설계 패턴

GoF Pattern Catalog

- Creational Patterns
  - Abstract Factory
  - Prototype
  - Singleton
  - Factory Method
  - Builder
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Flyweight
  - Façade
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Creational Patterns?

- UML Diagram
  - Structure Diagram
    - class diagram, object diagram, component diagram
    - …
  - Behavioral Diagram
    - activity diagram, state diagram, use case diagram
    - …

- Creational Diagram은 없다
  - Creational Patterns?

객체지향 설계 패턴은 만병 통치약?

- 설계 패턴을 많이 적용할 수록 좋은 구조인가?
- 설계 패턴의 단점은 없는가?
- 객체지향의 단점은 없는가?
- 설계 패턴은 암기 과목인가?
  - 많은 패턴을 이해하고 외워서 개발에 적용한다

객체지향 기본원칙 (SOLID)

단일 책임 원칙

- Single Responsibility Principle

- 클래스는 하나의 책임만 가져야 한다.

- 책임 == 수정 이유
  - 클래스를 수정하는 이유는 한 개 뿐이어야 한다.
  - 만약 클래스를 수정하는 이유가 여러 개라면,
    클래스를 쪼개야 한다

개방/폐쇄 원칙

- Open/Closed principle

- 클래스는 기능 확장에 대해 열려 있어야 하고,
  소스코드 수정에 대해서는 닫혀 있어야 한다.

- 소스코드 수정 없이 기능을 확장할 수 있어야 한다.
  - 상속하여 메소드 재정의
  - 호환되는 객체로 치환

리스코프 치환 원칙

- Liskov substitution principle

- 부모 타입의 참조 변수를 사용하여 구현된 코드에서,
  그 참조 변수에 자식 타입 객체를 대입해도,
  아무 문제 없이 잘 작동해야 한다.

인터페이스 분리 원칙

- Interface Segregation Principle

- 클라이언트는 자신이 사용하지 않는 인터페이스에 의존하지 않아야 한다.

- 클라이언트가 의존하는 인터페이스에는,
  그 클라이언트가 호출하는 메소드들만 포함되어야 한다.

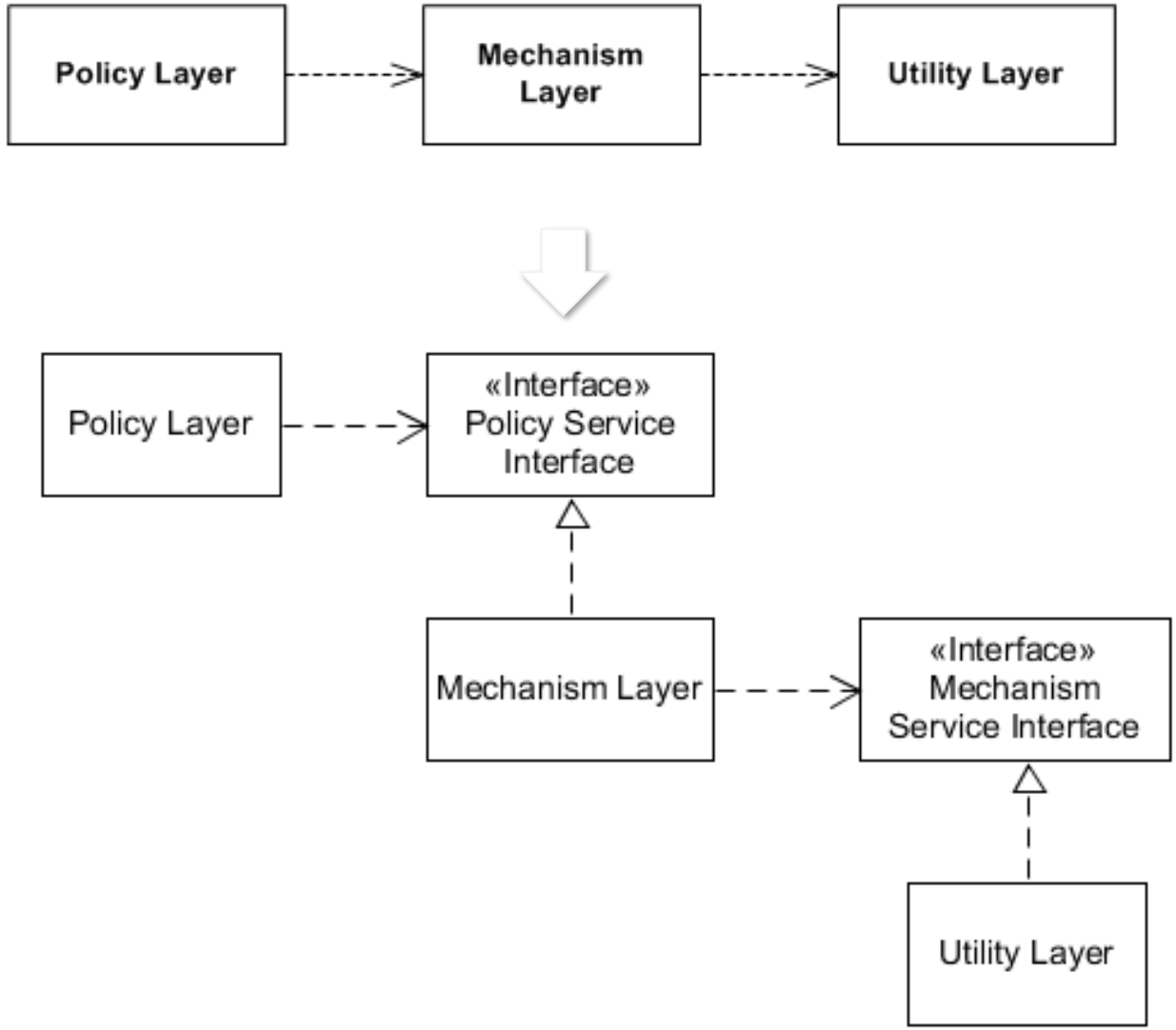
의존성 역전 원칙

- Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).

- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

의존성 역전 원칙


Policy Layer
Mechanism Layer
Utility Layer
Policy Layer
«Interface» Policy Service Interface
Mechanism Layer
«Interface» Mechanism Service Interface
Utility Layer

Iterator

예제 분석

- 객체 인터페이스는 어떤 관점에서 설계되어야 하는가?
  - 내부 구현
  - 사용 목적

- 예제 소스코드 분석
  - iterator.e1, iterator.e2 구현을 비교하시오
  - 장단점은?
  - 개선할 부분은?

부품의 표준화

- 내부 구현이 다르더라도, 사용 목적이 같다면 사용법도 같아야 한다
- HDD와 SSD의 규격
- 엔진 자동차와 전기 자동차의 운전법
- 부품 표준화의 장점과 단점은?

예제 분석

- 예제 소스코드 분석
  - iterator.e2, iterator.e3 구현을 비교하시오

- iterator.e2 구현의 심각한 단점을 해결 → iterator.e3

메소드 이동

- iterator.e4
  - isEnd, getNext 메소드를 Position 클래스로 이동

- 예제 소스코드 분석
  - iterator.e3, iterator.e4 구현을 비교하시오
  - 개선할 부분은?

다형성 구현

- iterator.e5
  - MyArray.Position, MyList.Position 클래스에 다형성 구현
  - 다형성 구현의 효과는?

- 예제 소스코드 분석
  - iterator.e4, iterator.e5 구현을 비교하시오

다형성 구현

- iterator.e6
  - MyArray, MyList 클래스에 다형성 구현
  - 다형성 구현의 효과는?

- 예제 소스코드 분석
  - iterator.e5, iterator.e6 구현을 비교하시오

객체 분리

- 자료구조 탐색 기능을 자료구조 클래스에 구현하는 것이 좋을까?
  - 아니면 별도의 클래스로 분리?

- 단일 책임 원칙

- 역할과 책임 관점에서 객체 분리
  - 자료구조 탐색은 자료구조 클래스의 역할인가?

- 유지보수 단위로 객체 분리
  - 자료구조의 유지보수 & 자료구조 탐색의 유지보수

역방향 탐색

- iterator.e6 예제에 역방향 탐색을 구현하시오

- 역방향 탐색을 구현한 후,
  더 이상 탐색 관련 유지보수 이슈가 없다면,
  어떻게 구현하는 것이 좋은가?

- iterator 인터페이스를 수정하지 않고
  새 클래스만 추가해서 역방향 탐색을 구현할 수는 없을까?

구현 실습 – 유지보수 이슈가 있는 경우

- iterator.e7 구현을 개선하시오

- 역방향 탐색을 구현한 후에도
  탐색 관련 유지보수 이슈가 있을 것으로 예상된다면,
  어떻게 구현하는 것이 좋은가?

- 예상되는 유지보수 이슈
  - 주어진 조건식을 만족하는 항목만 순방향 탐색 기능 추가
  - 주어진 조건식을 만족하는 항목만 역방향 탐색 기능 추가

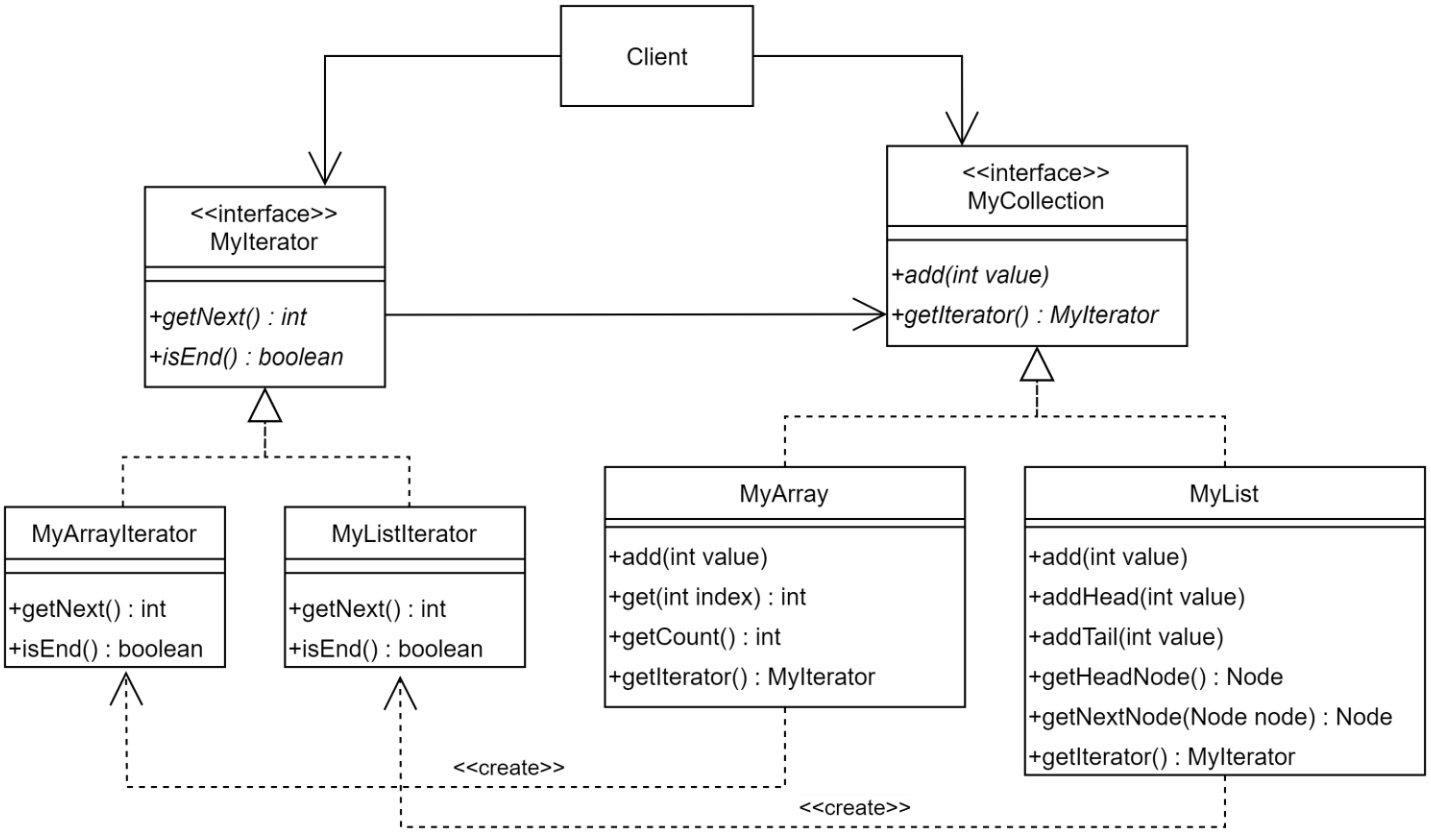
- 개방/폐쇄 원칙 적용해야 함
  - 새 탐색 기능이 추가될 때 마다 수정해야 할 코드는,
    별도의 클래스(소스코드 파일)로 분리해야 한다.

Iterator 패턴 요약

- 자료구조 탐색 기능을 별도의 객체로 분리해서 구현한다
  - 자료구조 객체
  - 자료구조 탐색 객체

- 자료구조 탐색 객체의 사용법을 표준화 한다
  - Iterator 인터페이스

Iterator – 구조


Client
<<interface>>
MyIterator
+getNext() : int
+isEnd() : boolean
<<interface>>
MyCollection
+add(int value)
+getIterator() : MyIterator
MyArrayIterator
+getNext() : int
+isEnd() : boolean
MyListIterator
+getNext() : int
+isEnd() : boolean
MyArray
+add(int value)
+get(int index) : int
+getCount() : int
+getIterator() : MyIterator
MyList
+add(int value)
+addHead(int value)
+addTail(int value)
+getHeadNode() : Node
+getNextNode(Node node) : Node
+getIterator() : MyIterator
<<create>>
<<create>>

Iterator – 결과

- 자료구조 종류에 무관한 일관된 탐색 구현

- 재사용성
  - 특정 자료구조와 무관한 클라이언트 로직 구현 가능

- 유지보수성
  - 새 탐색을 구현할 때, 기존 코드 수정 없이 탐색 클래스만 추가 구현하면 됨

설계 실습: remove 메소드 어디에 구현?

- iterator 현재 위치의 항목을 제거하기 위한 remove 메소드를 어디에 구현하는 것이 좋을까?

- iterator.remove()
  - iterator의 메소드로 구현한다

- collection.remove(iterator)
  - collection의 메소드로 구현한다

- 장단점은?

Iterator – 구현 이슈

- iterator 로 자료구조를 탐색하고 있는 도중에
  자료구조에 새 객체를 추가하거나 삭제하는 것을 허용?

- 탐색 도중의 수정을 허용할 것인가?
  - 허용한다면 구현은 어떻게?
  - 자료구조가 수정되면 이를 iterator 들에게 어떻게 알려주나?

- 탐색 도중의 수정을 금지할 것인가?
  - 설계 지침, 코딩 지침으로만 금지할 것인가?
  - 수정을 못하도록 막는 코드를 구현할 것인가?
  - 수정 금지를 구현한다면 어떻게 구현할 것인가?

여러 종류의 자료구조와 Iterator

- 순방향 탐색은 모든 자료구조에서 구현 가능
  - getFirst(), getNext()

- 순방향 탐색, 역방형 탐색, 랜덤 액세스 구현이 불가능한 자료구조도 있다
  - getFirst(), getNext()
  - getLast(), getPrev()
  - getCurrentPosition(), gotoPosition()

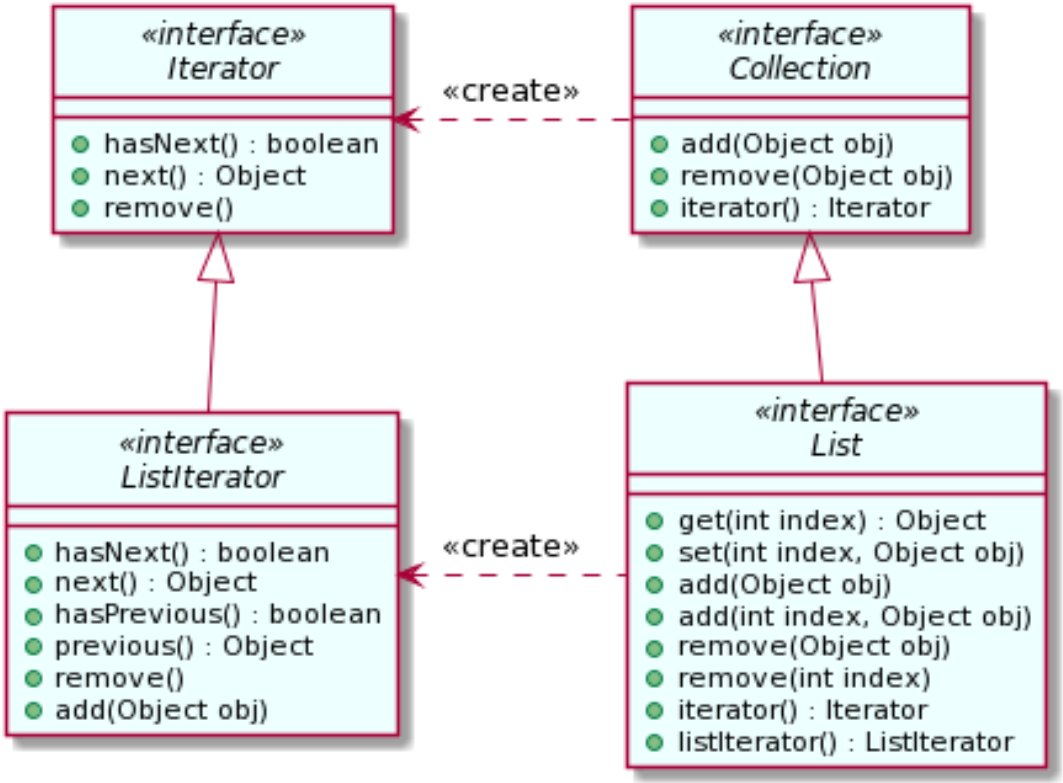
여러 종류의 자료구조와 Iterator

- iterator 표준을 어떻게 정의할 것인가?

- 하향 평준화 인테페이스 정의
  - 모든 자료구조가 지원 가능한 메소드만 iterator 표준에 포함한다
  - → 구현하기 쉽고 단순하지만 기능의 제약

- 상향 평준화 인터페이스 정의
  - iterator 표준에 모든 메소드를 다 정의하고, 각 iterator를 구현할 때 가능한 것만 구현한다
  - → 구현이 복잡해진다

Iterator 상속


«interface»
Iterator
hasNext() : boolean
next() : Object
remove()
«create»
«interface»
Collection
add(Object obj)
remove(Object obj)
iterator() : Iterator
«interface»
ListIterator
hasNext() : boolean
next() : Object
hasPrevious() : boolean
previous() : Object
remove()
add(Object obj)
«create»
«interface»
List
get(int index) : Object
set(int index, Object obj)
add(Object obj)
add(int index, Object obj)
remove(Object obj)
remove(int index)
iterator() : Iterator
listIterator() : ListIterator

Polymorphism

add all 구현

- iterator.e1 예제에 addAll 메소드 구현 -> polymorphism.e1
- iterator.e4 예제에 addAll 메소드 구현 -> polymorphism.e2
- iterator.e6 예제에 addAll 메소드 구현 -> polymorphism.e3

- java8의 default method 문법을 활용할 수 없다면
  -> polymorphism.e4

- 예제 소스코드 분석
  - polymorphism.e1, e2, e3, e4 예제를 비교하시오

Prototype

Prototype – 문제

- 임의의 객체의 복제를 만든다

- 예: 파워포인트에서
  - 임의의 도형을 선택하고 Ctrl-D 를 누르면
  - 선택된 도형의 복제가 만들어 진다

- 예: 클립보드
  - Ctrl-C 를 누르면, 선택된 도형의 복제가 클립보드에 저장된다.
  - Ctrl-V 를 누르면 클립보드에 복사된 도형이 또 복제되어 문서에 삽입된다
  - Ctrl-V 를 누를 때마다 계속 복제된다.

Prototype – 문제

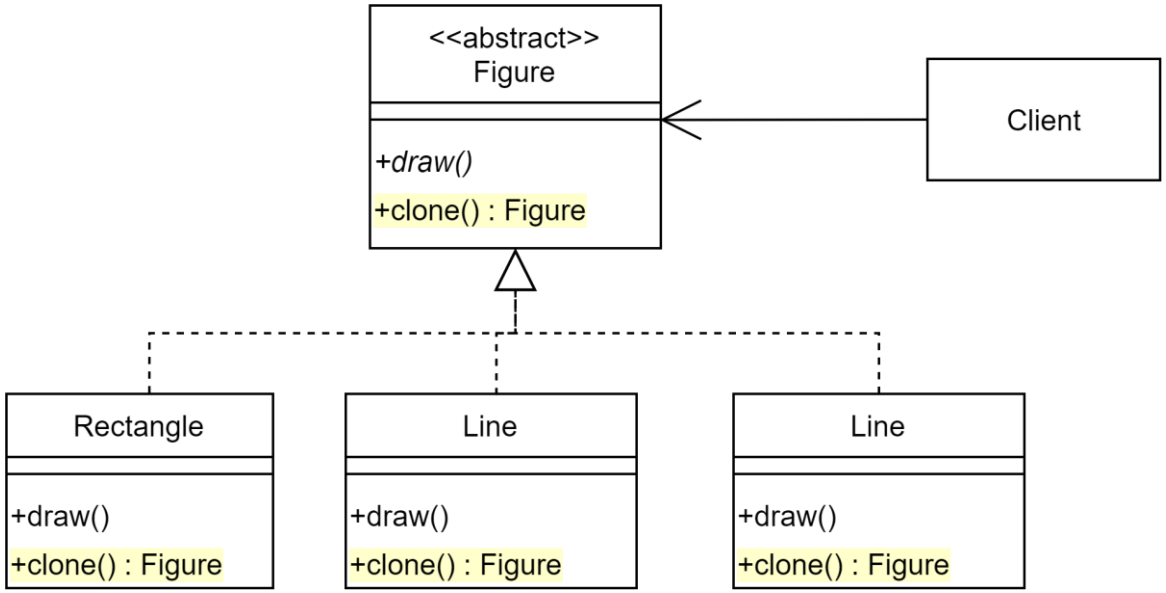
- 선택된 임의의 객체와 동일한 객체를 만들때,
  클래스 이름을 언급하지 않고 객체를 생성할 수 있는가

- 객체 생성 다형성

Prototype – 해결

- 자신의 복제를 만드는 기능을 각 객체에 구현한다.
- 이 메소드에 다형성을 구현한다
- Java, C# 언어의 virtual machine이 객체 복제를 해준다 (shallow copy)

Prototype – 구조



Prototype – 결과

- 주어진 임의의 도형 객체의 복제본을 만들 때,
  그 객체의 clone() 메소드를 호출하면 된다.

- 구체적인 클래스를 참조하지 않고 임의의 객체 복제 가능

Java 에서 clone() 메소드 구현

- 부모 클래스가 Cloneable 인터페이스를 구현하면,
  모든 하위 클래스에 clone() 메소드 구현이 상속된다.
  (shallow copy)

```
abstract class Figure implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
    . . . .
}
```

C++ 에서 clone() 메소드 구현

- 먼저 copy constructor를 구현한다.
- copy constructor를 이용하여 clone() 메소드를 구현한다.

- shallow copy에 해당하는 copy constructor는 C++ 컴파일러가 자동으로 생성해 줌 (default copy constructor)

- deep copy가 필요할 때만 copy constructor를 구현하면 된다.

```
class Rectangle : public Figure {
    public Figure* clone() {
        return new Rectangle(*this);
    }
    . . . .
}
```

예제 코드 분석

- 도형 클래스들에 prototype 패턴 구현
  - prototype.f1 (Java)
  - prototype.f1 (C++)

- 컬렉션 클래스에 prototype 패턴 구현
  - prototype.e1

- shallow copy 구현의 흔한 실수
  - prototype.e1 구현에 버그가 있다.
  - 이 버그의 원인은?

Composite

범용 컬렉션

- 앞 예제의 MyArray, MyList 는 int 만 저장한다

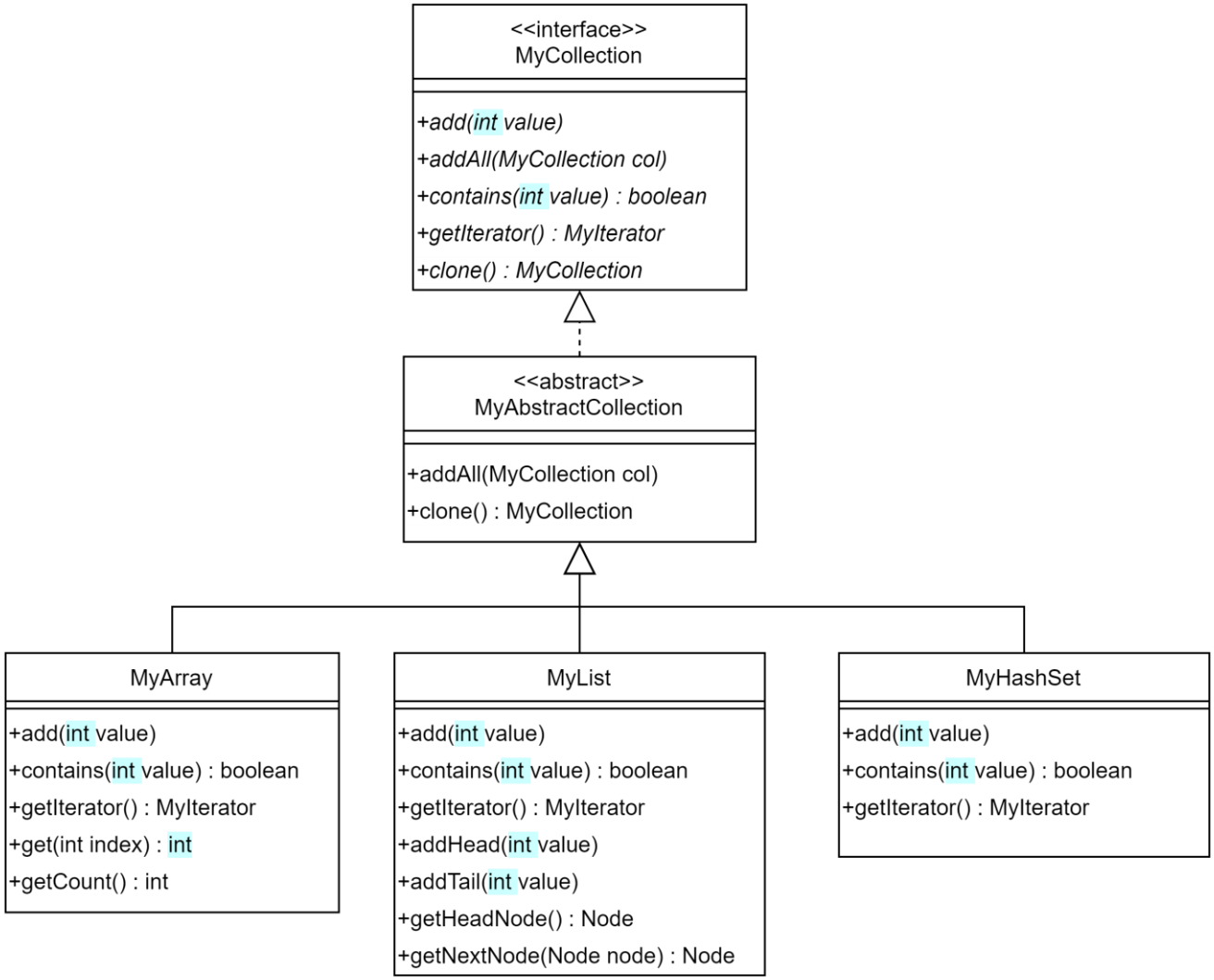
- 아무 데이터나 저장할 수 있으려면
  - 자료구조를 c++ 언어의 template class 로 구현하거나
  - 아니면 객체의 참조(주소)만 저장해야 한다

범용 컬렉션

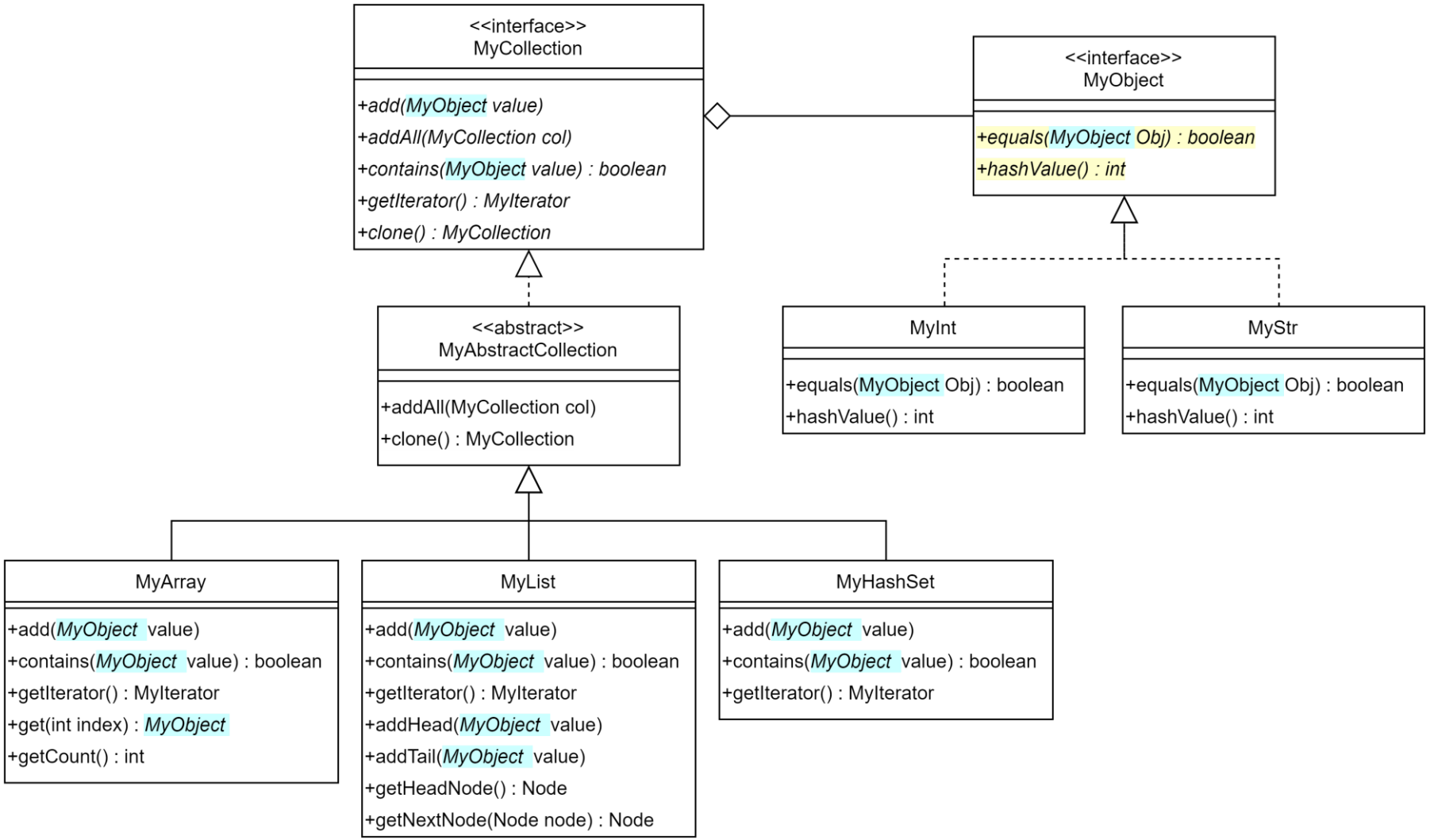
- up casting
  - 참조 변수는 자식 타입 객체에 대한 참조를 저장할 수 있다

- MyObject 클래스의 자식 타입 객체에 대한 참조 저장
  - void add(MyObject value);
  - MyObject[] array = new MyObject[10];

composite.e1 – int 컬렉션



composite.e2 – 범용 컬렉션



예제 분석

- 예제 소스코드 분석
  - composite.e1, composite.e2 구현을 비교하시오

- composite.e2 구현은 아직 composite 패턴이 아니다

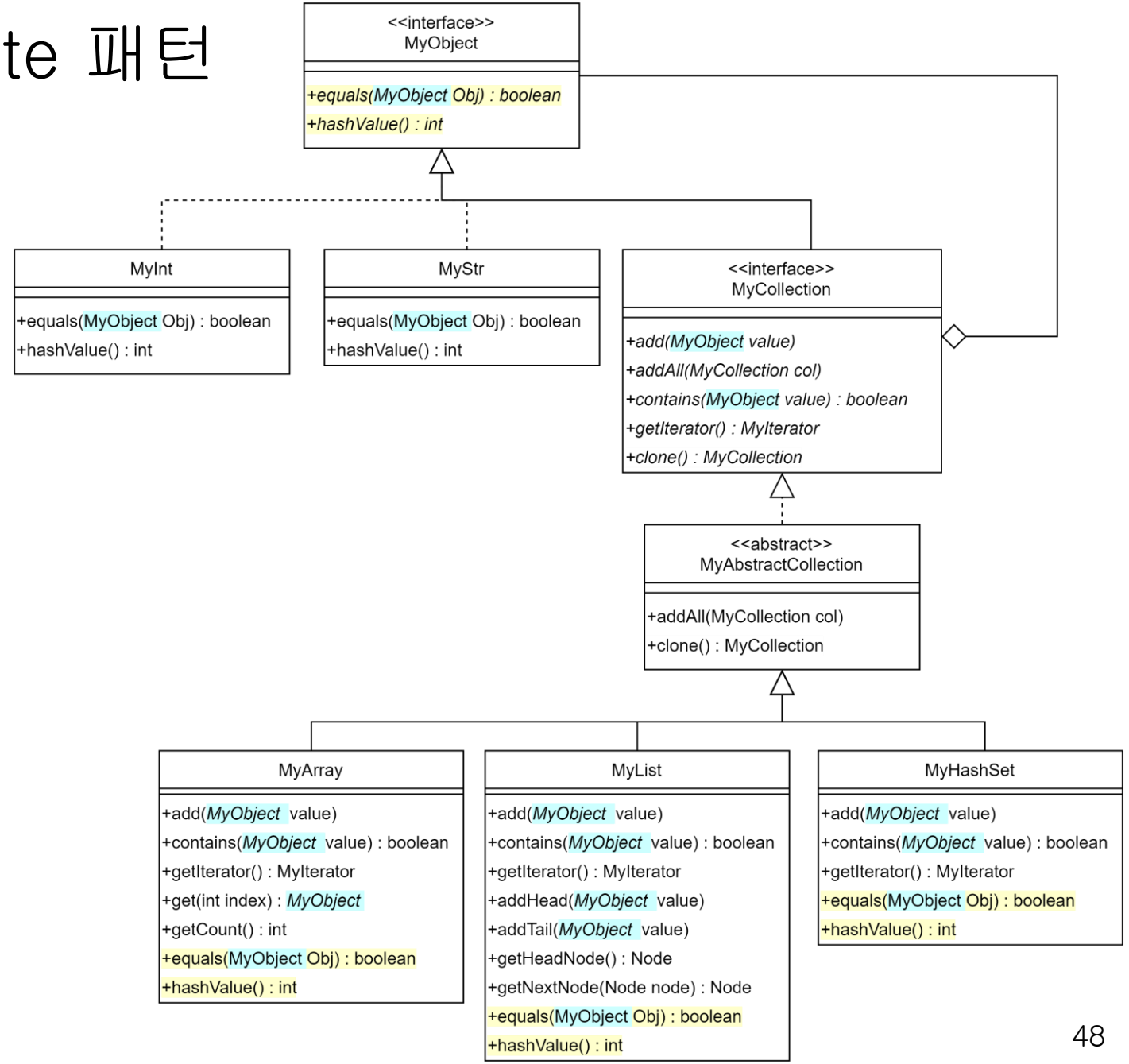
Composite 패턴

- MyArray, MyList 객체도 MyArray, MyList 컬랙션에 저장할 수 있으려면?

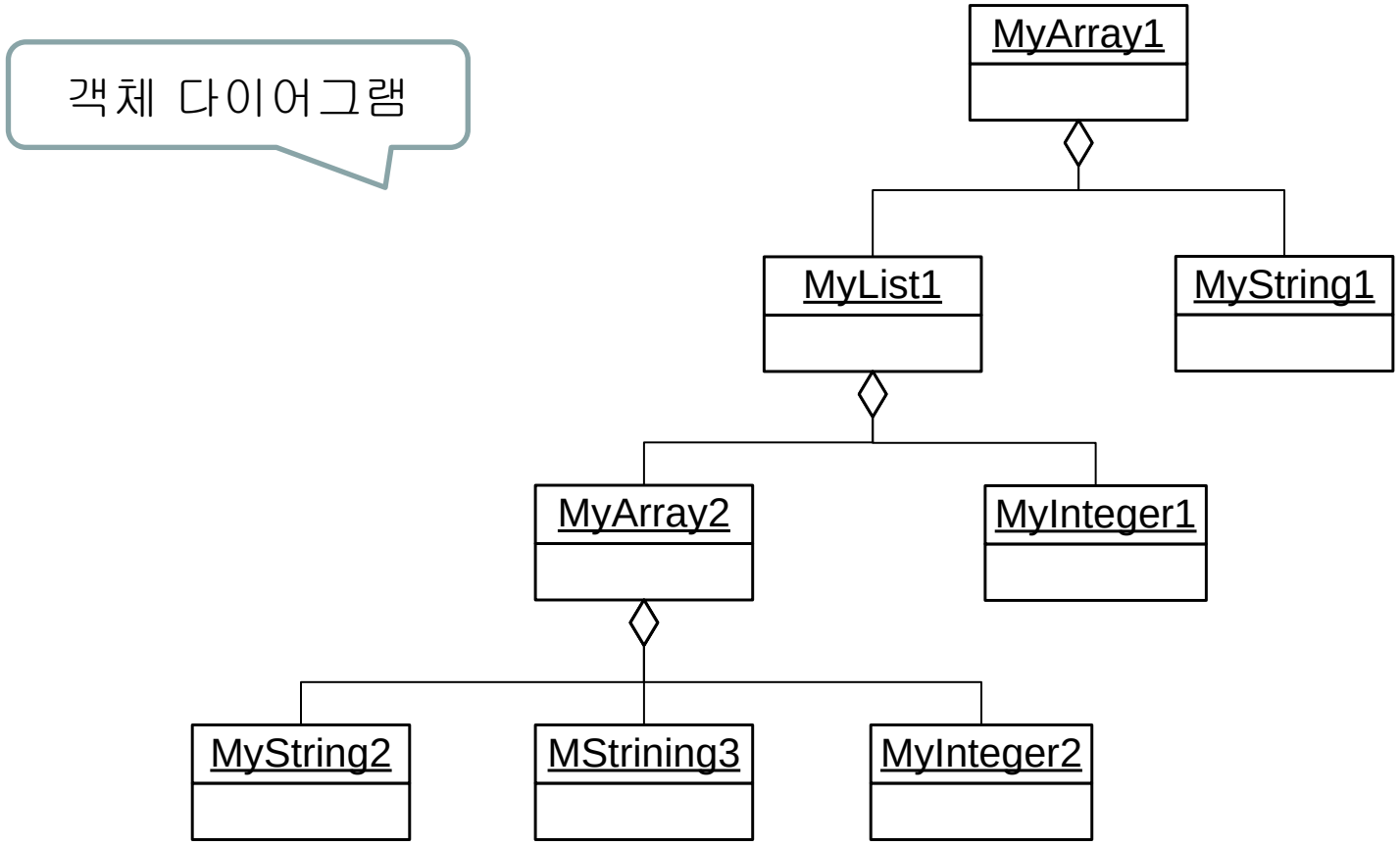
- MyInt, MyStr 객체를 MyArray, MyList 컬렉션에 저장할 수 있는 이유는 MyObject 자식 객체이기 때문이다.

- MyArray, MyList 객체도 MyObject 자식 객체이면,
  MyArray, MyList 컬랙션에 저장할 수 있다.

Composite 패턴



Composite – 객체 다이어그램



예제 분석

- 예제 소스코드 분석
  - composite.e2, composite.e3 구현을 비교하시오

- composite.e3 구현은 composite 패턴

예제 분석

- composite.e3 구현에 유사한 코드 중복이 많다.
  - 코드 중복 제거 → composite.e4
  - composite.e3, composite.e4 구현을 비교하시오

- 유사 코드 중복 제거에 활용된 패턴은?

구현 실습

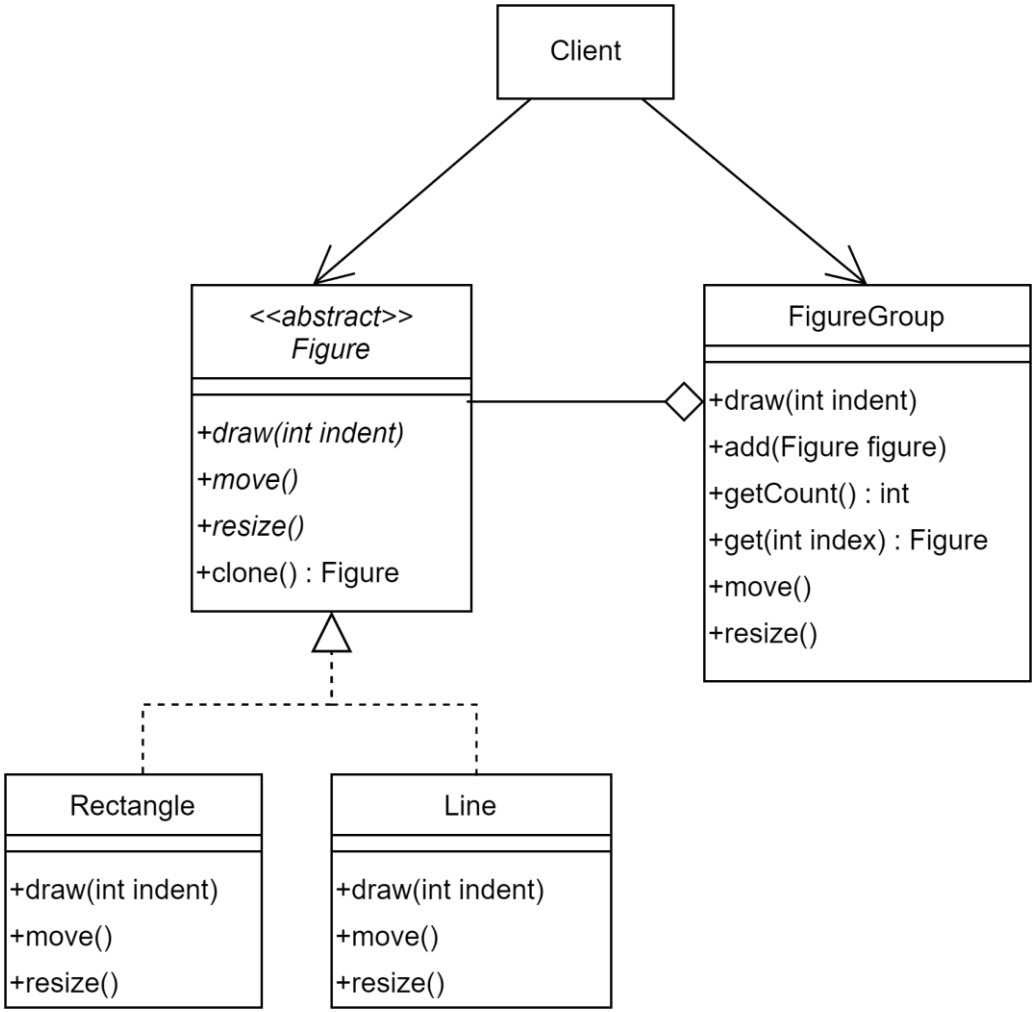
- composite.e4 예제의 Example4 클래스에도
  유사한 코드 중복이 많다.

- 아래 메소드들의 유사 코드 중복을 제거하시오
  - createCompositeArray, createCompositeList,
    createCompositeHashSet
  - testArray, testList, testHashSet

- 힌트
  - prototype 패턴 활용
  - createComposite### 메소드 3개와
    test### 메소드 3개를 각각 한 개로 줄일 수 있다.

Composite – 다른 문제

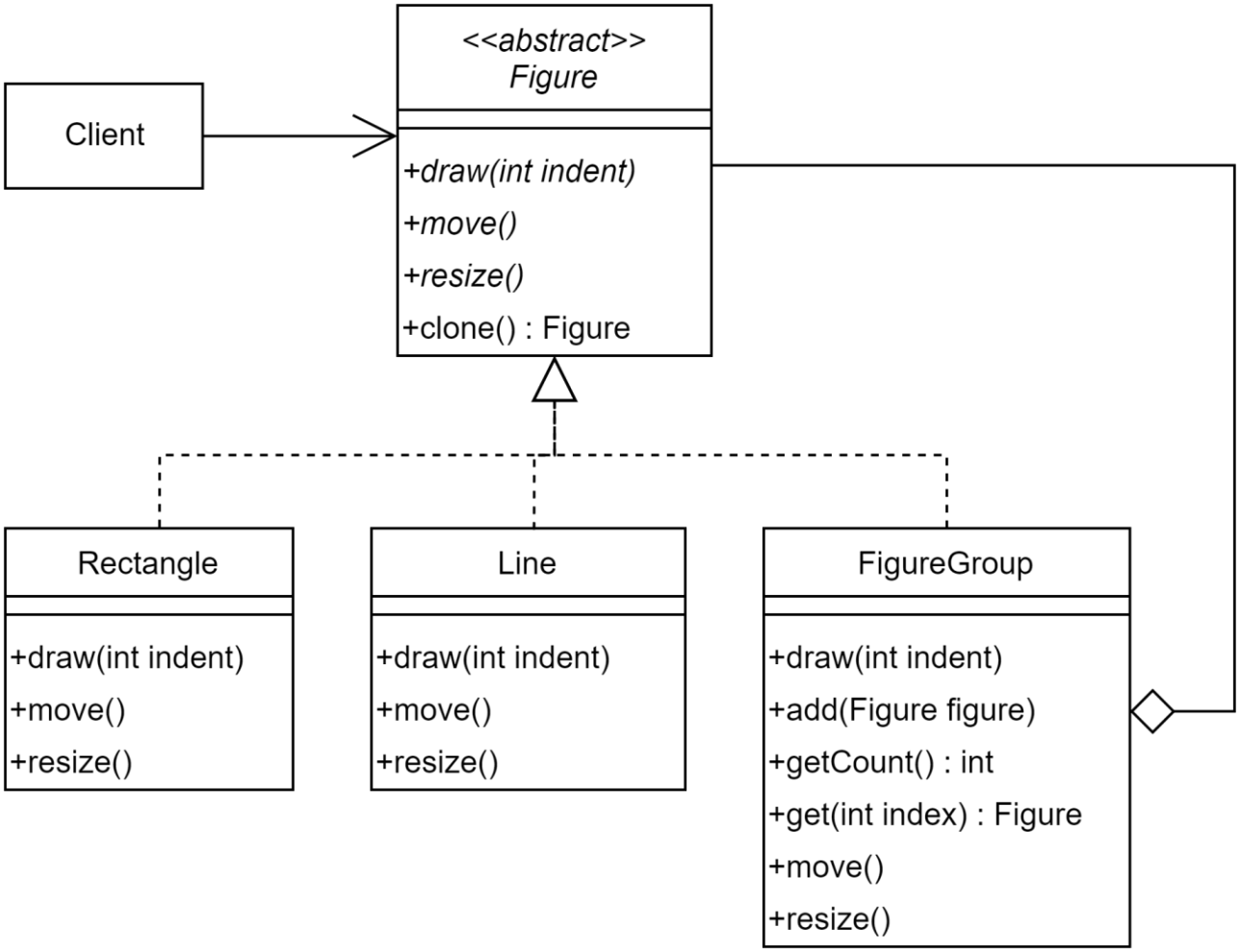
- 그래픽 에디터에서 도형의 그룹 객체를 생성할 수 있다



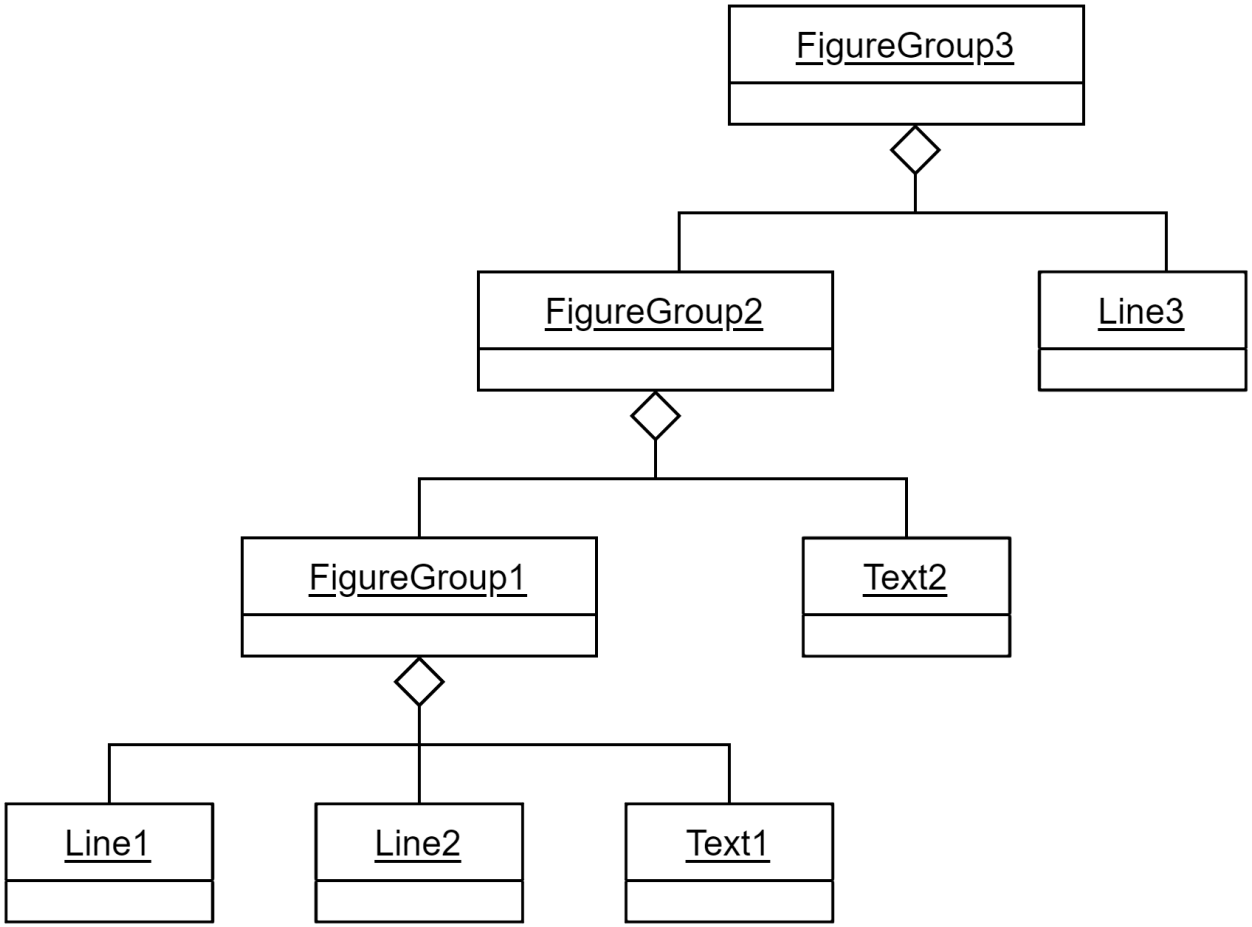
Composite – 다른 문제

- 앞 슬라이드의 구조에 다음 요구 사항을 추가로 구현하면?
- 그룹 객체 내부에 그룹 객체가 포함될 수 있어야 한다
- 도형 그룹 객체도 도형과 똑같이 다룰 수 있어야 한다
  - 이동, 크기 변경, 속성 변경, copy cut paste ...

Composite – 해결



객체 계층 구조


FigureGroup3
FigureGroup2
Line3
FigureGroup1
Text2
Line1
Line2
Text1

예제 분석

- 예제 소스코드 분석
  - prototype.f1, composite.f2 구현을 비교하시오

- composite.f2 구현은 composite 패턴

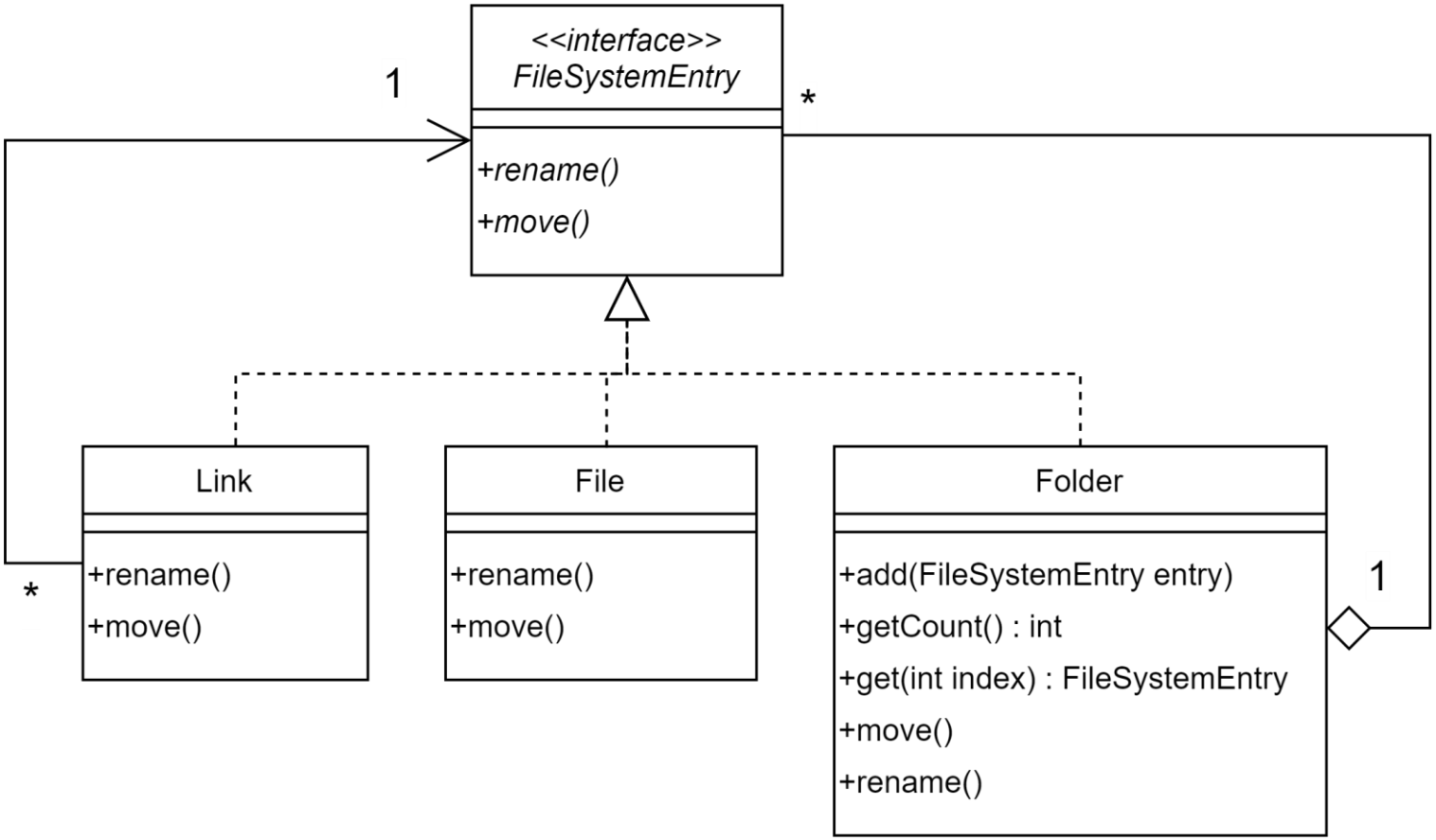
Composite – 결과

- graphic editor 는 group 객체와 primitive 객체를 구분하지 않고 구현될 수 있다
  - graphic editor 코드가 단순해 진다

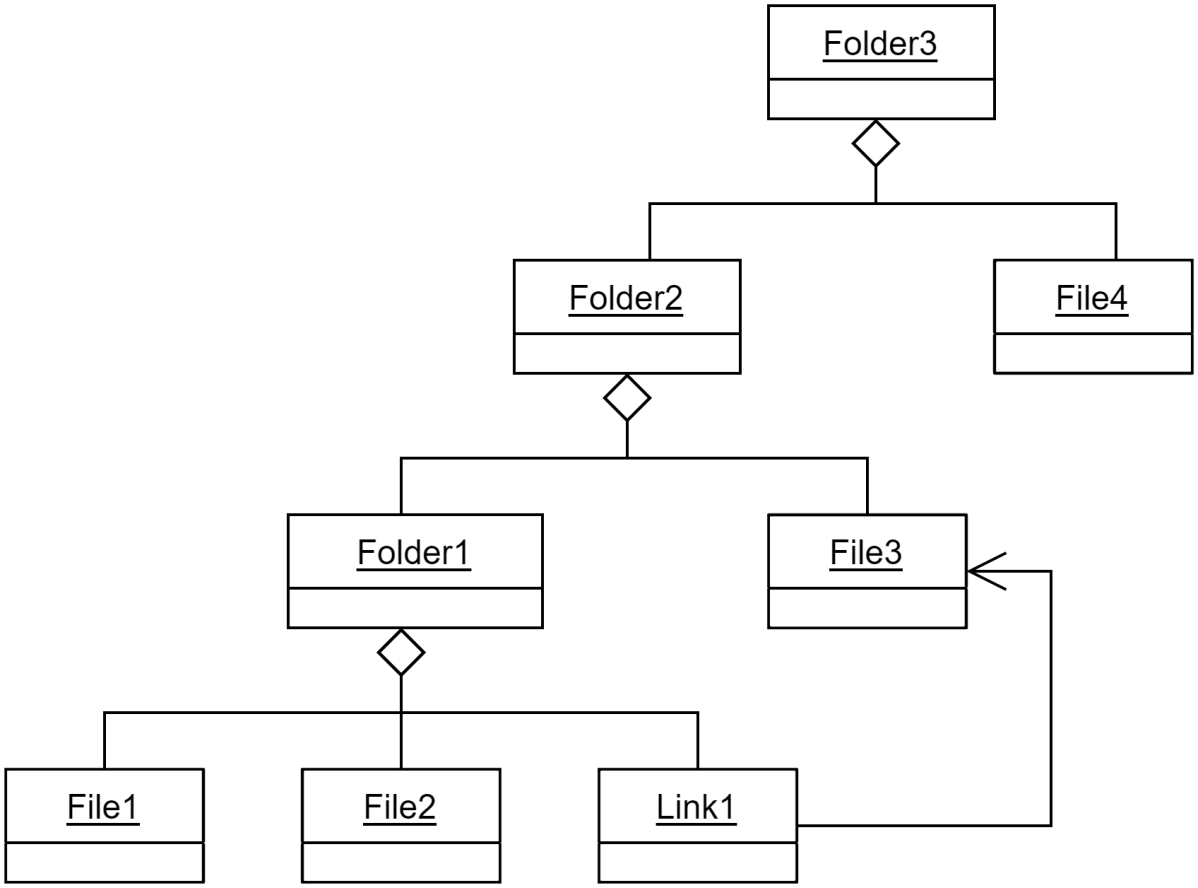
- 여러 단계의 group 이 가능하다

- 새 도형이 추가되어도
  - group 기능을 수정할 필요 없이
  - 새 도형도 group 으로 묶일 수 있다

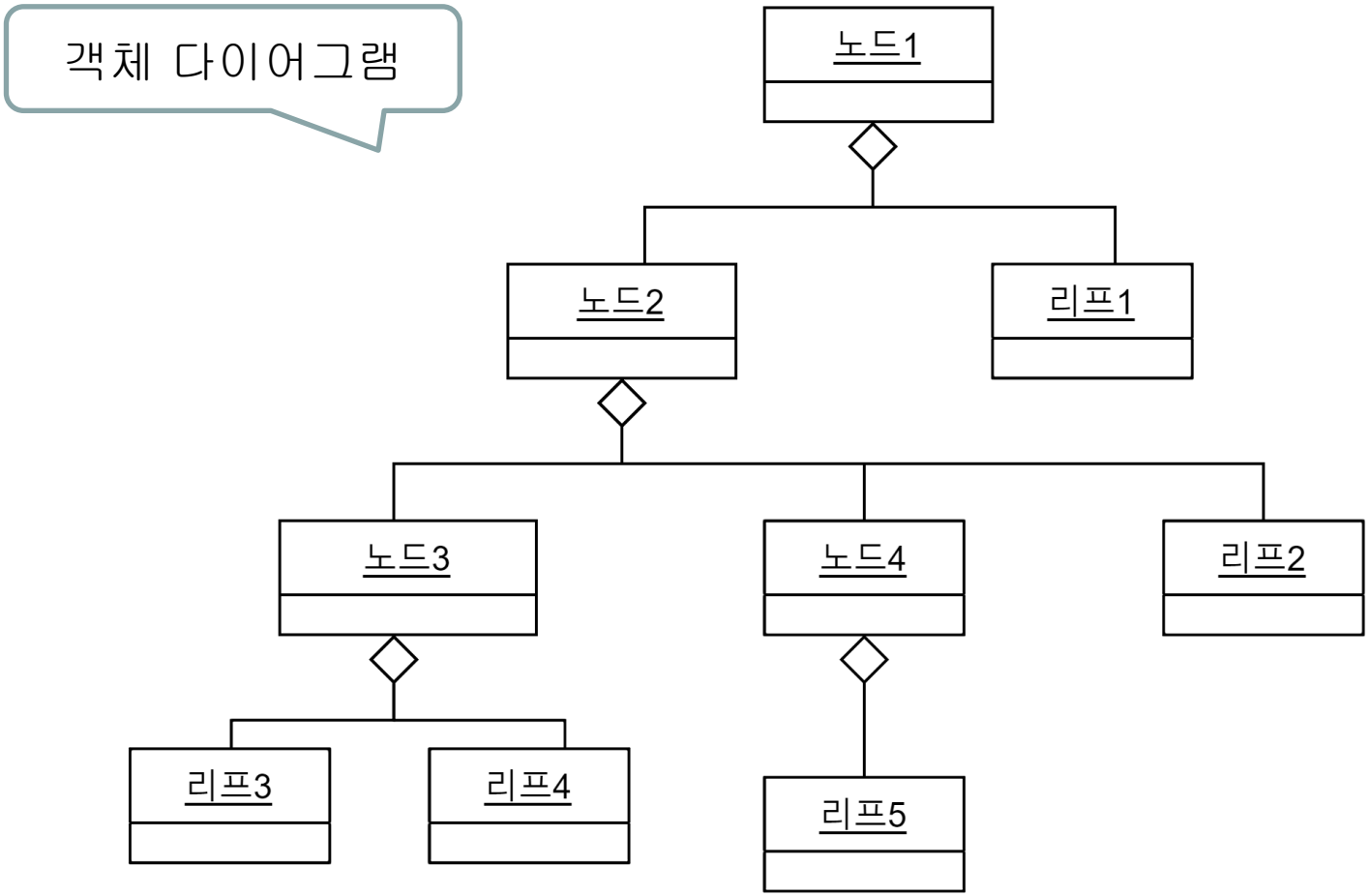
Composite – 예


<<interface>>
FileSystemEntry
+rename()
+move()
1
*
Link
+rename()
+move()
*
File
+rename()
+move()
Folder
+add(FileSystemEntry entry)
+getCount() : int
+get(int index) : FileSystemEntry
+move()
+rename()
1

Composite – 예



Composite 패턴 요약



Composite 패턴 요약

- 앞 슬라이드와 같은 임의의 tree 구조를 구현하려면?
  - node 클래스 수 제한 없음
  - leaf 클래스 수 제한 없음
  - node의 자식 node 수 제한 없음
  - node의 자식 leaf 수 제한 없음

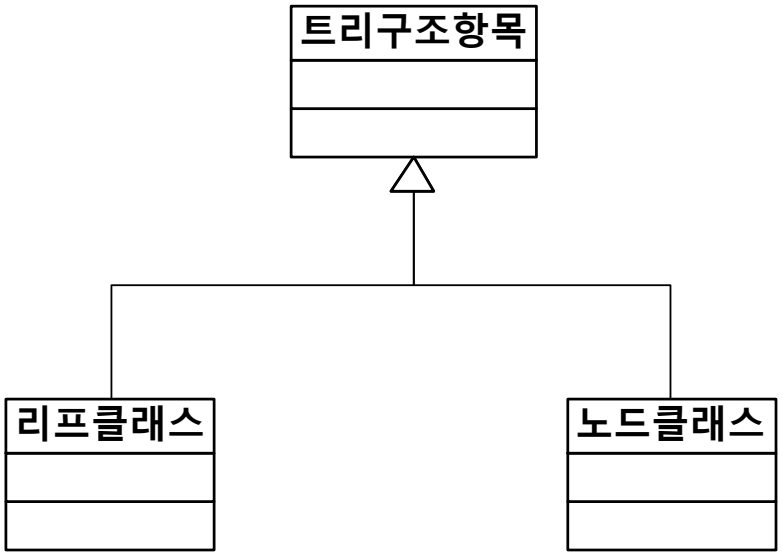
- → composite 패턴

Binary Tree & Linked List

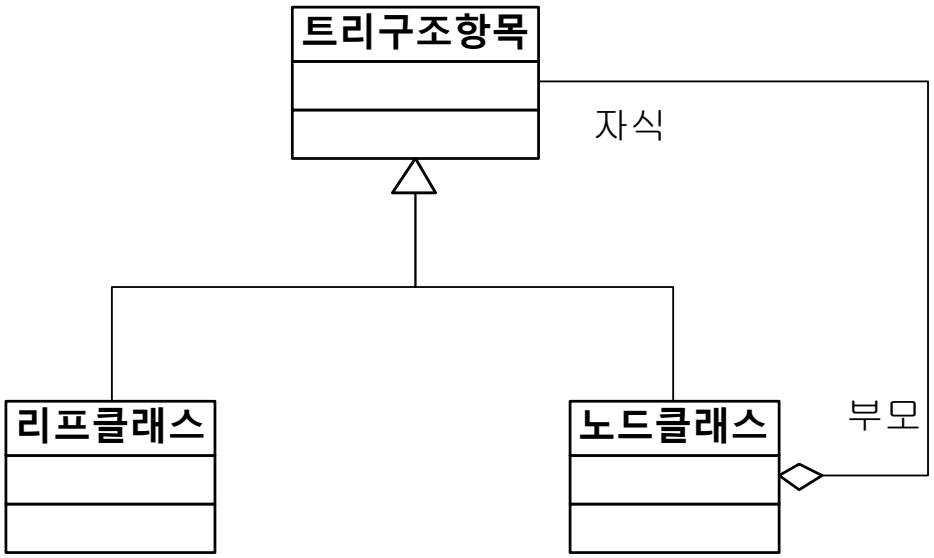
- Binary Tree
  - node 클래스 1개
  - node의 자식 node 수 2개 이하
  - node의 자식 leaf 수 1개

- Linked List
  - node 클래스 1개
  - node의 자식 node 수 1개 이하
  - node의 자식 leaf 수 1개

Composite 패턴 요약

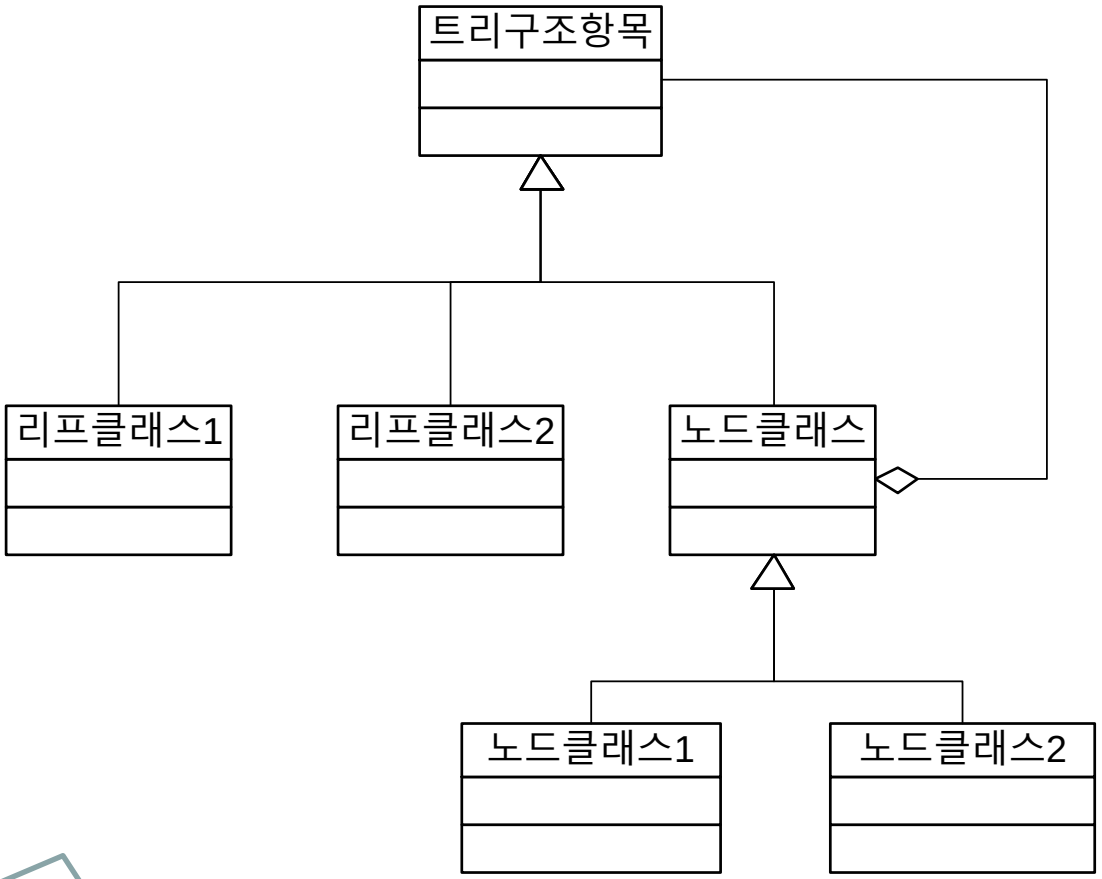


Composite 패턴 요약



Composite 패턴

Composite 패턴 요약



Composite 패턴

Command

Command – 문제

- undo/redo 기능을 구현하려면?
- 명령을 접수하면 일단 대기 큐에 넣고, 하나씩 순서대로 꺼내서 실행해야 한다면? (일시적으로 명령이 너무 많이 전달될 수 있어서 대기 큐가 필요하다)
- 접수된 순서가 아닌, 우선순위 순서로 명령을 실행하려면?
- 매크로 명령 기능을 구현하려면?
- 메뉴나 툴바 버튼이 눌러졌을 때 실행될 명령을 사용자가 변경할 수 있으려면?

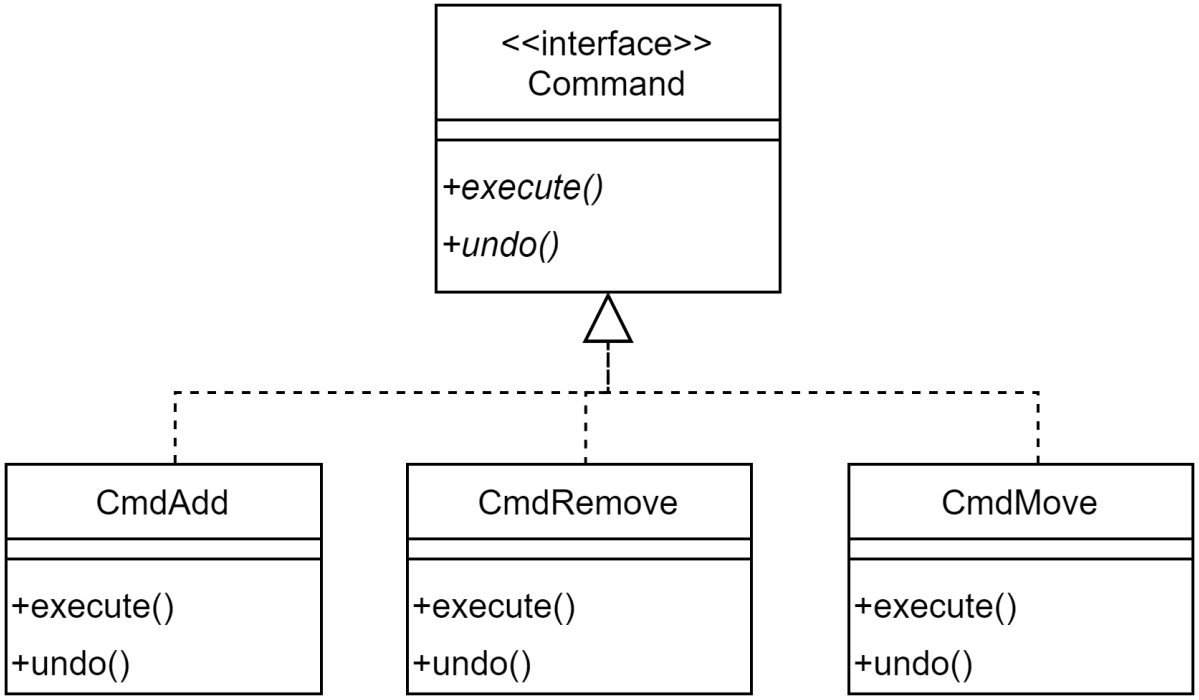
Command – 해결

- 명령을 객체로 구현한다.

- 요청이 발생하자마자 즉시 실행하는 것이 아니고
  요청이 발생하면 명령 객체를 만들어 전달한다

- 명령 객체는 대기 큐나, 우선 순위 큐에 넣어
  순서대로 꺼내어 실행할 수 있다

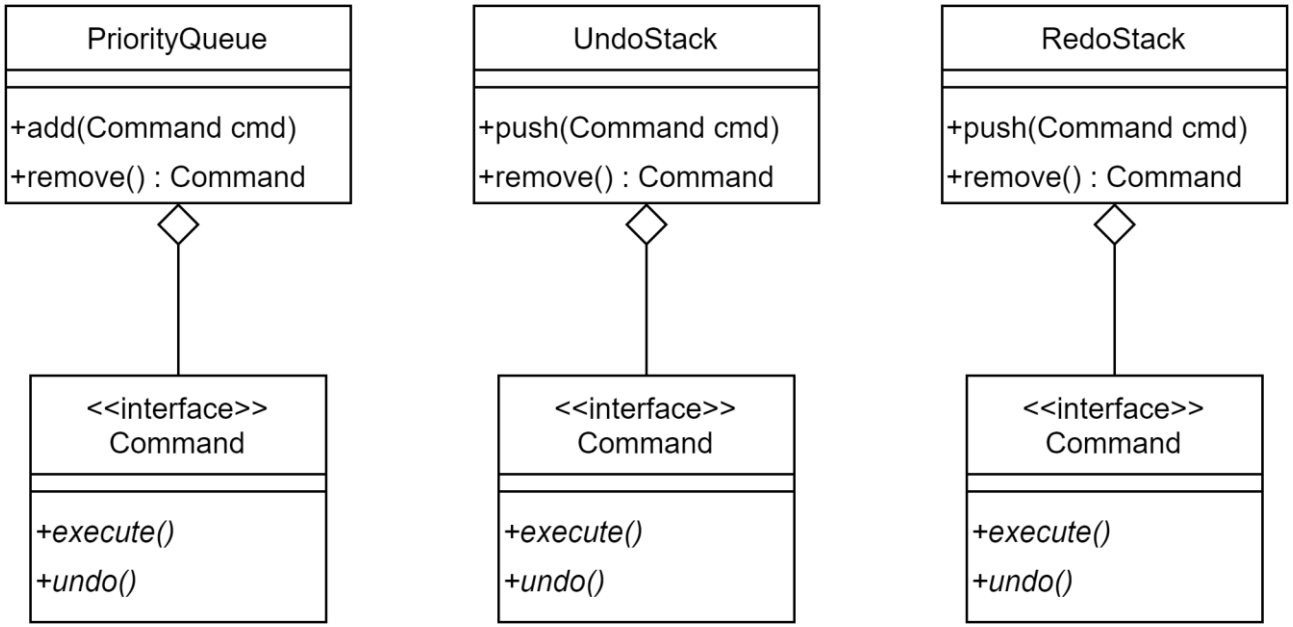
Command – 구조


<<interface>>
Command
+execute()
+undo()
CmdAdd
+execute()
+undo()
CmdRemove
+execute()
+undo()
CmdMove
+execute()
+undo()

Command – 결과

- 표준을 준수하는 일련의 명령 객체들이 만들어진다
- 명령 객체들은 표준을 준수하므로 동일하게 다뤄질 수 있다

Command – 구조

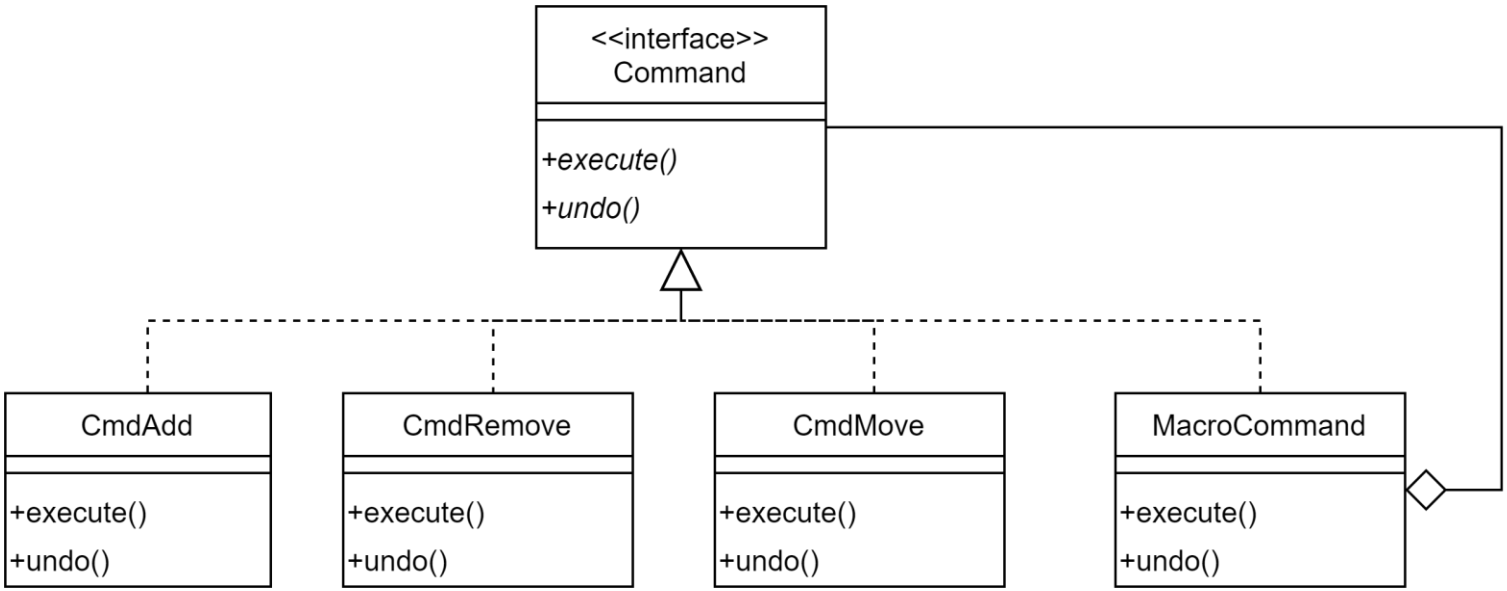

PriorityQueue
+add(Command cmd)
+remove() : Command
<<interface>>
Command
+execute()
+undo()
UndoStack
+push(Command cmd)
+remove() : Command
<<interface>>
Command
+execute()
+undo()
RedoStack
+push(Command cmd)
+remove() : Command
<<interface>>
Command
+execute()
+undo()

Command – 구조

- 명령 객체는 개별적인 상황에서 각자 생성되지만

- 일단 생성되고 나면 명령 객체들은
  priority queue, undo stack, redo stack 에서
  동일하게 다뤄진다

Command – 구조


<<interface>>
Command
+execute()
+undo()
CmdAdd
+execute()
+undo()
CmdRemove
+execute()
+undo()
CmdMove
+execute()
+undo()
MacroCommand
+execute()
+undo()

Command – 구조

- 앞 슬라이드의 MacroCommand 구조는 어떤 패턴인가?

예제 분석

- 예제 소스코드 분석
  - composite.f2, command.e2 구현을 비교하시오

- composite.e2 구현은 command 패턴

Command – undo & redo

- 명령을 실행해야 할 상황이 발생하면
- 명령 객체를 만들어 priority queue 에 넣는다
- priority queue 에서 우선 순위가 높은 명령 객체를 하나 꺼낸다
- 명령을 실행한다
- 실행된 명령 객체는 undo stack 에 넣는다
- undo 가 요청되면 undo stack 에서 명령을 꺼내어 undo 를 실행

- undo 를 실행한 명령 객체는 redo stack 에 넣는다

- redo가 요청되면, redo stack에서 pop해서
  execute 메소드를 호출하고 undo stack에 push

구현 실습

- command.e2 구현에 redo 기능을 구현하시오

Command 패턴 요약

- 각각의 명령을 클래스로 분리하여 구현한다
- 명령 클래스들에 다형성을 구현한다