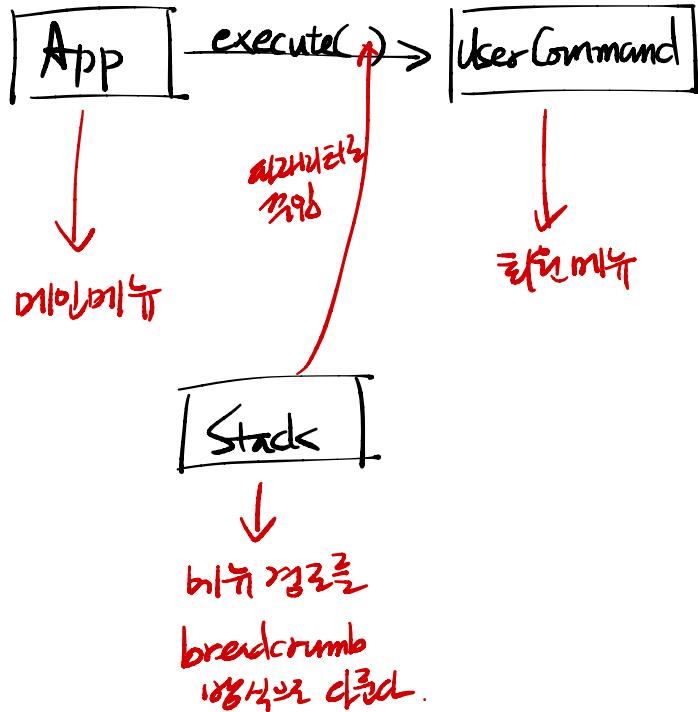
* Queue



First In First Out
(FIFO)

- ① 예상
- ② 앞으로나오면서 큐를 나온다 = 정상 처리
- ③ 예상외나오기 = Event Queue

* 디足迹 제목을 소리으로 하기



* String

String str = "";

str += "aaa";

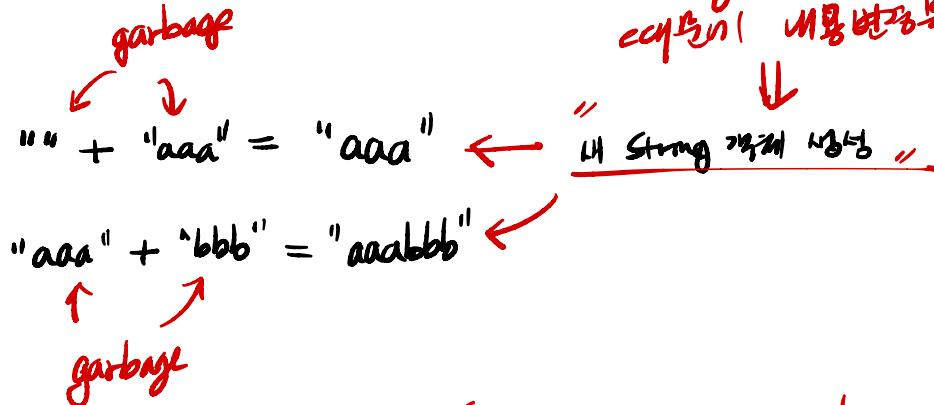
str += "bbb";

★ 왜 Java에서는
String은 오직 같은
StringBuffer는
StringBuilder는
StringBuffer와
StringBuilder는
같은 특성을 가진다.
↳ garbage는 같은 특성을 가진다.

thread-Safe

동시접근 방지 가능
↳ lock/unlock
함수 안에서

StringBuffer, StringBuilder



Strong immutable \Rightarrow 안전한
copy는 내용을 바꾸지 않!

문자열은 대량의 내용이 \Rightarrow Strong \Rightarrow 안전한
문자열은 garbage \Rightarrow 안전한
문자열은 thread-Safe.

thread-Safe \Rightarrow 안전한



mutable \Rightarrow 안전한

thread-Safe \Rightarrow 안전한
↳ thread-Safe의 경우 내용은 바뀌지 않
↳ 다른 스레드의 경우 내용은 바뀌지 않
↳ lock/unlock을 통해 내용은 바뀌지 않
↳ obj → lock/unlock → 안전한

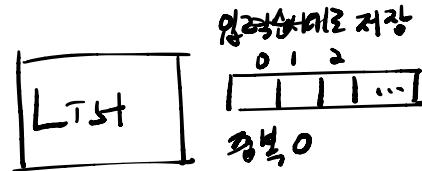
22. Iterator 깊이 파악

이전 문서화하는 강점인 하위 목록

· 재사용성↑ · 유지보수성↑ · SOLID / GRASP 부합

문서
사용 주제에 따라
선택하는
수행하는
방법성이
다르다!

방식 (정수)
get()



toArray()



key 가짐
get()

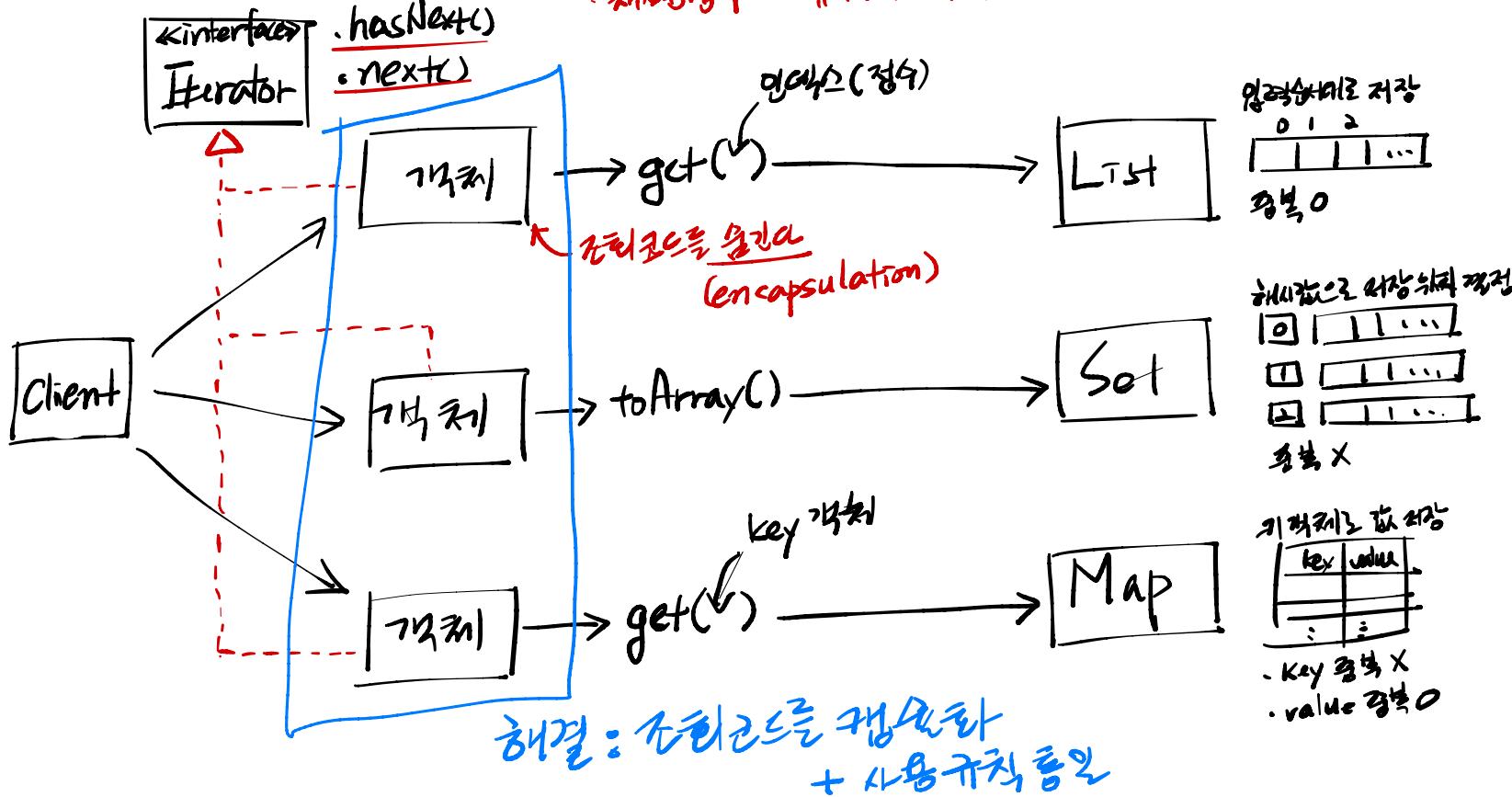


- Key 정복 X
- value 정복 O

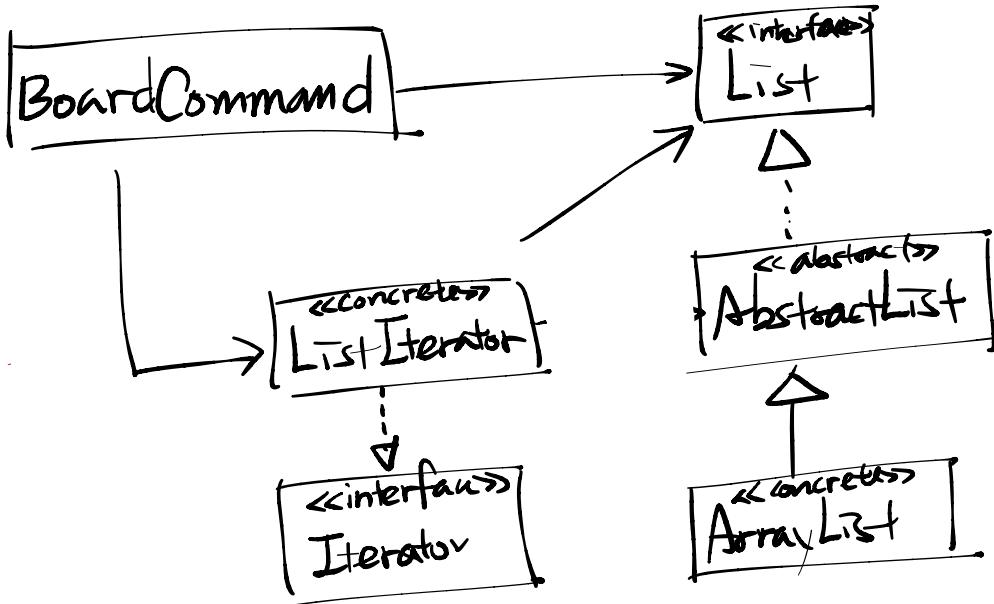
22. Iterator 기능적 패턴

이전 문서화하는 강제된 흐름 방식

· 재사용성↑ · 유지보수성↑ · SOLID / GRASP 부합

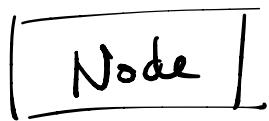


22. Iterator 틀 구조



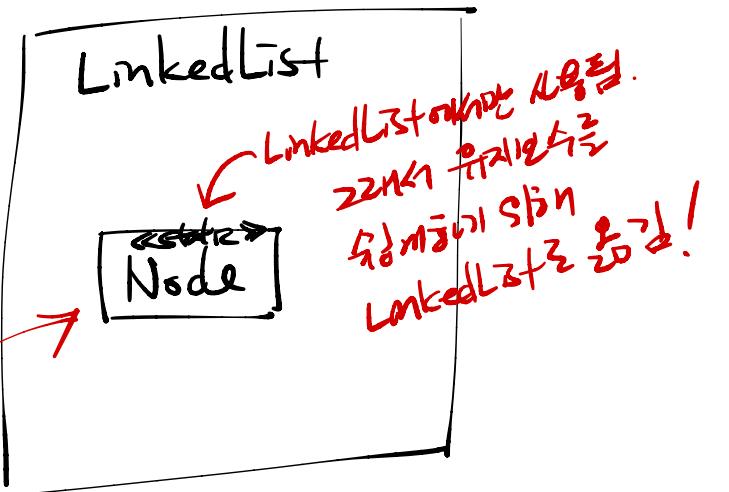
23. 중첩 클래스

before



↑
package member class

after



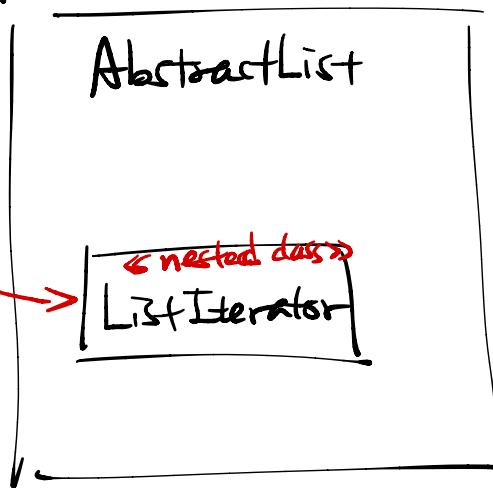
Nested class
(static nested class)

23. 중첩 클래스

before



after



* enhanced for 문법

for(변수선언 : 배열 또는 Iterable 구현체)

ex) String[] names = {"홍길동", "임꺽정", "유관순"};

for(String name : names) { }

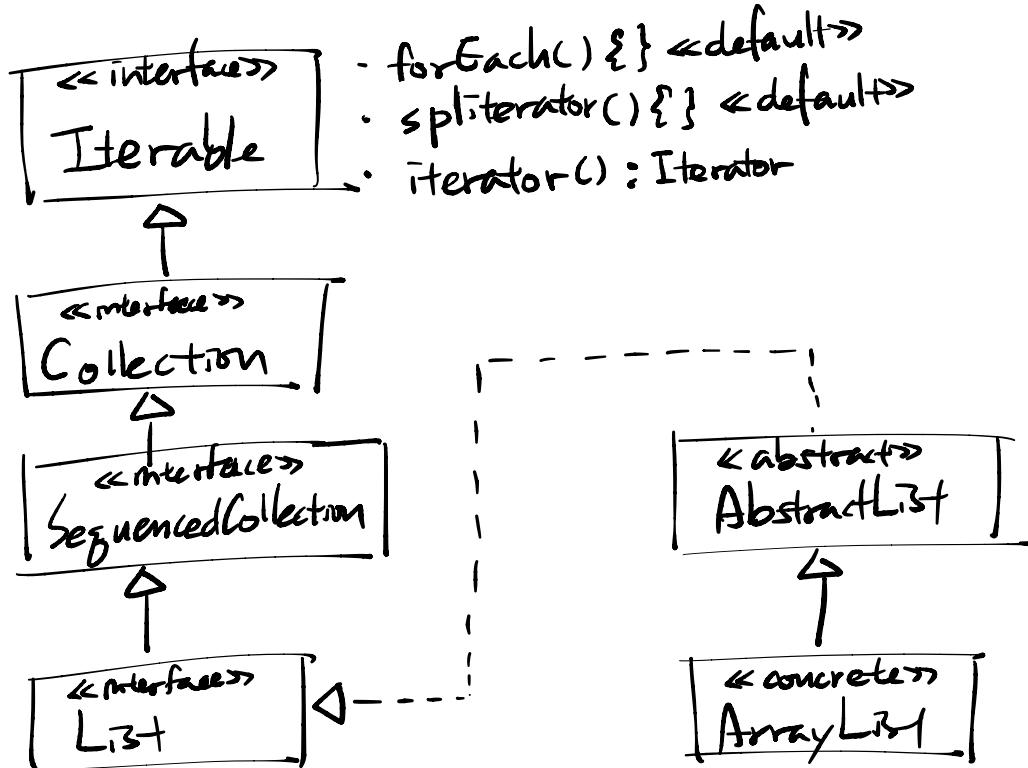
변수

ex) ArrayList list = new ArrayList();
list.add("홍길동"); list.add("임꺽정"); list.add("유관순");

for(Object item : list) { }

Iterable 구현체

* Iterable ↗^{3연자}



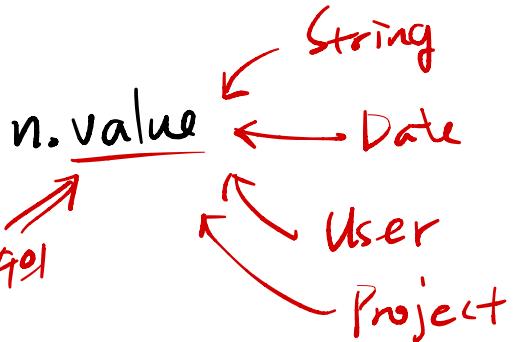
24. Generic 타입 사용하기

```
class Node {
```

```
    Object value;
```

특정 타입의
인스턴스
만들기 위해
제한할 수
있을까?

```
Node n = new Node();
```



다형화 변수의
특성
하여정 변수의
특성
제한하는
목적.

⇒ Generic 타입

24. Generic 블록 사용하기

class Node<what> {

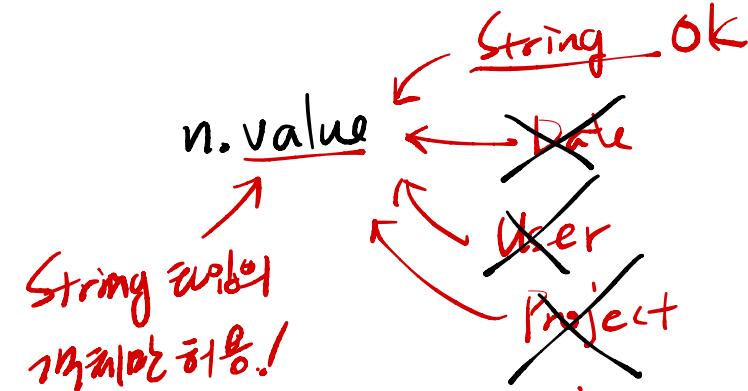
what value;

what이
어떤 타입인지

선택할 때
제한한다

타입 제한자 = 타입 정보를 넣은 변수

Node<String> n = new Node<String>();



제한자는 아니
타입 제한자로써는
타입 제한자로
제한 가능

* Type Parameter

↑ 타입 명보를 뱉는 변수

이거 { T (Type)
E (Element)
K (key)
V (Value)
S, U, V