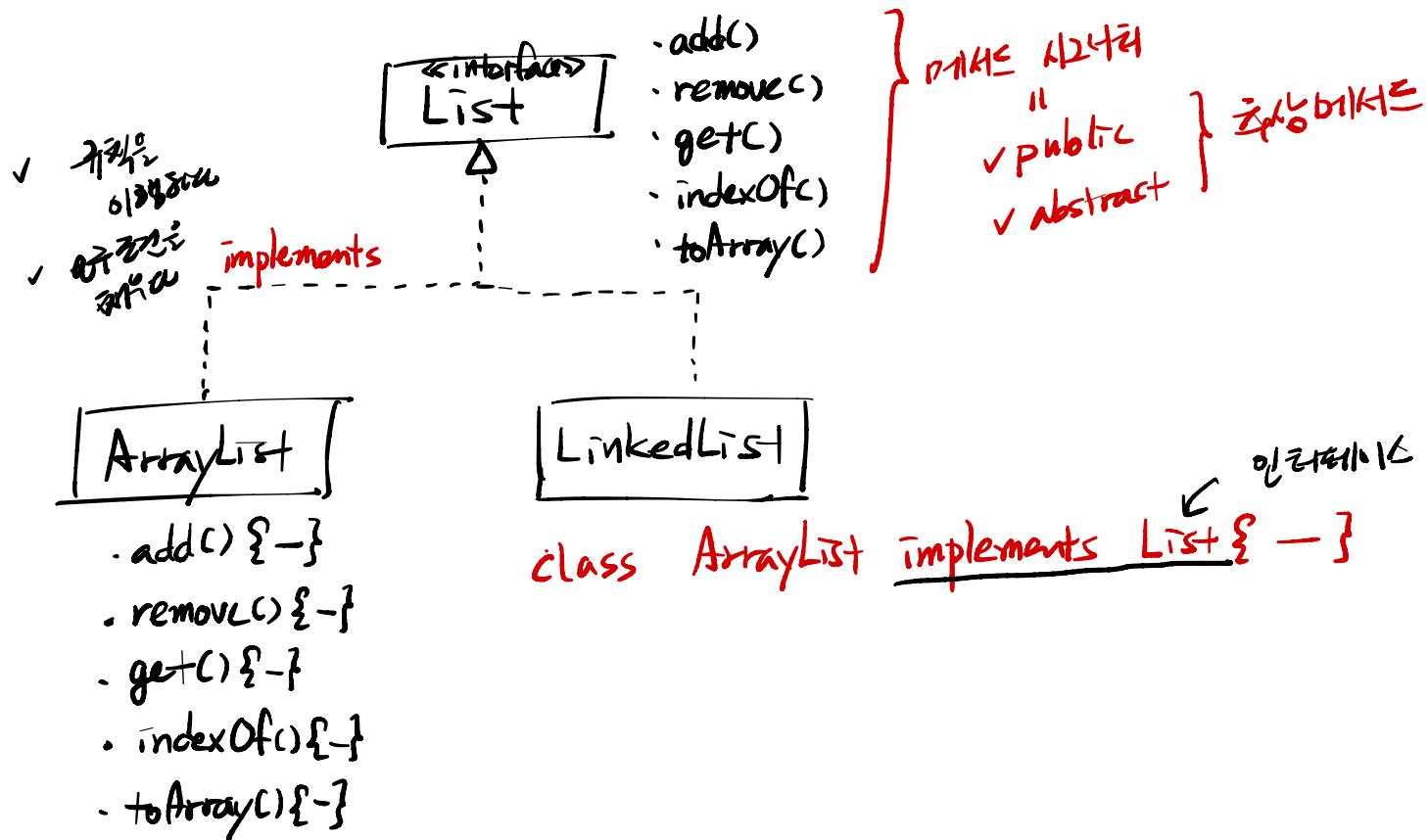
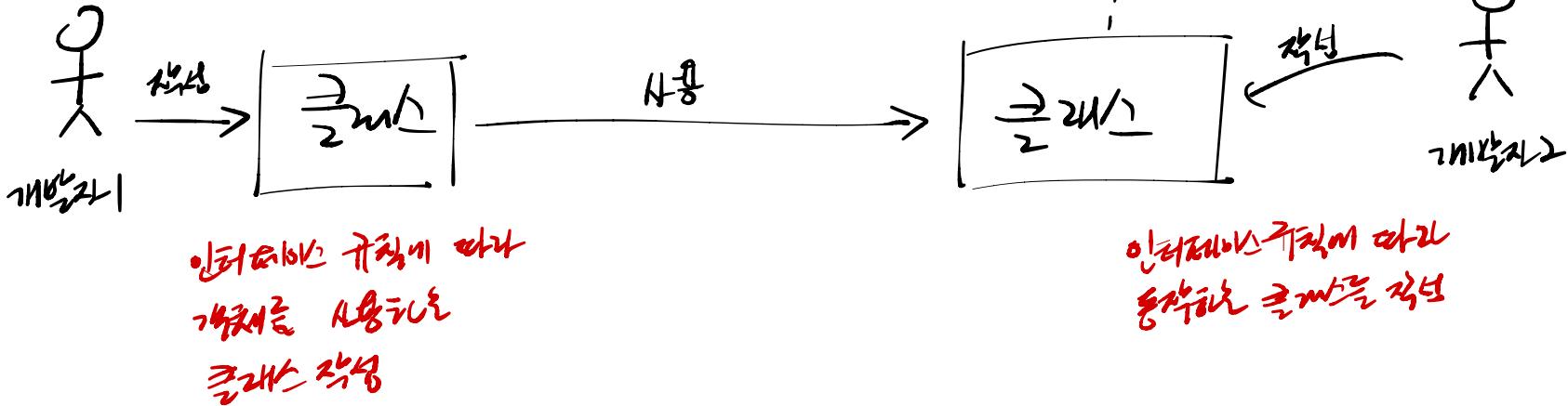
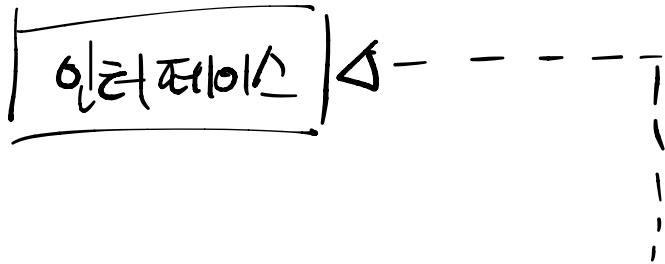


17. 인터페이스를 이용한 구조화된 접근



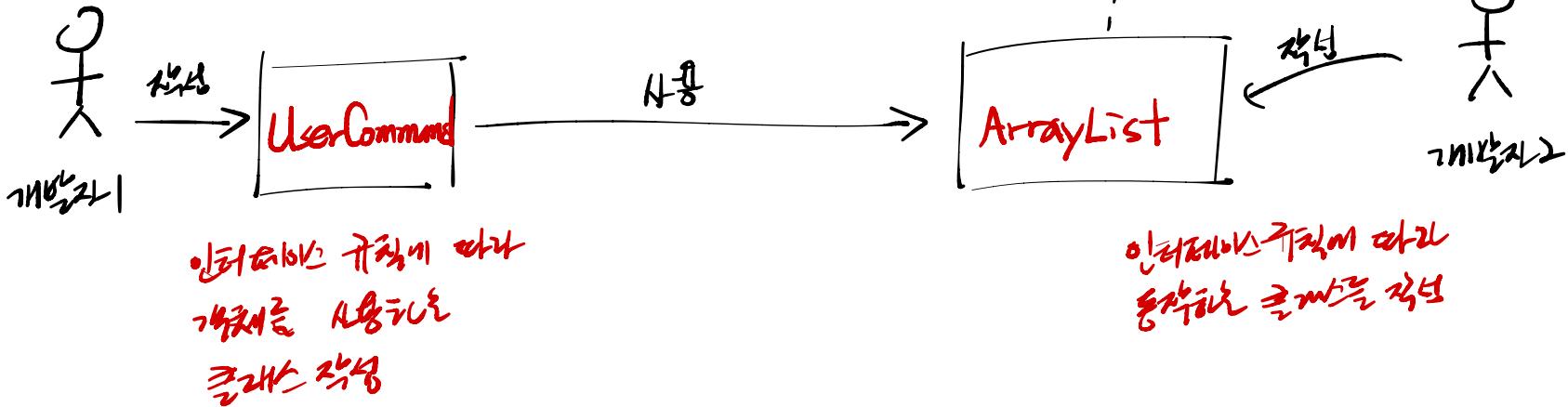
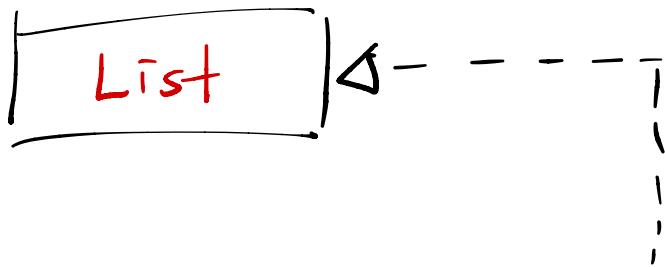
* 인터랙이스

기본 사용자인 경우에 보면
 ① 프로그램의 일반성이 만족할 수 있다.
 ② 교체가 가능하다.

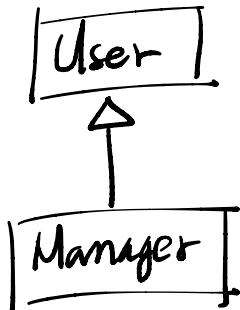


* 인터페이스

인터페이스는 구현체 없이
제공하는 기능의 만족도가 높다.



* instanceof vs getClass()



`equals() {`

≡

`}`

`equals(Object obj) {`

step ①

`if (this.getClass() != obj.getClass()) {`

`return false;`

`}`

step ②

`if (!(obj instanceof User)) {`

`return false;`

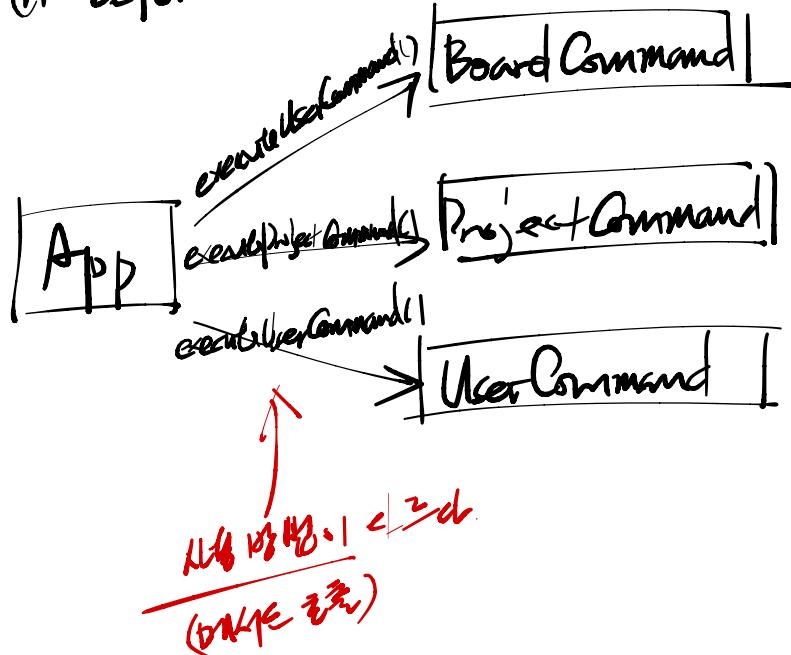
`}`

`User u1 = new User();`
`Strong str = new Strong();`
`User u2 = new User();`
`Manager m = new Manager();`

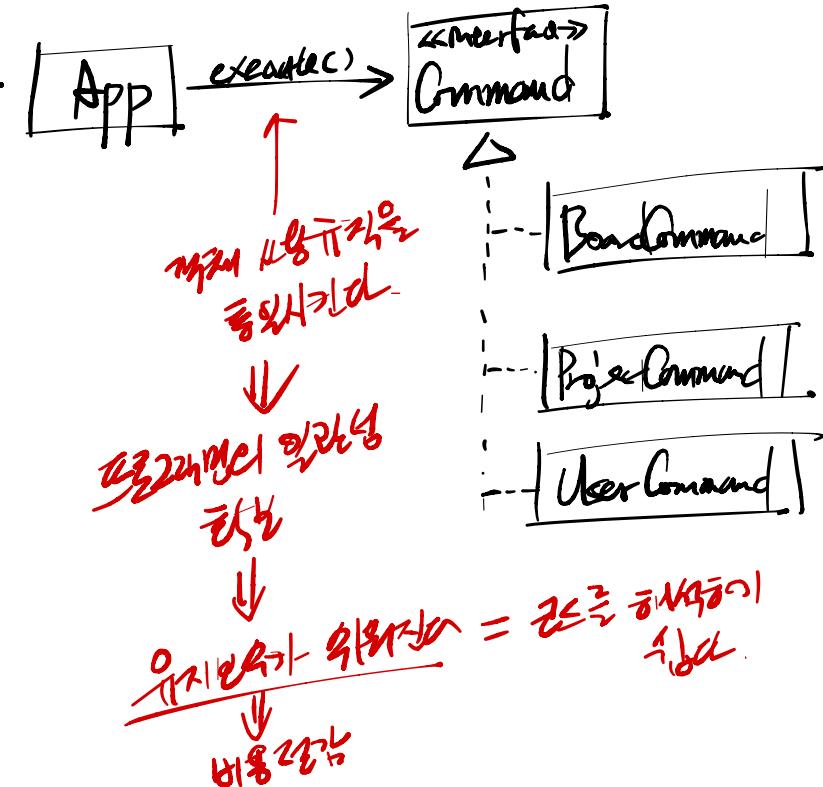
	u1.equals(str)	u1.equals(u2)
<code>u1.equals(str)</code>	F	F
<code>u1.equals(u2)</code>	T	T
<code>u1.equals(m)</code>	F	T

* 121 능력과 122 번의 학습자료

① before

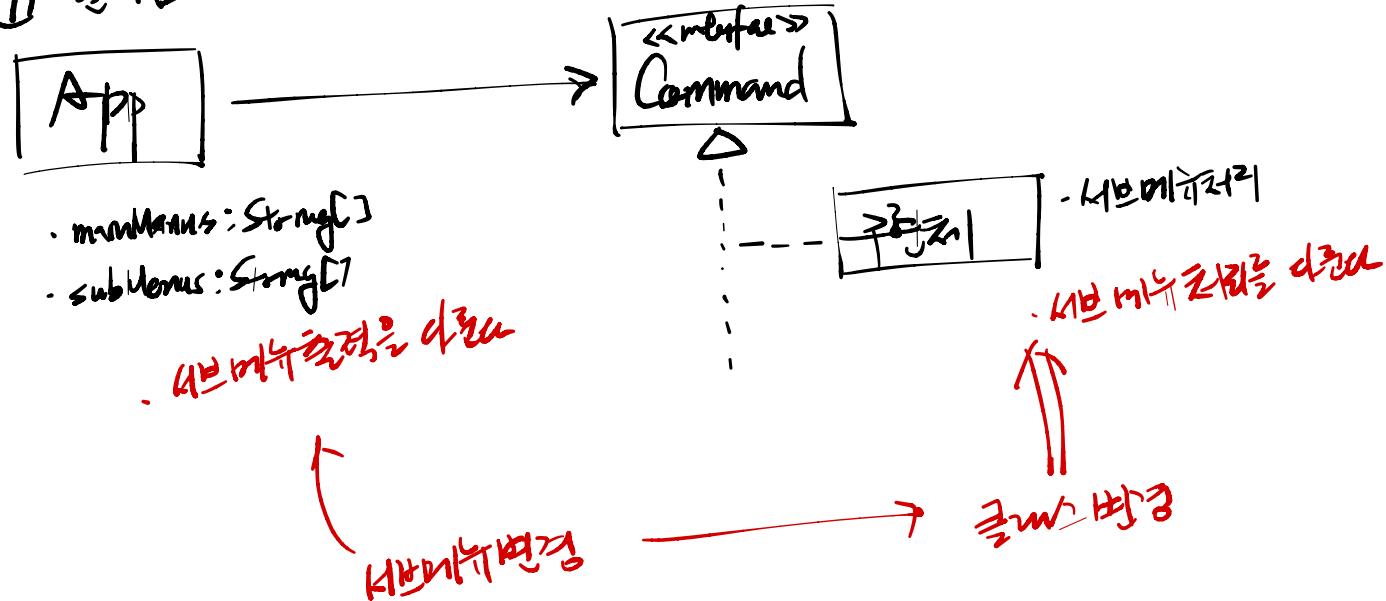


② after



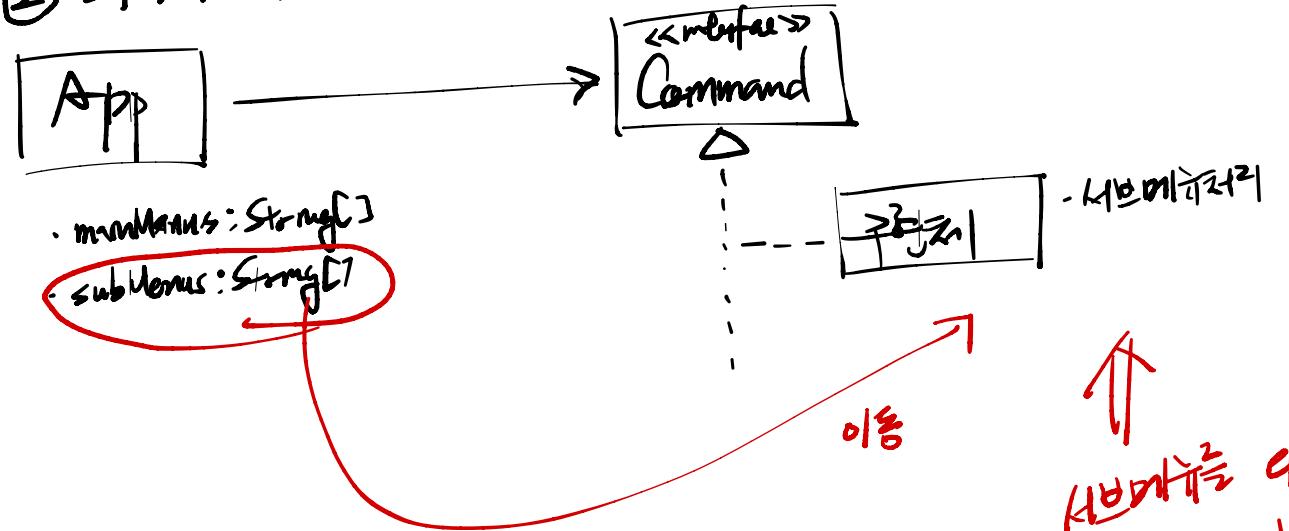
18. 커맨드 패턴

① 응용



18. 디자인 패턴

② 명령 : GRASP의 High Cohesion & Low Coupling



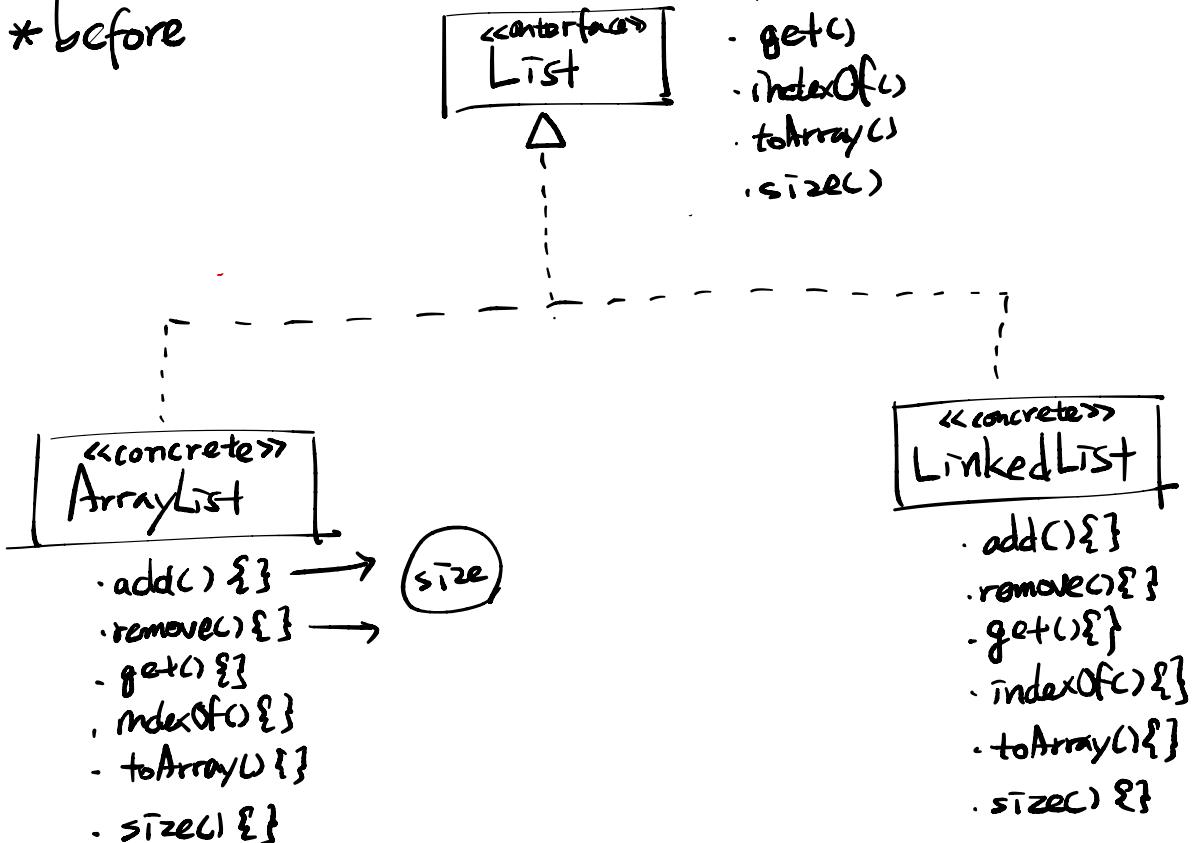
- 메뉴 이동

↑
메뉴 이동
Command 구현하기
방법 .

||
이동 메뉴를 실행

19. 상속의 Generalization - ①

* before



19. 상속의 Generalization - ①

* after

①

상속을 통한
공통 기능은 대상으로
설정하는 경우
하는 경우

②

인스턴스 사용하기
조작 가능성이 있다!



- add()
- remove()
- get()
- indexOf()
- toArray()
- size()
- size: int
- size() { } →



- add() { } → size
- remove() { } → size
- get() { }
- indexOf() { }
- toArray() { }

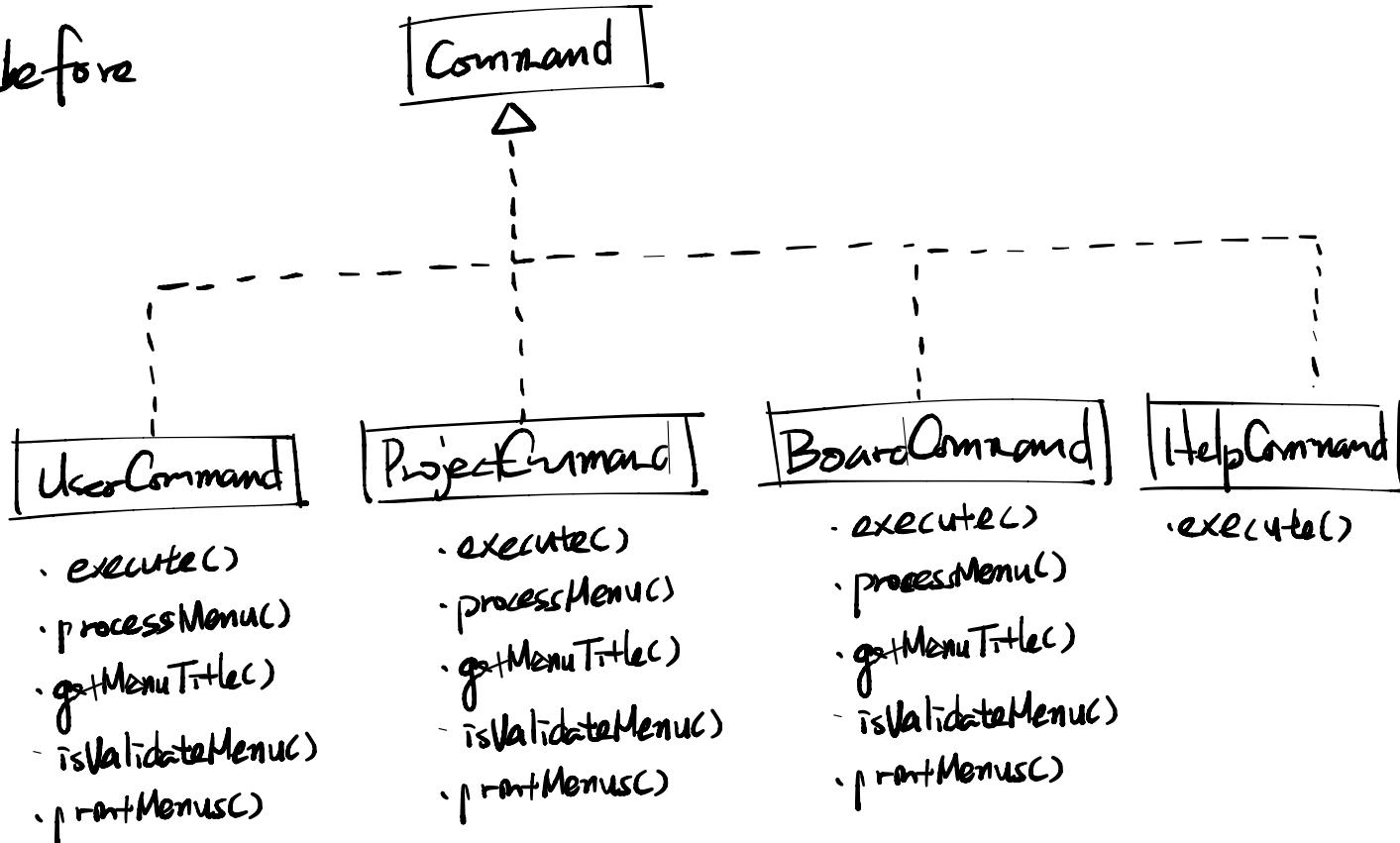


- add() { }
- remove() { }
- get() { }
- indexOf() { }
- toArray() { }

인스턴스를 생성해서
변수 값을 >= m↑

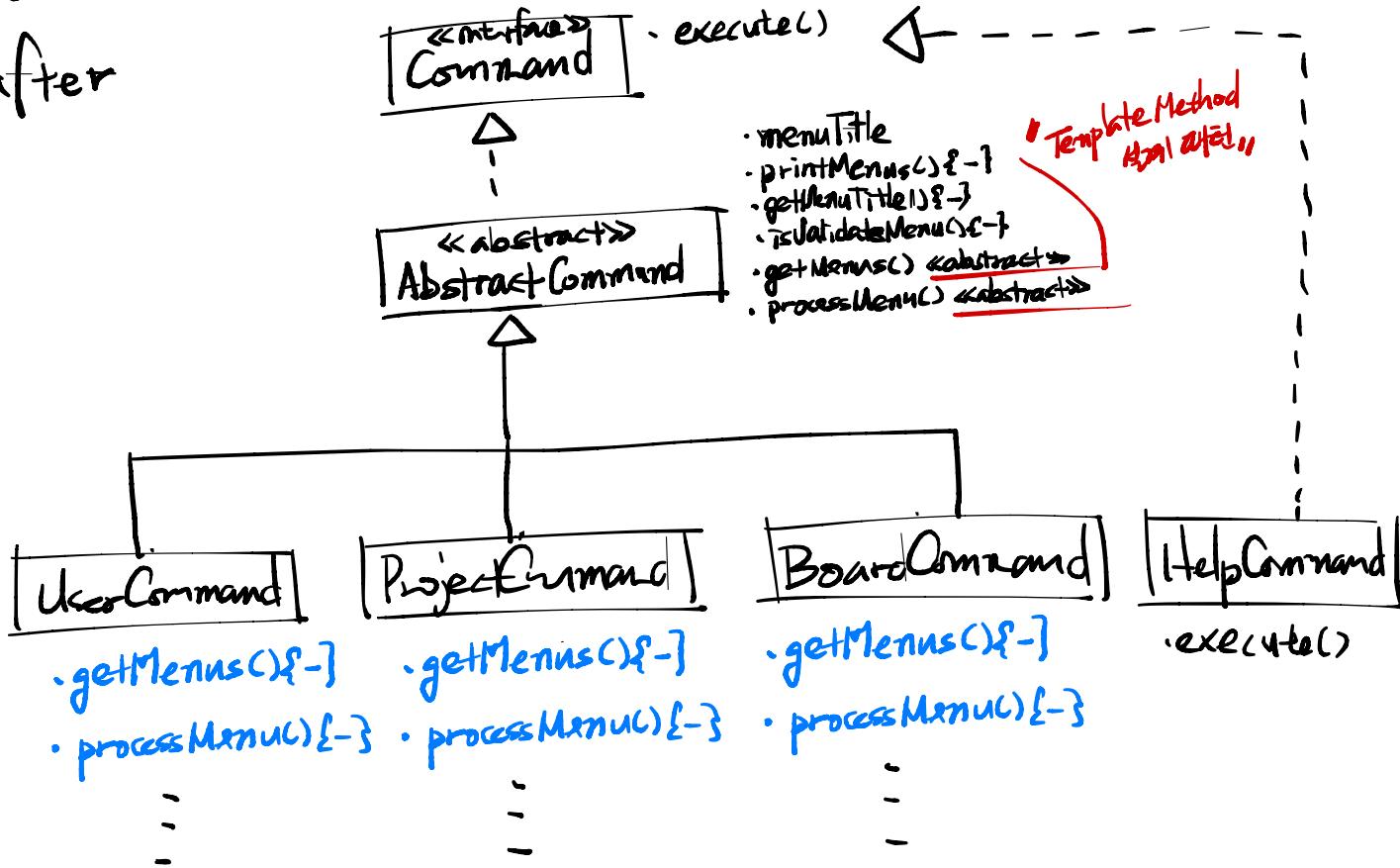
19. 상^k으로 Generalization - ②

* before

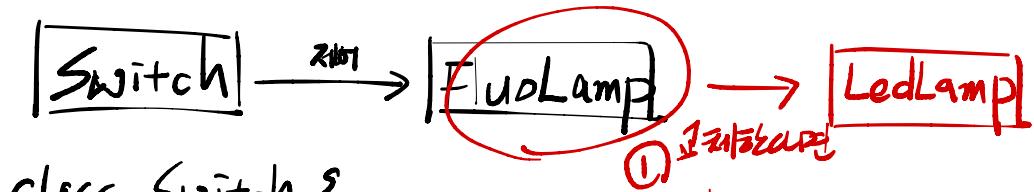


19. 一般化 Generalization - ②

* after



20. SOLID의 DIP와 GIRASP의 Low Coupling



class Switch {

 FluoLamp light;
 \equiv ② 반드시 연결해야 함.

③ Switch 클래스가

FluoLamp 클래스와

상호로 연결되어야

④ "강한 단계 관계" \Rightarrow 해결?

20. SOLID의 DIP와 GIRASP의 Low Coupling

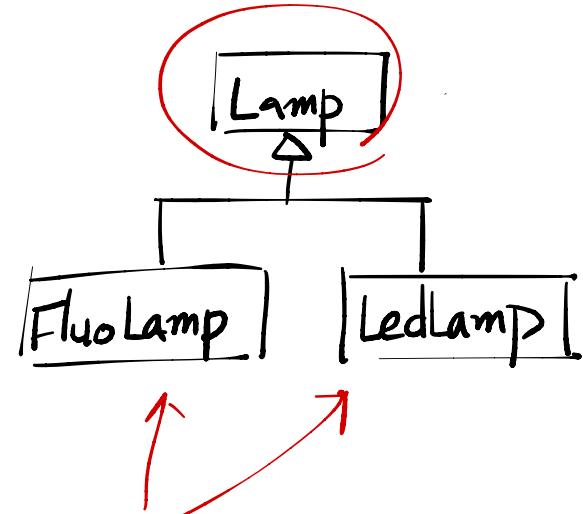


class Switch {

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 {

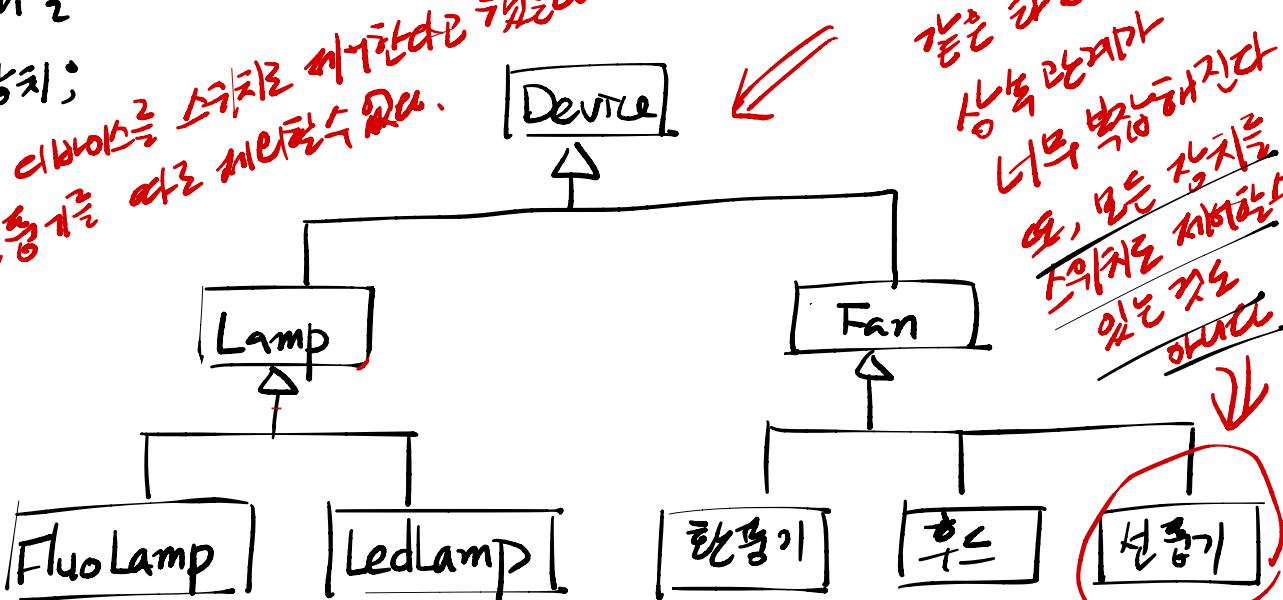
 Device 장치;

}

*② 모든 클래스를 스위치로 대체할 수 있다.
인접 클래스는 제외된다.*

① 여러 장치를 스위치로
대체하거나
같은 태스크로 묶거나
다른 장치가
너무 복잡해지면
여기, 모든 장치를
스위치로 처리하는
있는 것도
있다.

③
상속은 "간접화"의
유익성이 예술,
유연성 부족.



20. SOLID의 DIP와 GIRASP의 Low Coupling

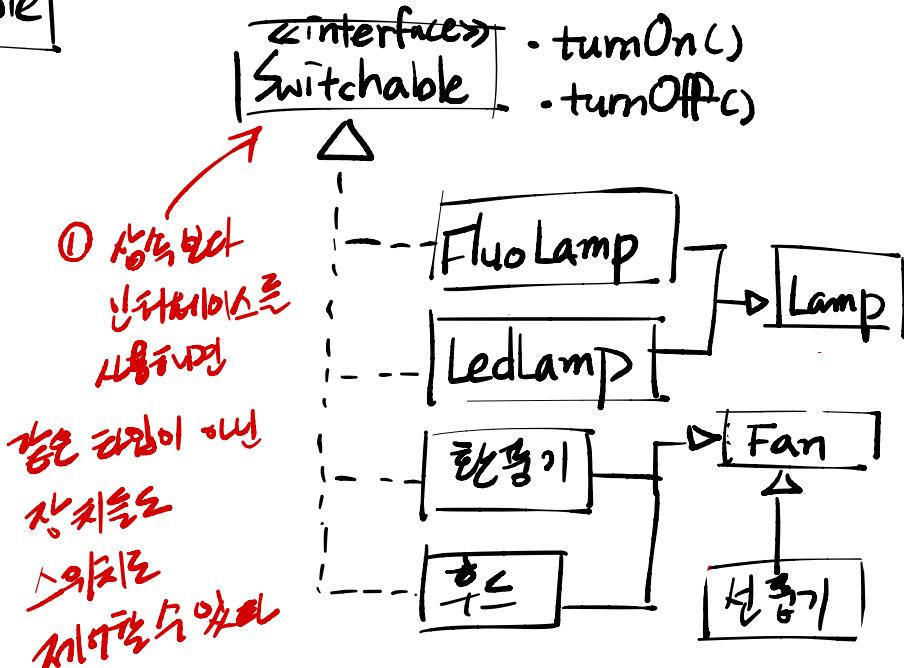


class Switch {

 Switchable 장치;

}

② 어떤 타입의든
Switchable 구조에
적합하는 것을 알아내기
가능하게



↳ ② 쌍방향 인터페이스로 “약관화” = 느슨한 연결

Low Coupling

* SOLID - Dependency Inversion Principle (DIP)

↳ 의존 구조를 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 구조를
설정하는 것 아니고
외부에서 주입 받는
방식으로 전환하면

- ③
✓ 광고가 된다
✓ 광고가 된다.

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

이 유연해진다

* DI

SOLID

Dependency
Inversion
Principle

(의존성 역전 원칙)

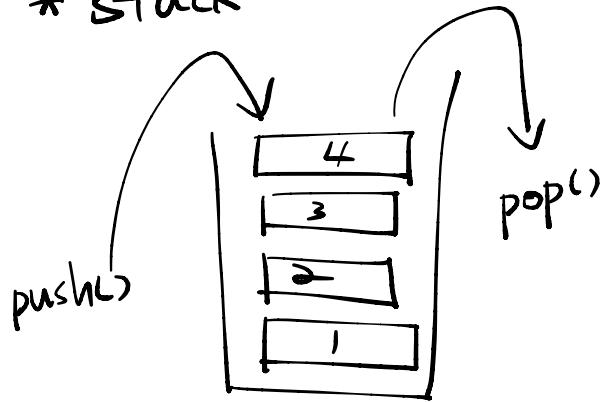
(제어권역)

Inversion of Control (IoC)

- ① Dependency Injection
② Listener = event handler

21. 자료구조 - Stack 와 Queue

* stack

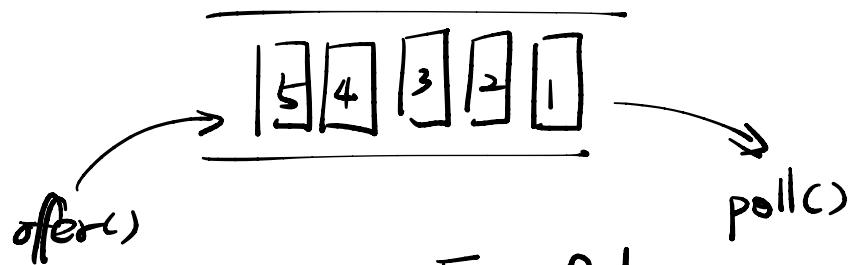


First In Last Out
(FILO)

"
Last In First Out

(LIFO) ① 뒤로나오기 ② 앞으로나오기
③ 예상외나오기

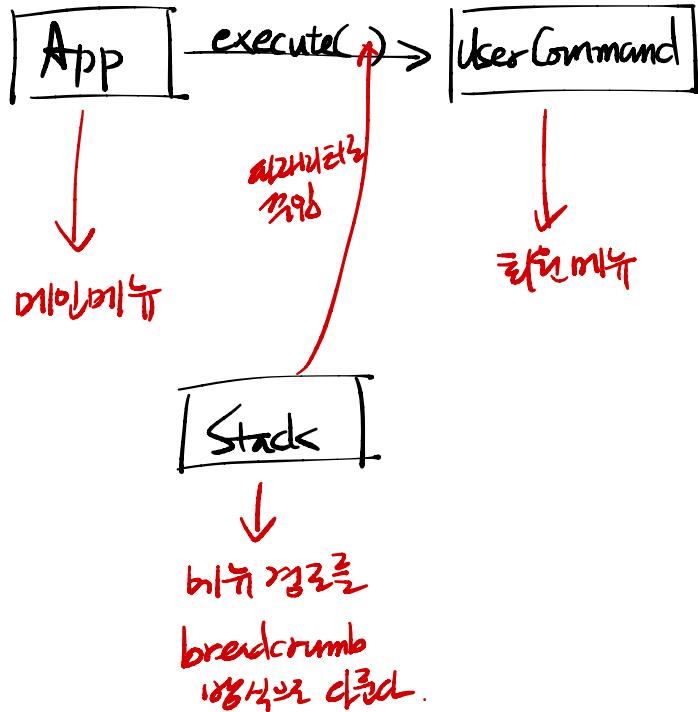
* Queue



First In First Out
(FIFO)

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

* 디足迹 제목을 소리으로 하기



* String

String str = "";

str += "aaa";

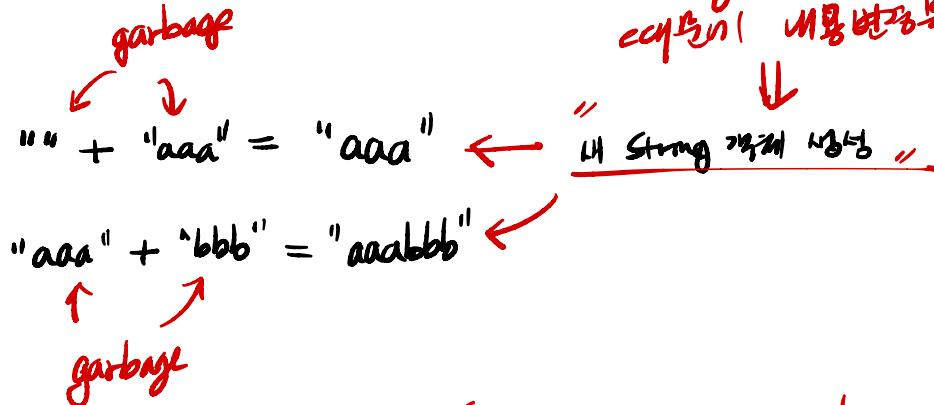
str += "bbb";

★ 왜 Java에서는
String은 오직 같은
StringBuffer는
StringBuilder는
이유는 그다지
아니다.
↳ garbage는 끊임없이
생성되고 해제된다.

thread-Safe

스레드 안전한지 알기
↳ lock/unlock
함수는 안전

StringBuffer, StringBuilder



Strong immutable \Rightarrow 안전한
copy는 만들지 않는다!

문자열은 대체로 같은 1H Strong \Rightarrow 안전하다
1H String은 \Rightarrow garbage \Rightarrow 안전.
문자열은 안전하다.

thread-Safe \Rightarrow 안전한지 알기
↳

thread-Safe \Rightarrow 안전한지 알기
↳ 스레드 안전한지 알기
↳ lock/unlock 함수는 안전
↳ obj锁 \Rightarrow 안전

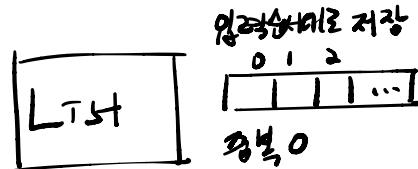
22. Iterator 깊이 파악

이전 문서화하는 강점인 하위 목록

· 재사용성↑ · 유지보수성↑ · SOLID / GRASP 부합

문서
수집하는
수집하는
방법이
다르다!

인덱스(정수)
get()



toArray()

Set

한시점으로 저장하는 형식

0	1	2	...

정복 X

key 가짐
get()

Map

기본적으로 값 저장

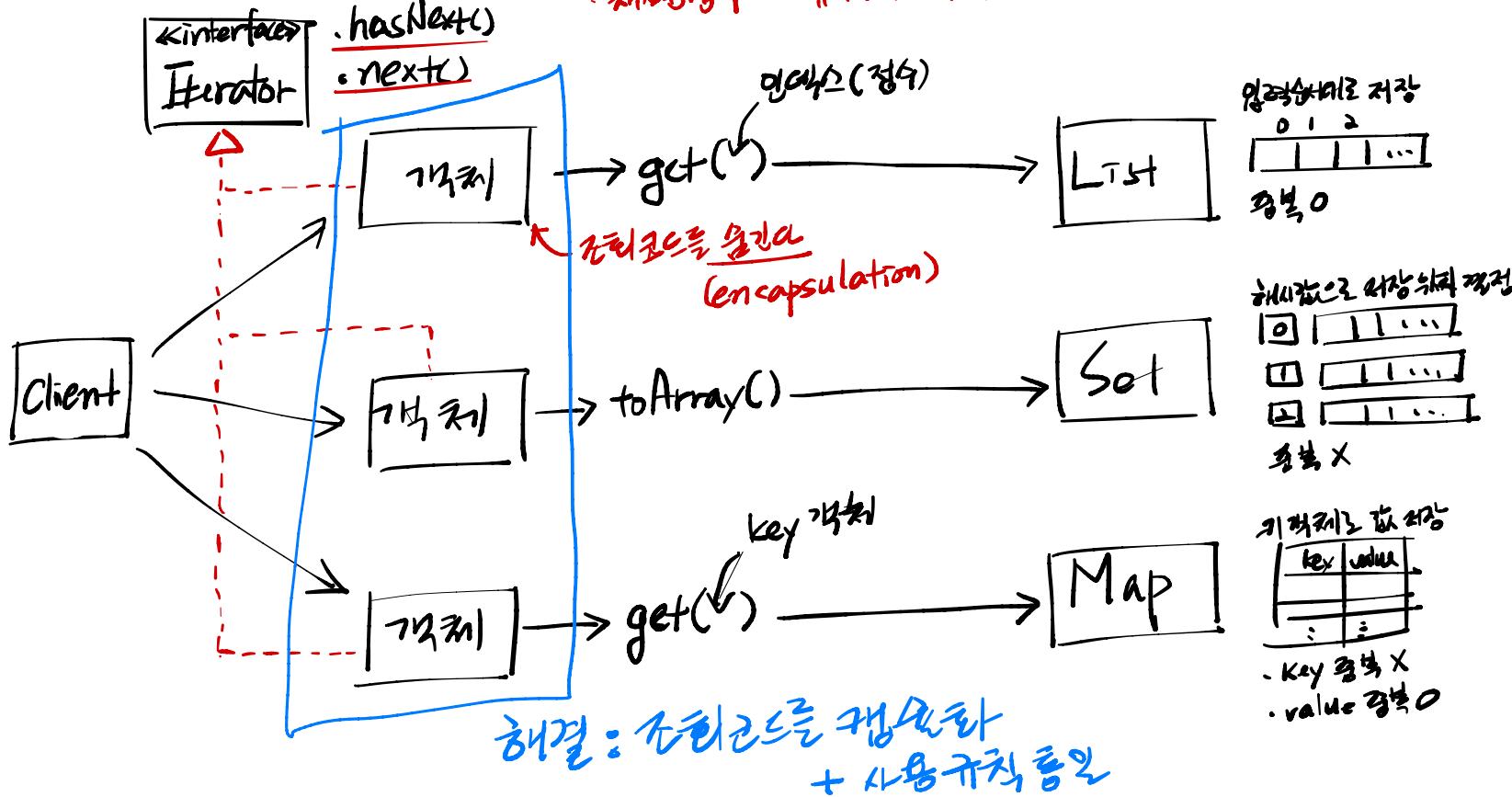
key	value

- Key 정복 X
- value 정복 O

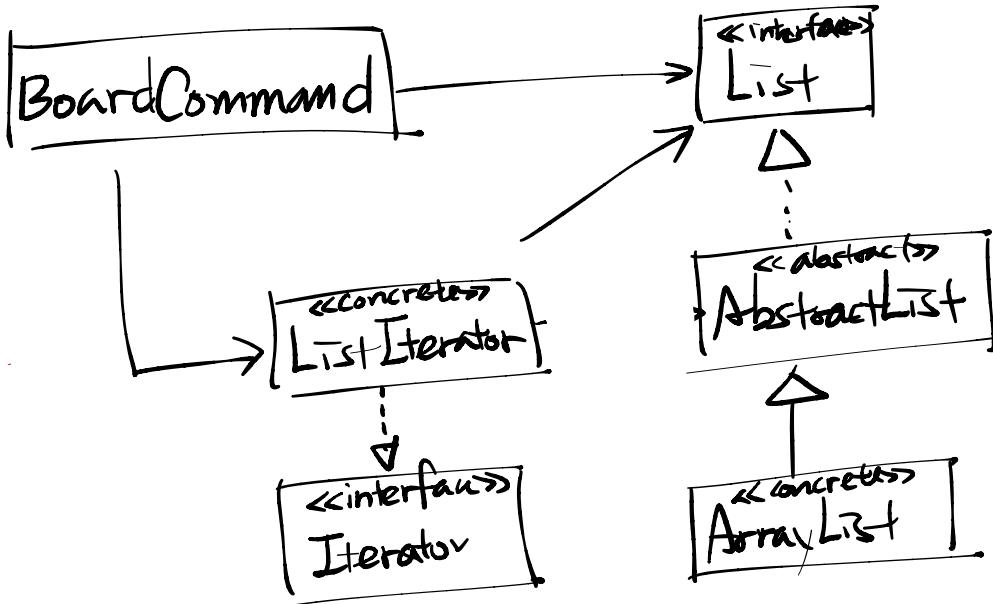
22. Iterator 기능적 패턴

이전 문서화하는 강제된 흐름 방식

· 재사용성↑ · 유지보수성↑ · SOLID / GRASP 부합

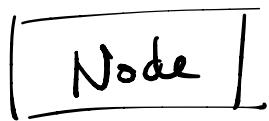


22. Iterator 틀 구조



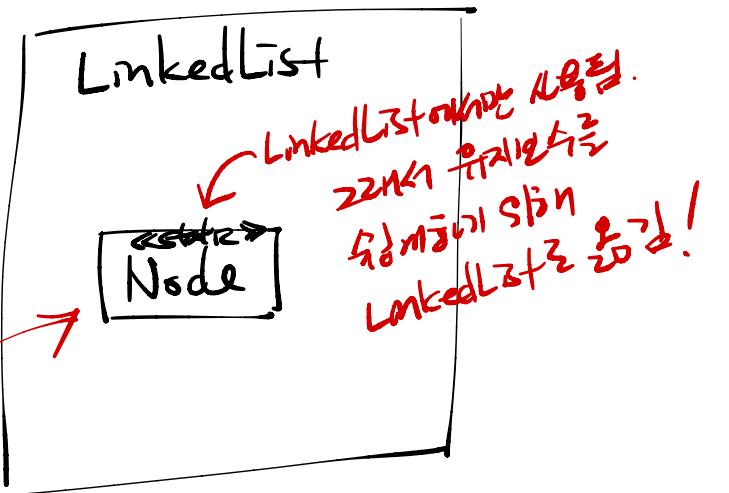
23. 중첩 클래스

before



↑
package member class

after

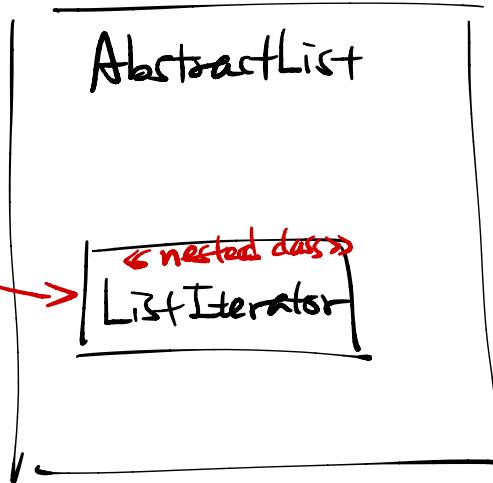


23. 중첩 클래스

before



after



* enhanced for 문법

for(변수선언 : 배열 또는 Iterable 구현체)

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

for(String name : names) { }

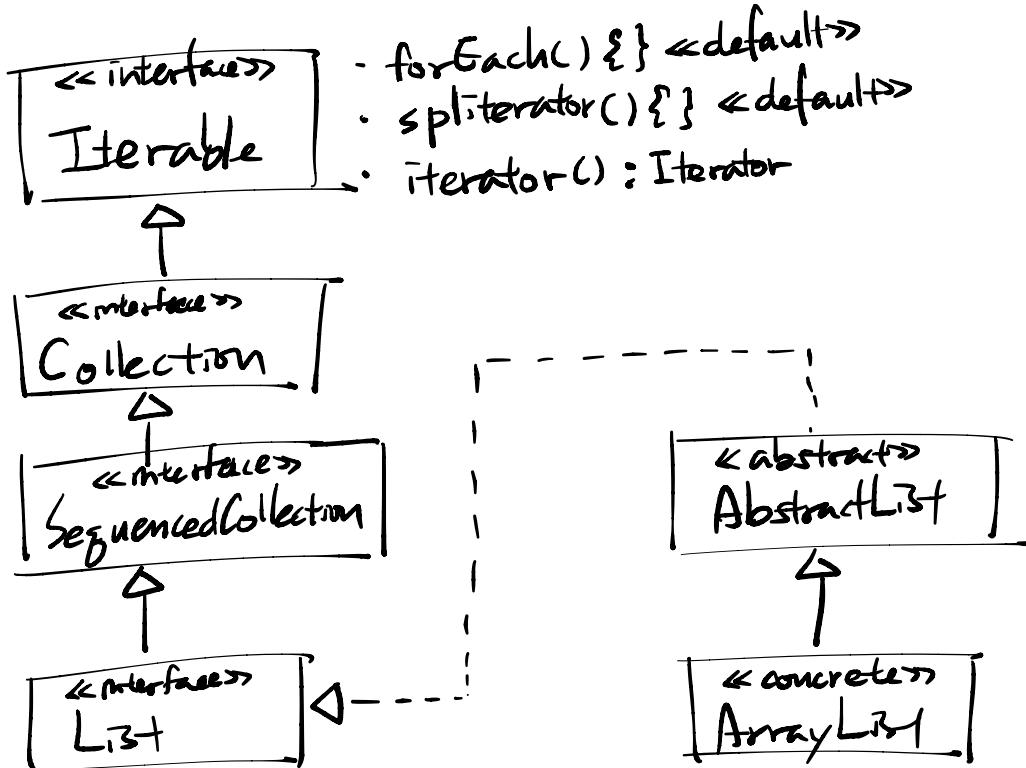
변수

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

for(Object item : list) { }

Iterable 구현체

* Iterable ↗^{3연자}



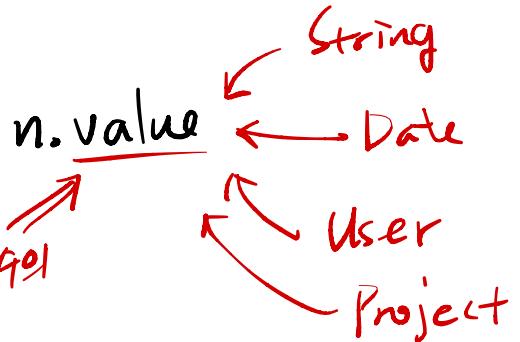
24. Generic 타입 사용하기

```
class Node {
```

```
    Object value;
```

특정 타입의
인스턴스
만들기 위해
제한할 수
있을까?

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



다양한 변수의
형식
하나의 타입의
사용 가능하기
때문.

⇒ Generic 타입

24. Generic 블록 사용하기

class Node<what> {

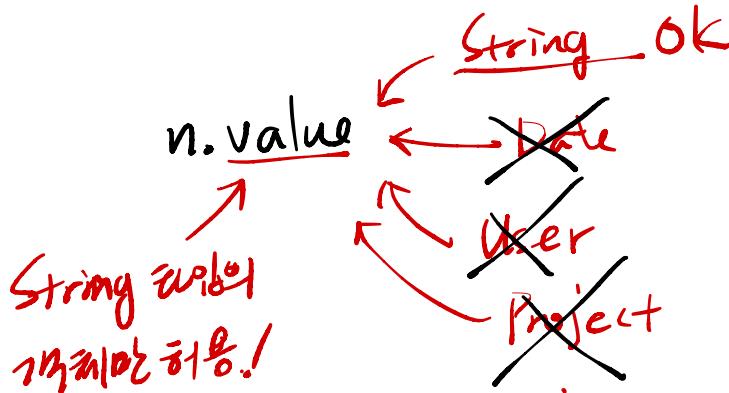
what value;

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

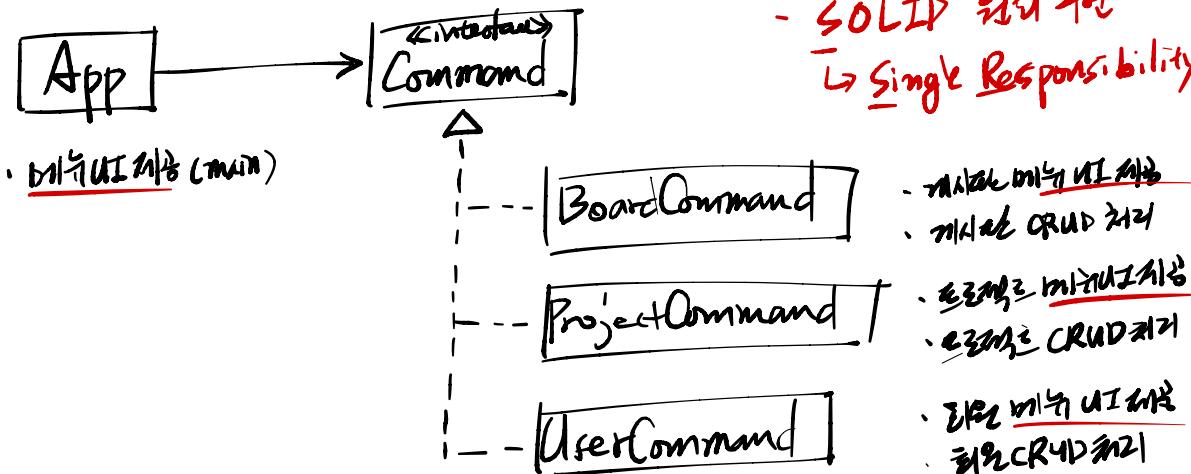
26. GoF의 Composite 패턴 대안

* before

↳ 개체가 트리 구조로 포함 관계임.

문제

- 메뉴나 UI 툴은 같은 정체
 - Command 패턴과 여러 모의 일을 처리
- ↳
- 메뉴나 UI 툴은 같은 정체 → 개체를 분리
 - SOLID 원칙 구현
- ↳ Single Responsibility Principle

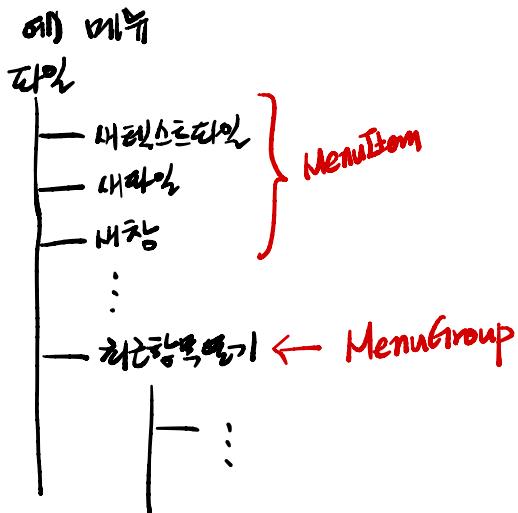
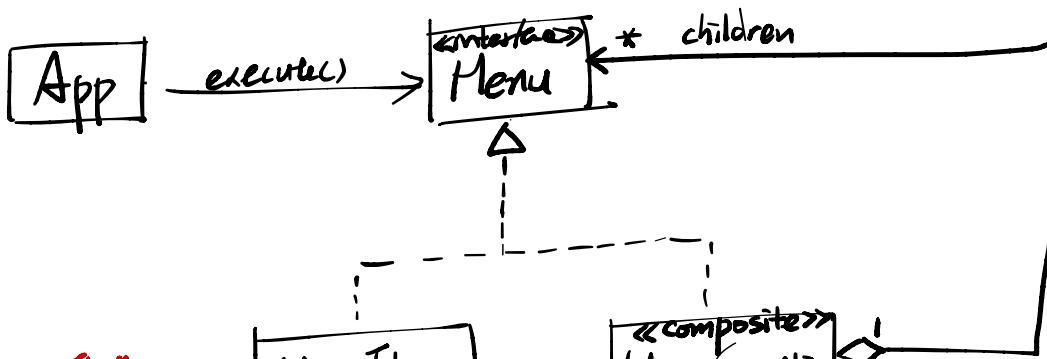


26. GOF의 Composite 패턴 대안

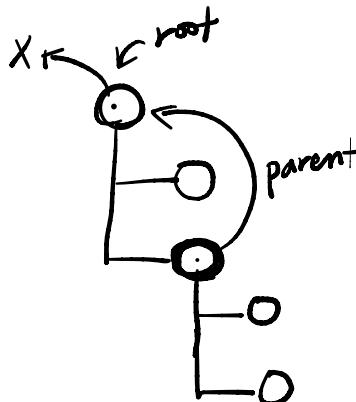
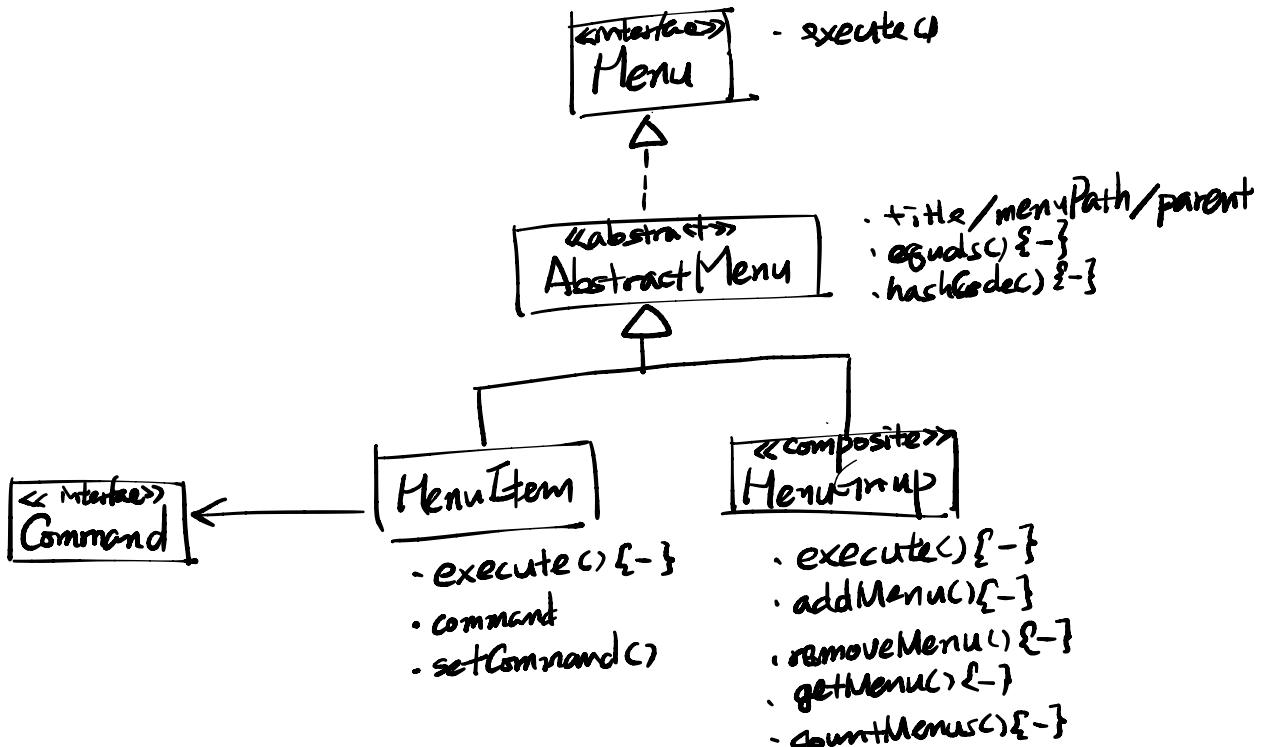
* after

↳ 개체가 트리 구조로 포함 관계임.

- { ① 메뉴
- ② 그룹
- ③ 파일 시스템

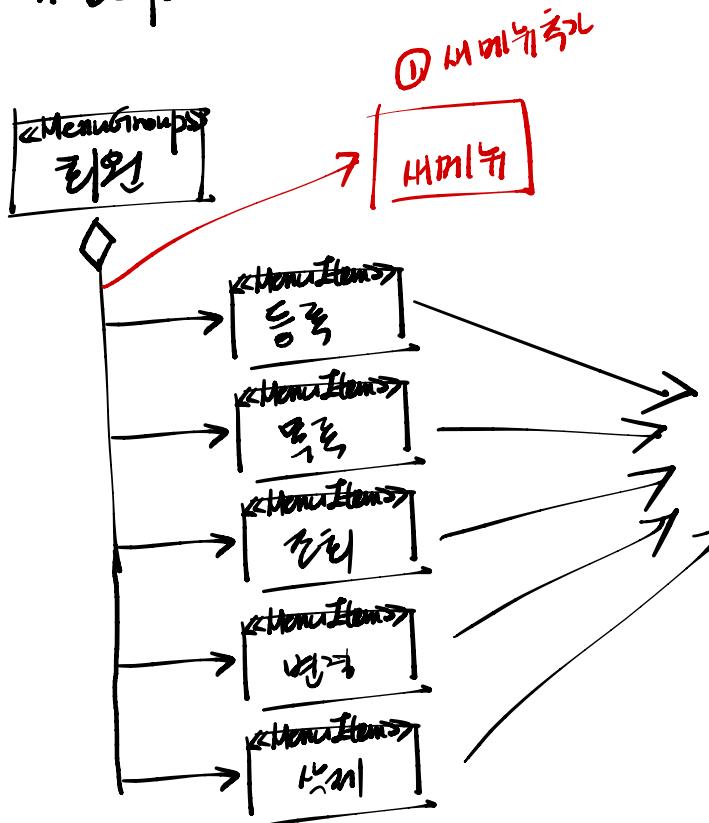


26. GOF의 Composite 패턴



27. 메뉴의 메뉴추가 기능을 개선하자 : GoTo Command 패턴처럼

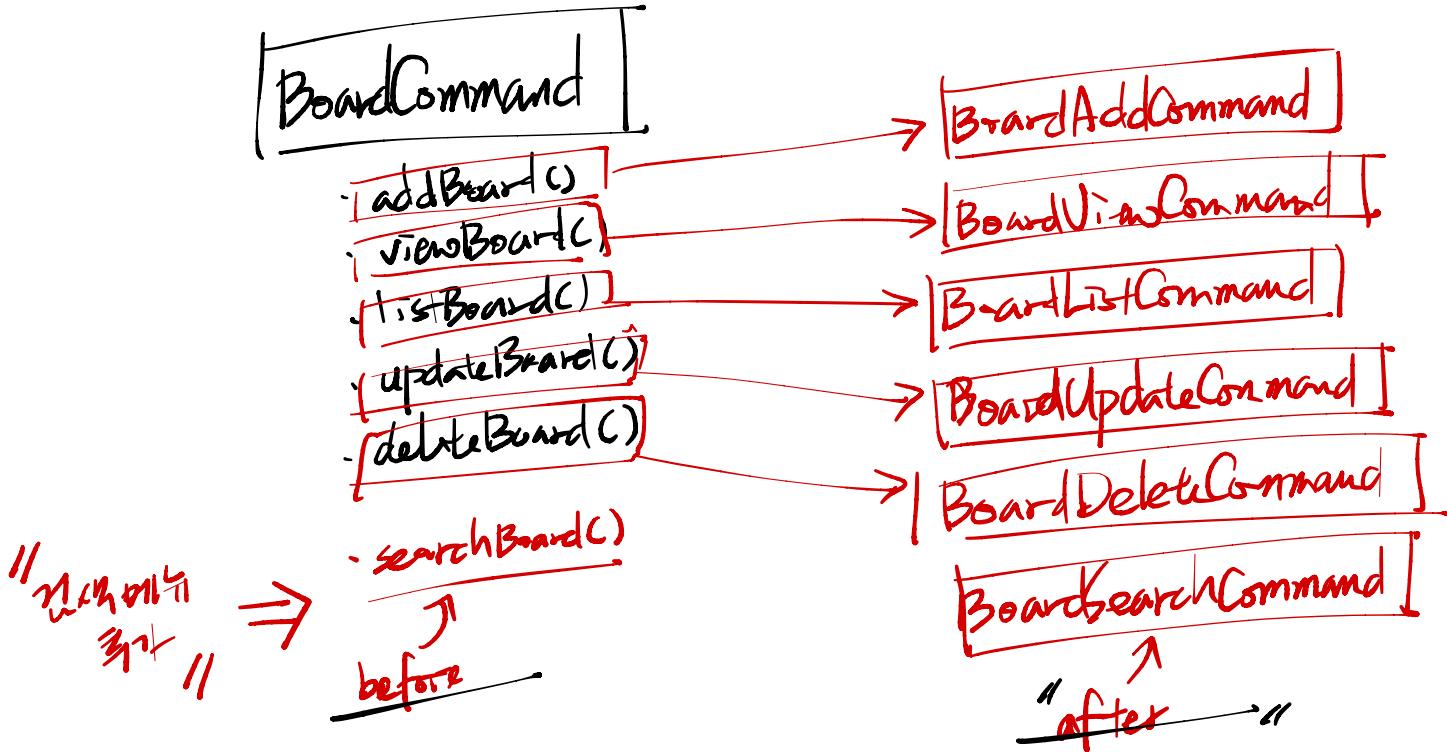
* before



② 메뉴를 처리할
코드를 추가
↓
SOLID의 OCP 원칙을
기반한
기능추가/기능제거
가능성이 있는 구조.

↑
기능추가/기능제거
가능성이 있는 구조.
↓
기능제거
기능추가

27. 게시판의 목록관리 기능을 구현하자 : GoTo Command from list
↳ list → list



27. 메뉴의 명령처리 기능을 구현하자 : GoTo Command 패턴 틀

* before

