Proxy

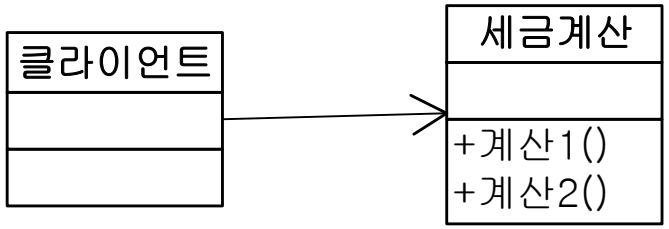
Proxy – 문제

- 객체가 제공하는 기능/서비스는 수정하지 않고,
  그 객체에 대한 접근 방법을 수정해야 한다

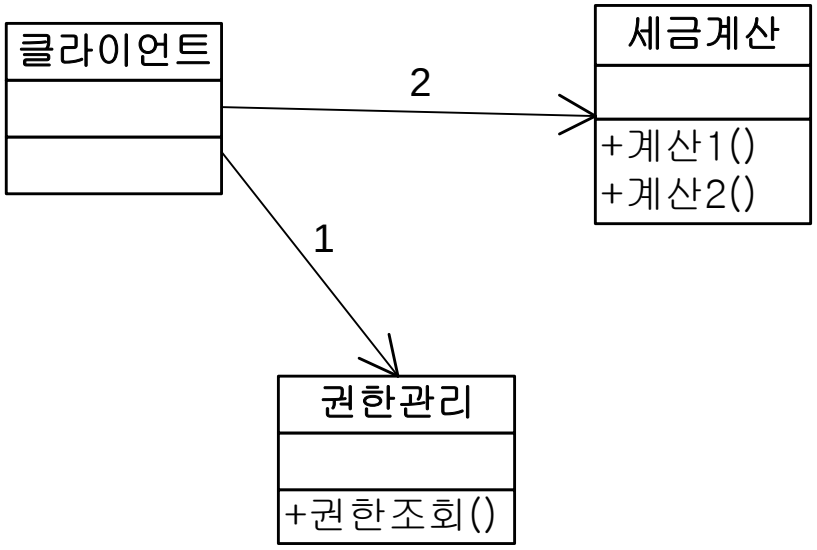
- 예: 세금 계산 클래스
  - 세금 계산식은 수정하지 않는다
  - 사용자 권한 검사 기능을 추가해야 한다

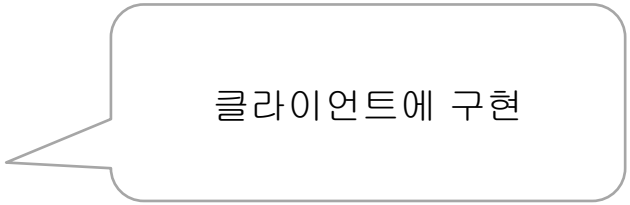
- 권한 검사 기능을 어디에 구현?
  - 세금 계산 서비스 클래스에?
  - 클라이언트에?

권한 정책 구현

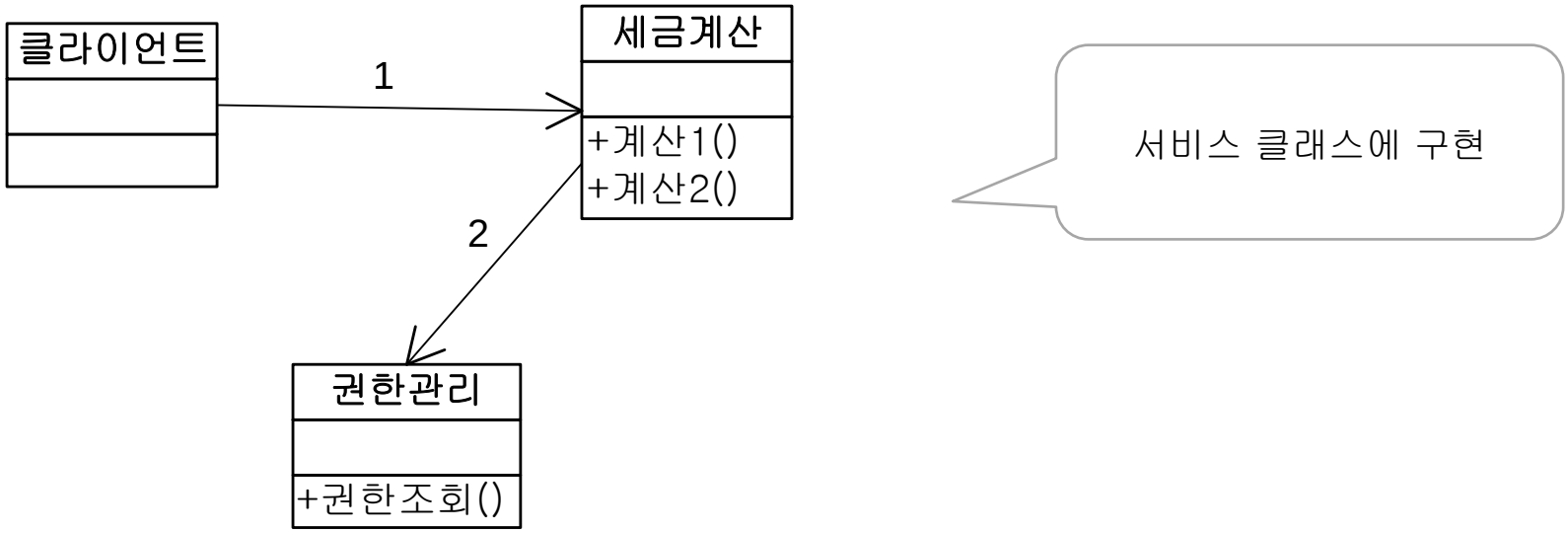

클라이언트
세금계산
+계산1()
+계산2()


before


클라이언트
2
세금계산
+계산1()
+계산2()
1
권한관리
+권한조회()


클라이언트에 구현

권한 정책 구현



클라이언트에 구현

- 단점
  - 코드 중복: 모든 클라이언트들에 구현해야 함
  - 유지보수성 하락: 권한 정책 수정될 때 마다 클라이언트들 유지보수

- 장점
  - 서비스 클래스가 오염되지 않는다
  - 서비스 클래스의 유지보수성, 재사용성이 좋다

서비스 클래스에 구현

- 단점
  - 서비스 클래스 오염됨
  - 서비스 클래스 유지보수성, 재사용성 하락

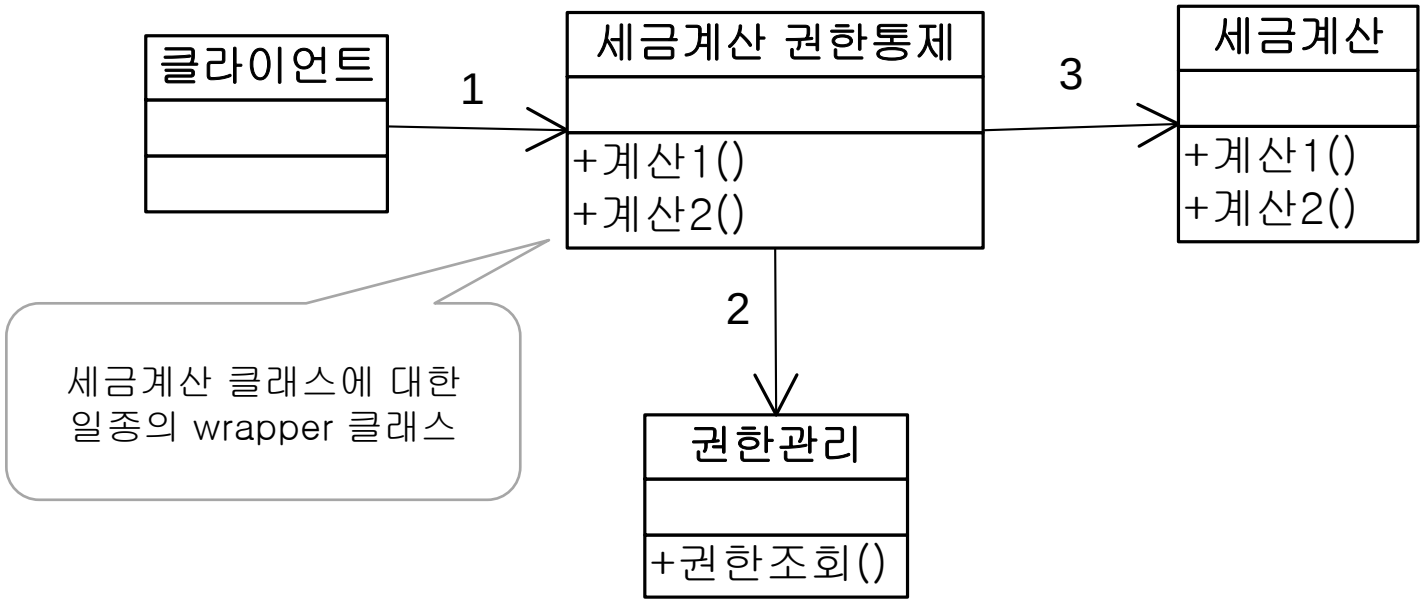
- 장점
  - 클라이언트 유지보수 문제가 없다

더 좋은 방법은?

- 단일 책임 원칙
  - 하나의 객체는 하나의 책임
  - 책임 단위로 객체 분리
  - 책임 = 유지보수 이유

- 권한 정책 변경이 유지보수 이유
  → 권한 정책 객체가 분리되어야 함

권한 정책 객체 분리


클라이언트
1
세금계산 권한통제
+계산1()
+계산2()
3
세금계산
+계산1()
+계산2()
2
권한관리
+권한조회()
세금계산 클래스에 대한
일종의 wrapper 클래스

권한 정책 객체 분리

- 세금계산 클래스, 권한관리 클래스
  → 깨끗한 서비스 클래스
  → 좋은 유지보수성, 재사용성

- 권한 통제 클래스
  → 유지보수 대상

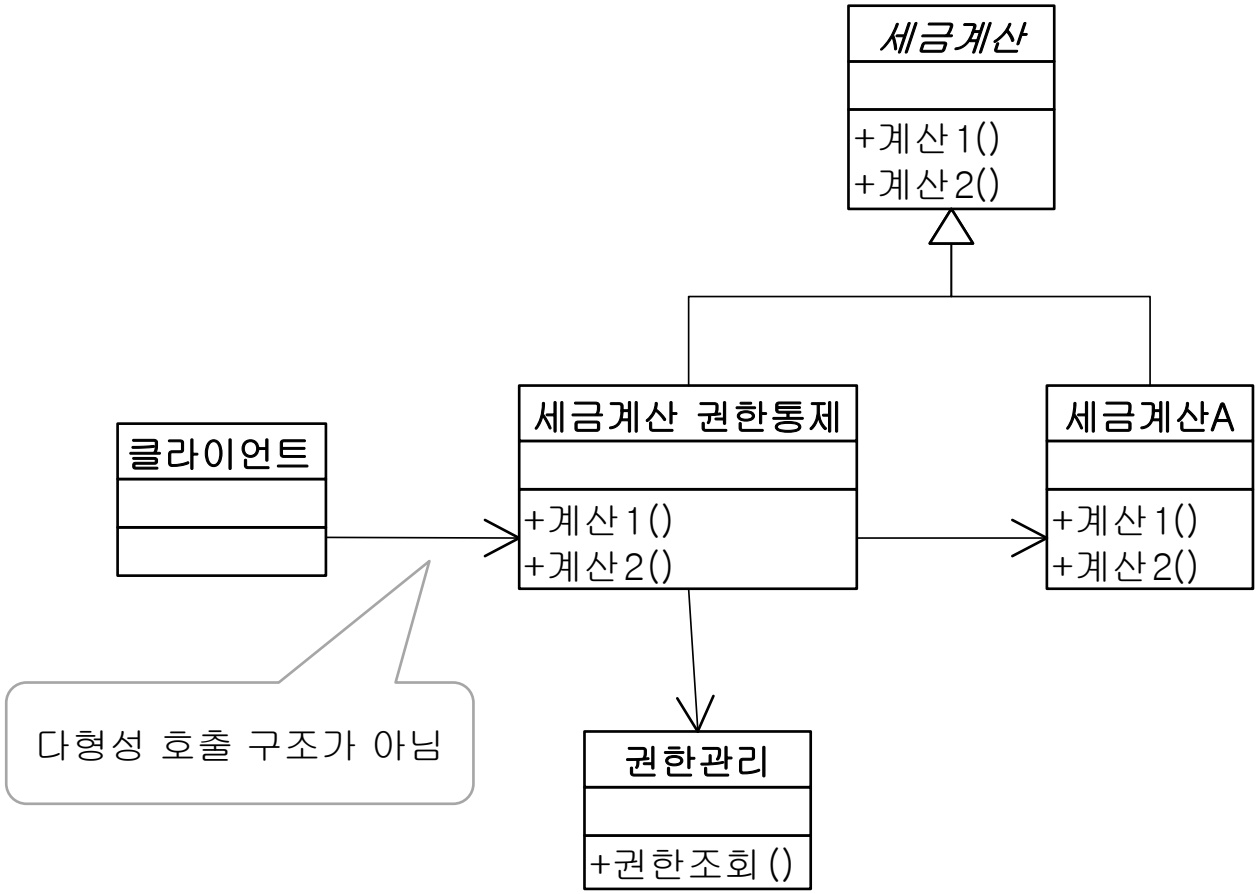
- 권한 통제 정책이 바뀌면
  → 권한 통제 클래스 수정 (클라이언트 유지보수 문제가 없다)
  → 새 권한 통제 클래스로 교체 (클라이언트 코드를 수정해야 하나?)

권한 정책 객체 분리

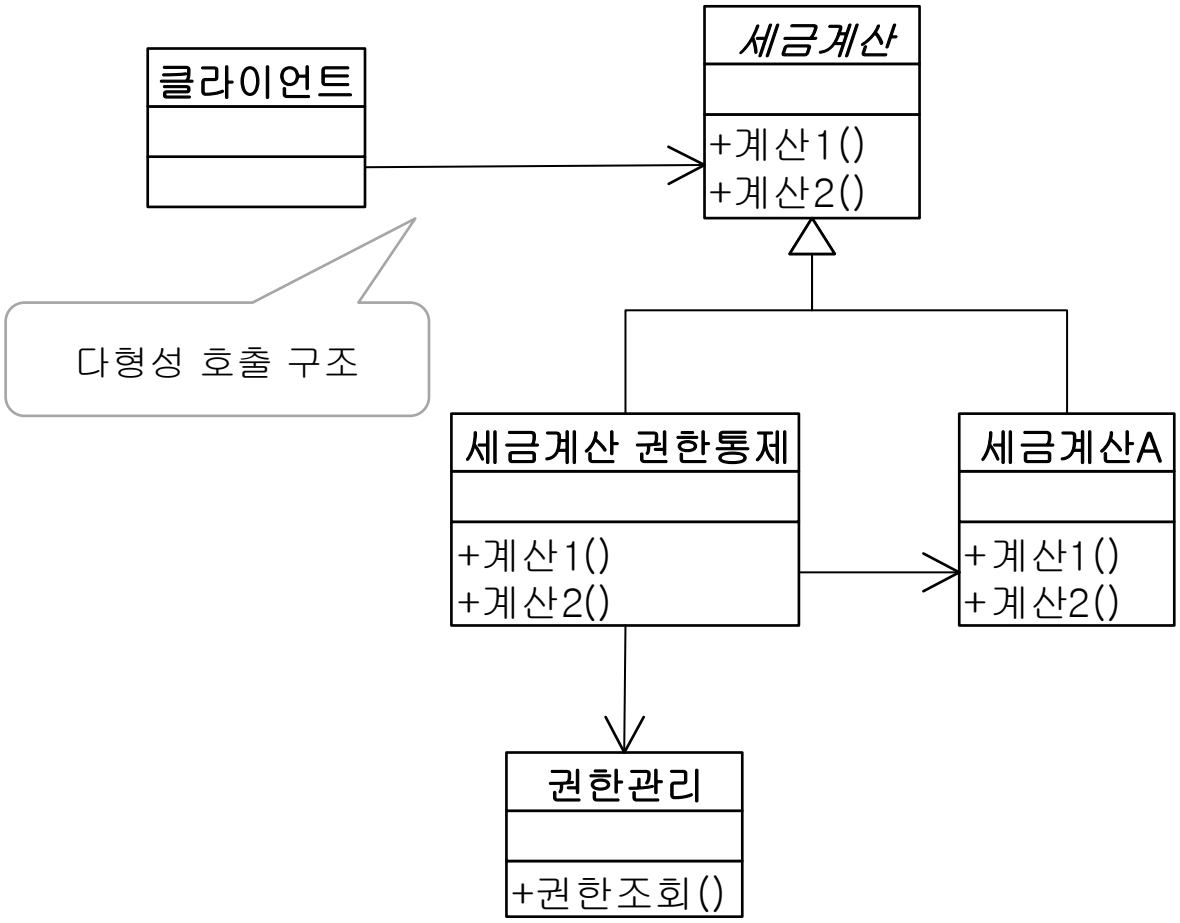
- 새 권한 통제 클래스로 교체될 때
  - 클라이언트를 유지보수 해야한다 (X)
  - 클라이언트 코드 수정 없이 교체할 수 있다 (O)

- 구현 방법은?

다형성 구현 #1


세금계산
+계산1()
+계산2()
클라이언트
세금계산 권한통제
+계산1()
+계산2()
세금계산A
+계산1()
+계산2()
다형성 호출 구조가 아님
권한관리
+권한조회()

다형성 구현 #1

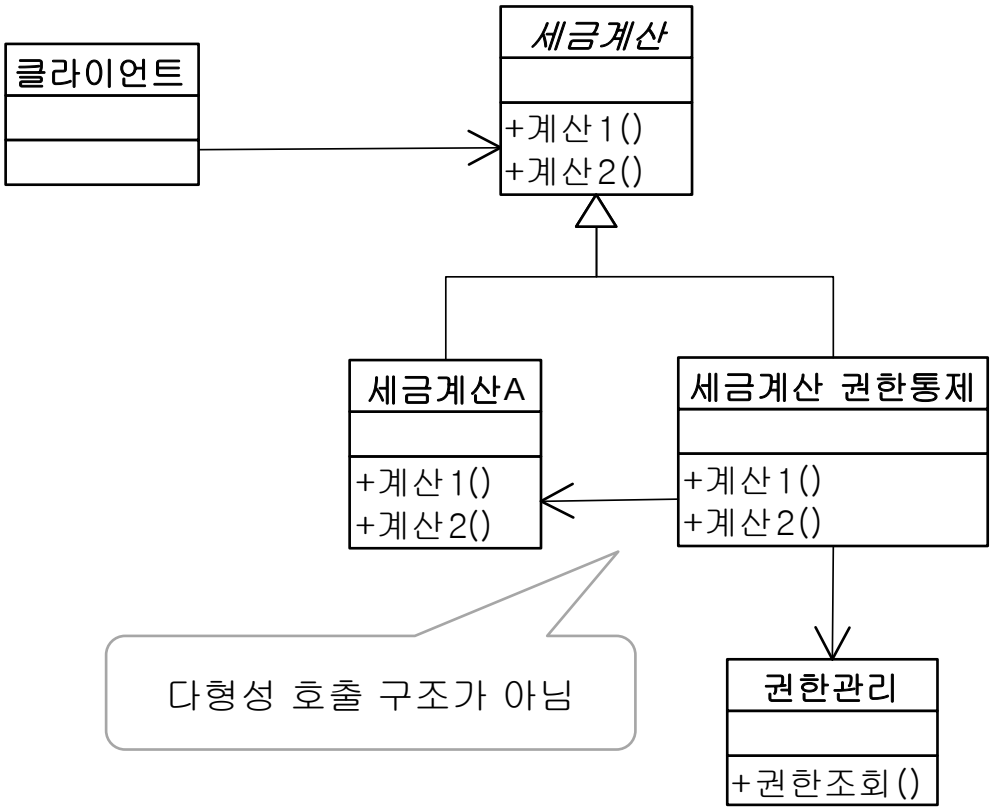

클라이언트
세금계산
+계산1()
+계산2()
다형성 호출 구조
세금계산 권한통제
+계산1()
+계산2()
세금계산A
+계산1()
+계산2()
권한관리
+권한조회()

다형성 구현 #1

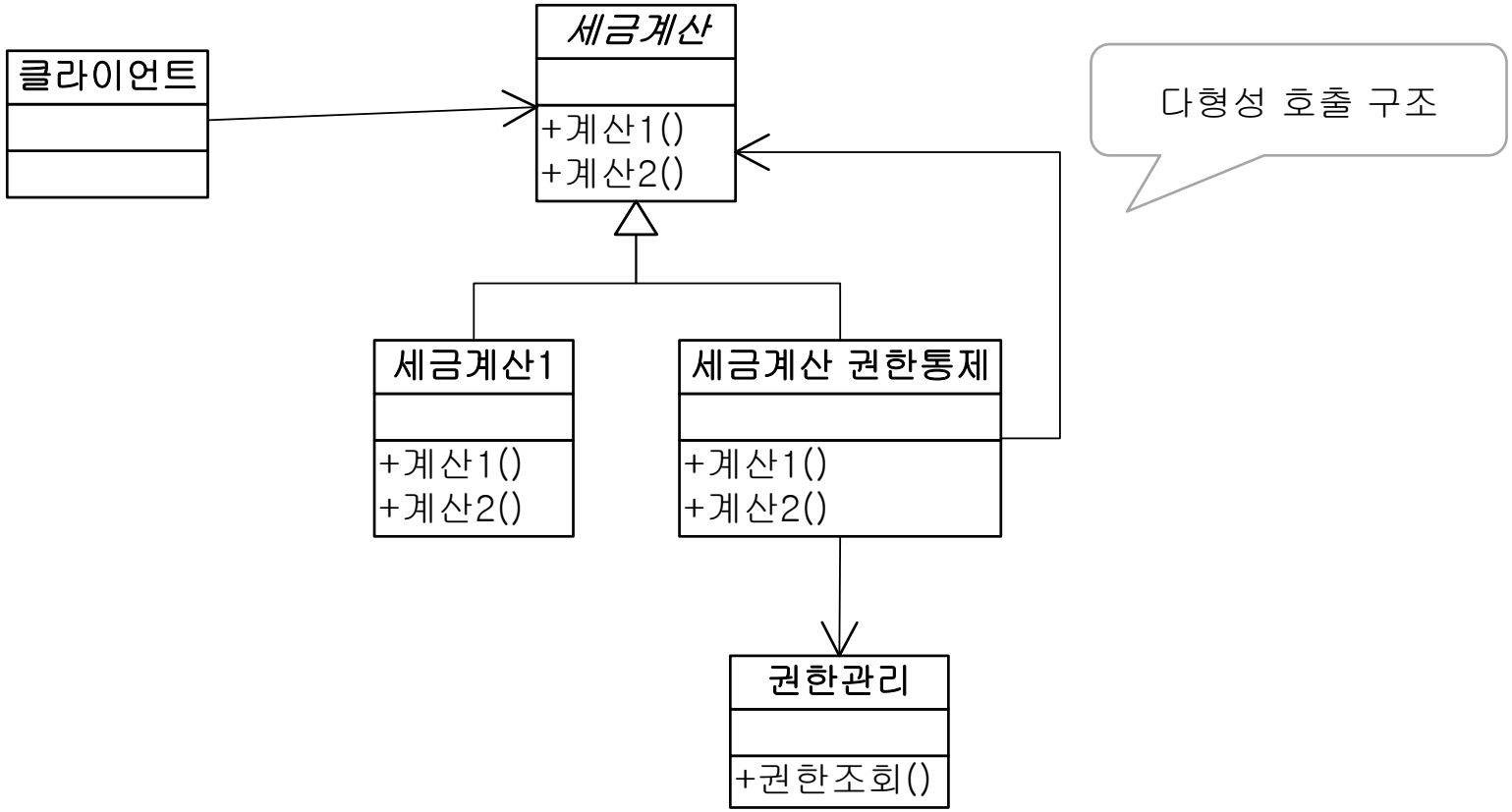
- 다형성 구조 구현
  - 권한 통제 클래스도 세금계산 인터페이스를 상속 받는다
  - 권한 통제 클래스의 사용 방법이 세금계산과 같다

- 권한 통제 클래스가 교체되어도
  클라이언트 코드 수정할 필요 없다

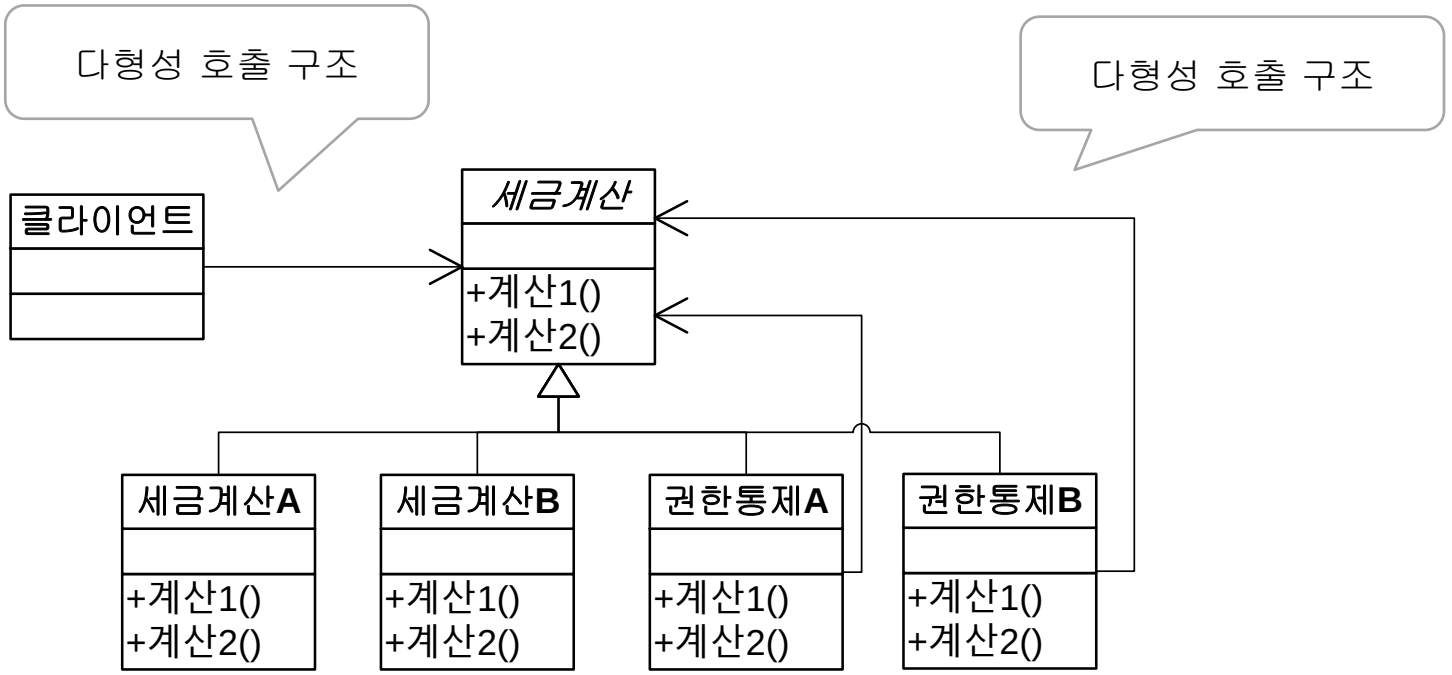
다형성 구현 #2


클라이언트
세금계산
+계산1()
+계산2()
세금계산A
+계산1()
+계산2()
세금계산 권한통제
+계산1()
+계산2()
다형성 호출 구조가 아님
권한관리
+권한조회()

다형성 구현 #2 – Proxy 패턴

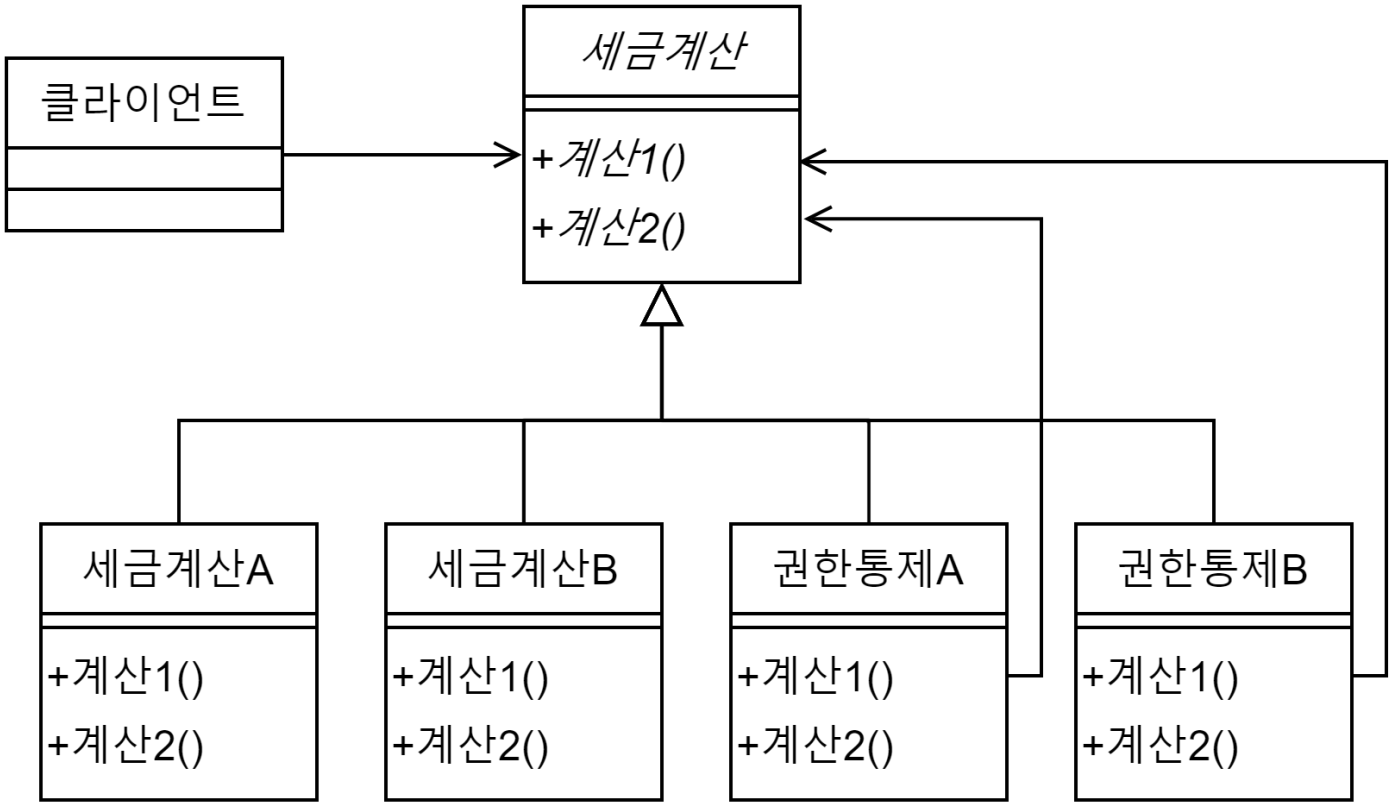


Proxy 패턴

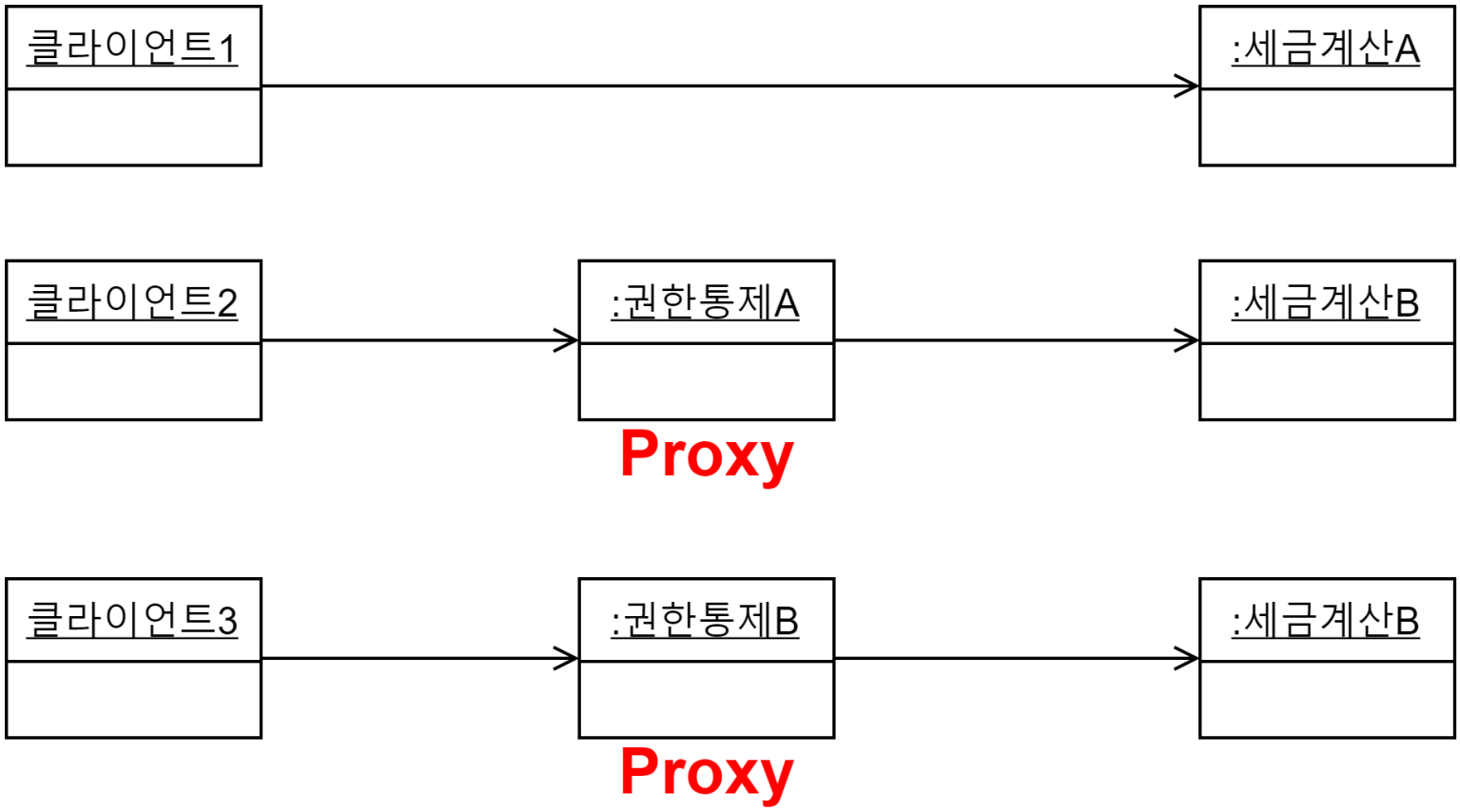


- 필요에 따라 여러 버전의 부품을 구현
- 동일한 클라이언트 코드로 여러 부품 사용 가능
- 권한 관리 클래스는 그림에서 생략함

Proxy 패턴


클라이언트
세금계산
+계산1()
+계산2()
세금계산A
+계산1()
+계산2()
세금계산B
+계산1()
+계산2()
권한통제A
+계산1()
+계산2()
권한통제B
+계산1()
+계산2()

Proxy 패턴 – 객체 다이어그램



Proxy 패턴 – 결과

- 권한 통제가 필요하면
  - 적절한 권한 통제 객체를 선택하여 클라이언트와 세금 계산 객체 사이에 끼워 넣으면 된다

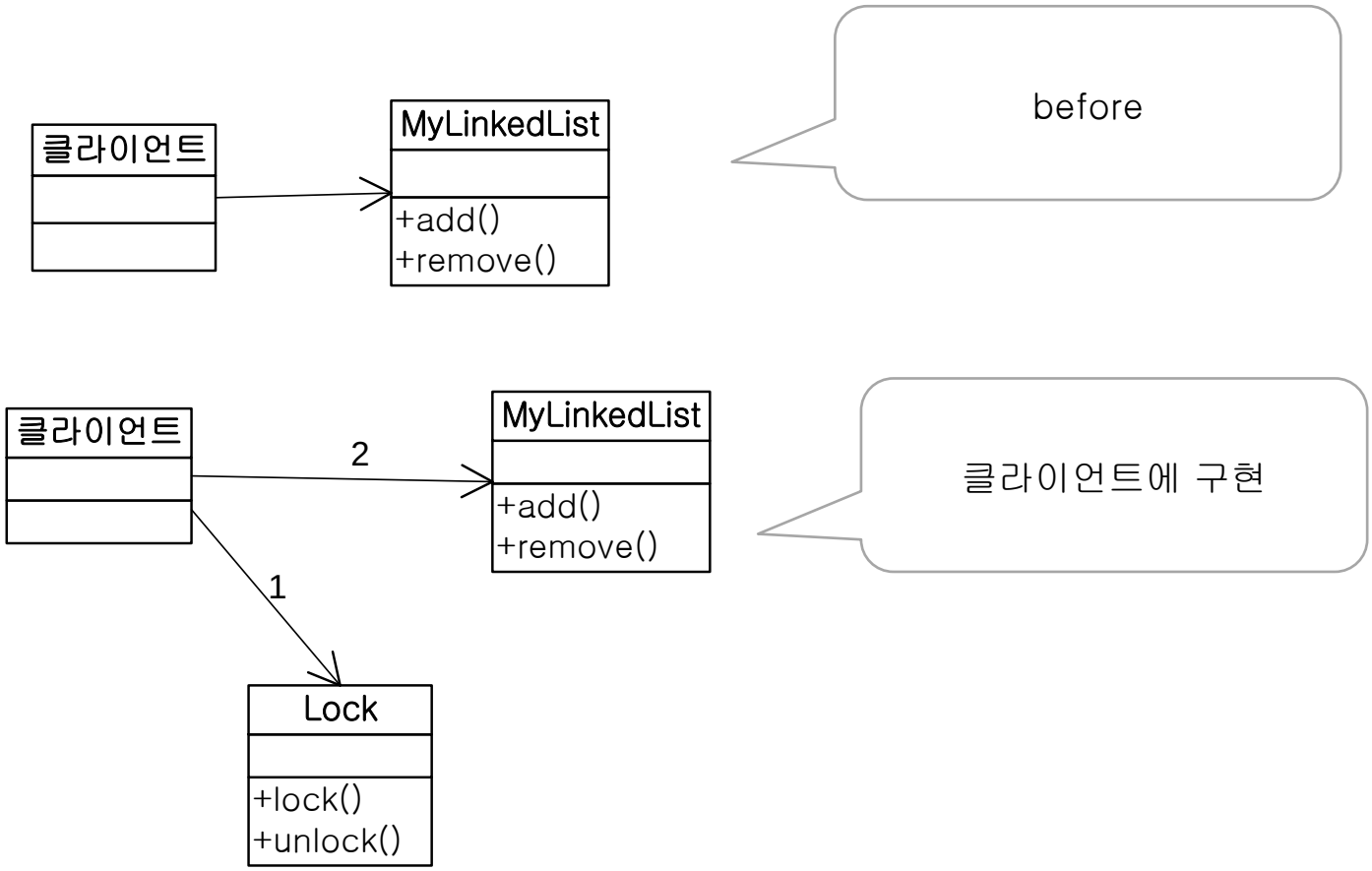
- 권한 통제가 필요 없으면
  - 클라이언트와 세금 계산 객체를 바로 연결하면 된다

- 권한 정책이 바뀌면
  - 바뀐 정책을 구현한 권한 통제 객체로 교체한다

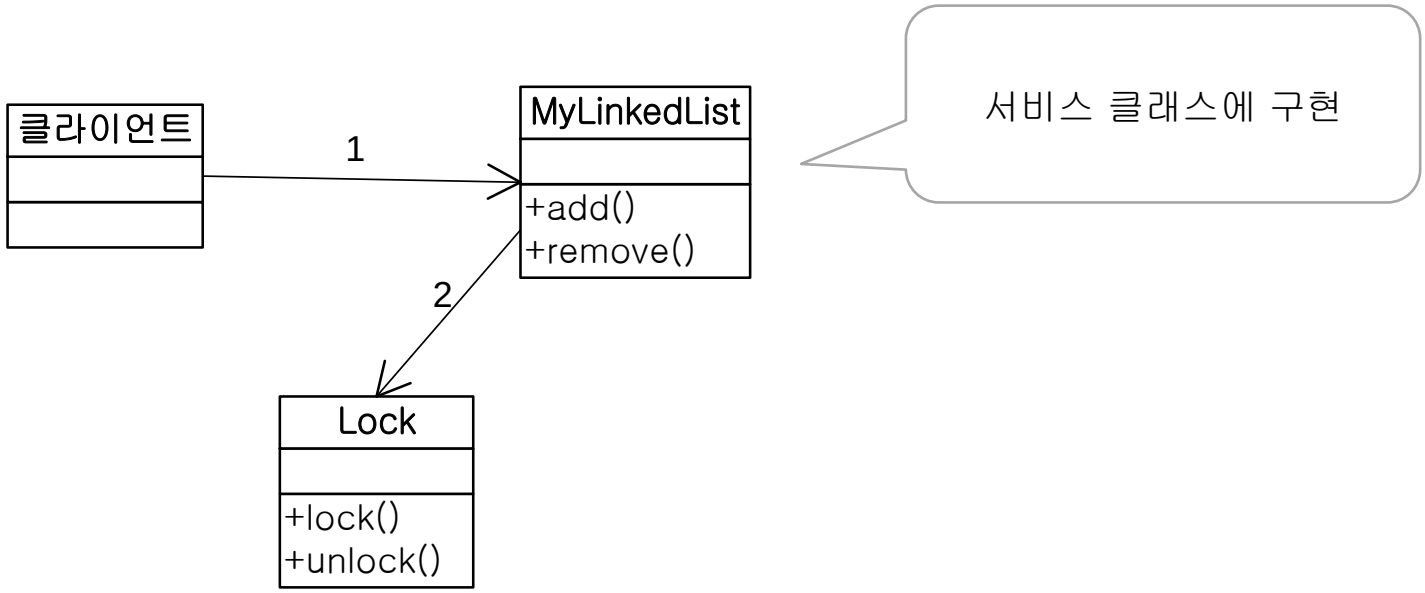
Proxy – 문제

- thread safe한 자료구조 클래스 구현하기
- lock/unlock 하는 코드를 어디에 구현?

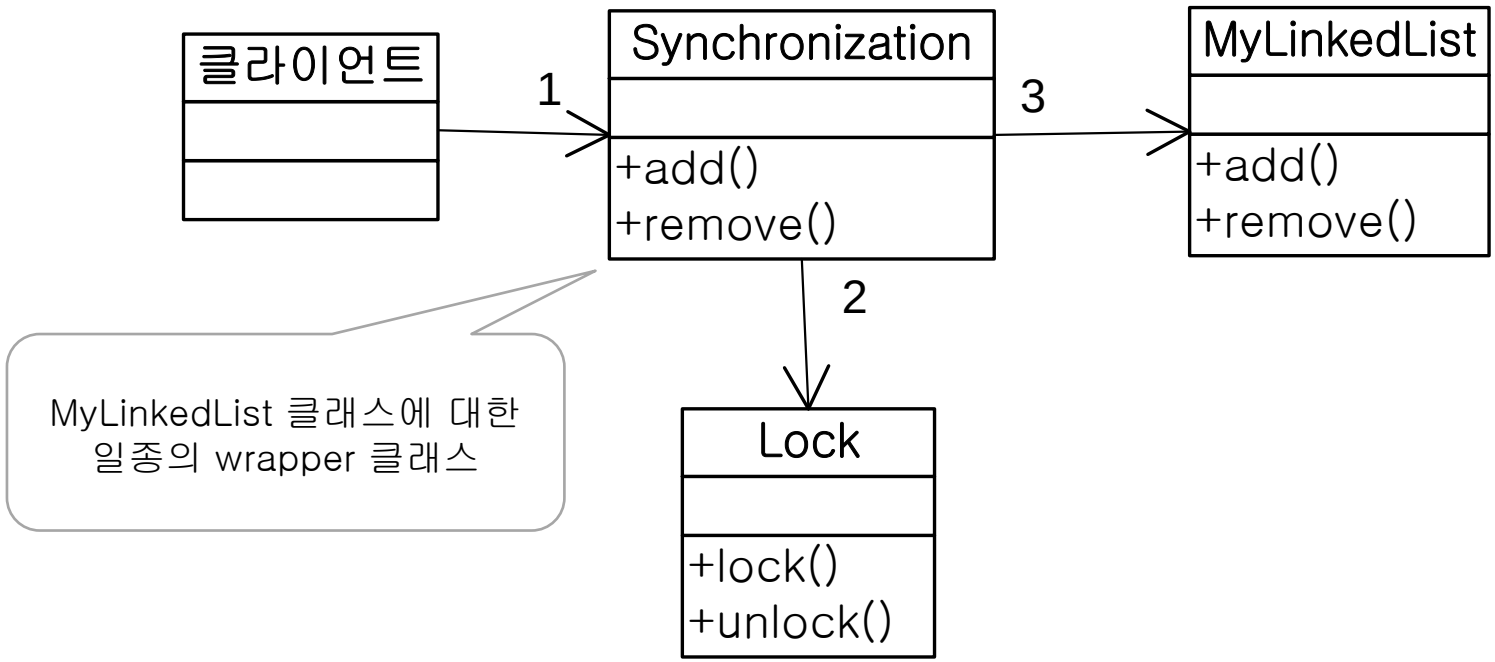
동기화 제어 구현 문제


클라이언트
MyLinkedList
+add()
+remove()
before
클라이언트
2
MyLinkedList
+add()
+remove()
클라이언트에 구현
1
Lock
+lock()
+unlock()

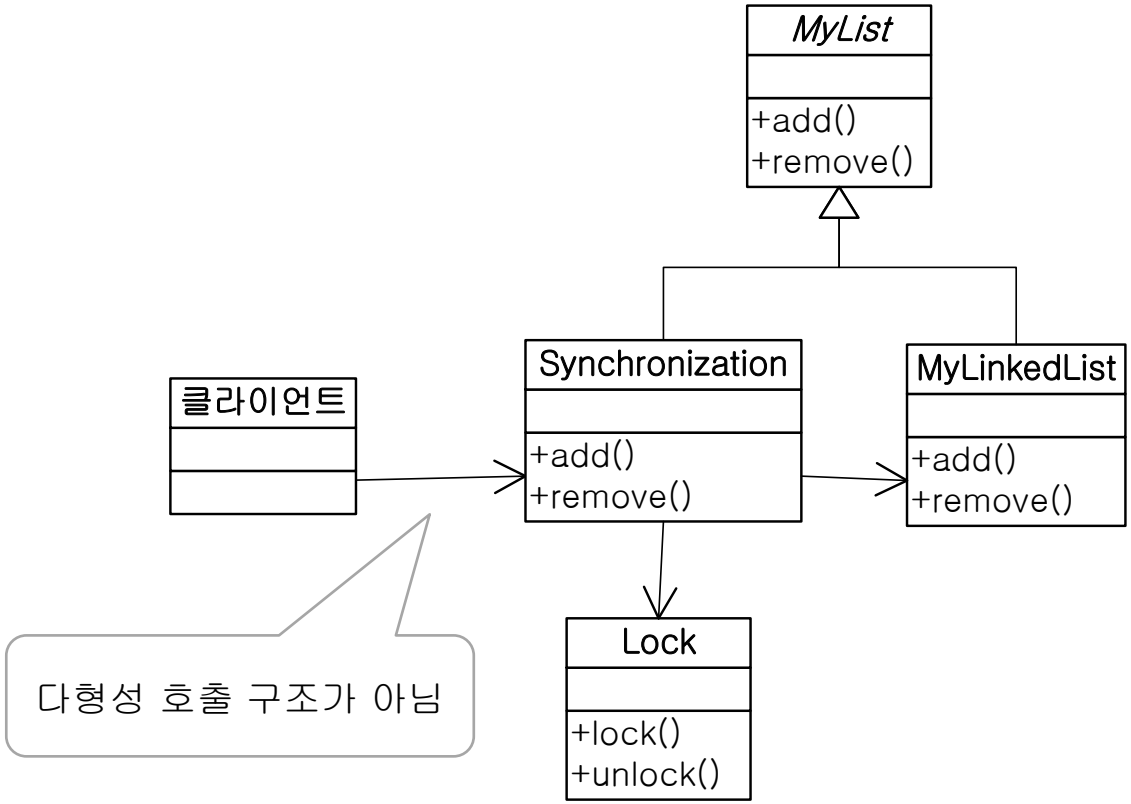
동기화 제어 구현 문제


클라이언트
1
MyLinkedList
+add()
+remove()
서비스 클래스에 구현
2
Lock
+lock()
+unlock()

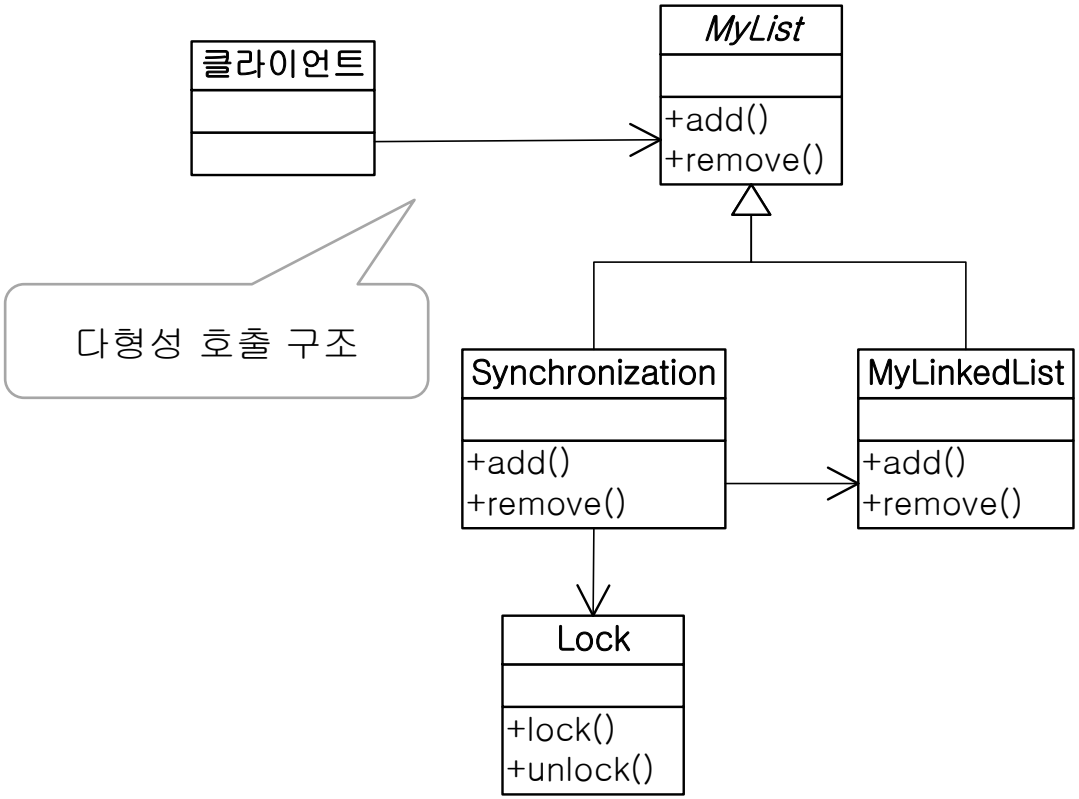
동기화 제어 분리 구현


클라이언트
Synchronization
+add()
+remove()
MyLinkedList
+add()
+remove()
1
2
3
Lock
+lock()
+unlock()
MyLinkedList 클래스에 대한
일종의 wrapper 클래스

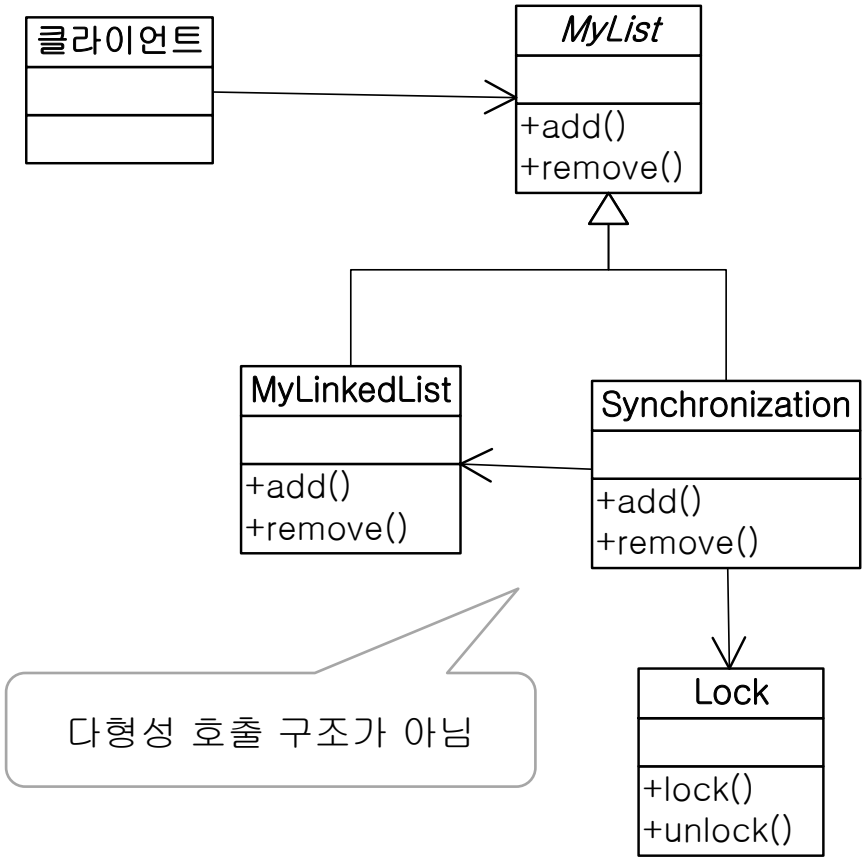
다형성 구현 #1


MyList
+add()
+remove()
Synchronization
+add()
+remove()
MyLinkedList
+add()
+remove()
클라이언트
Lock
+lock()
+unlock()
다형성 호출 구조가 아님

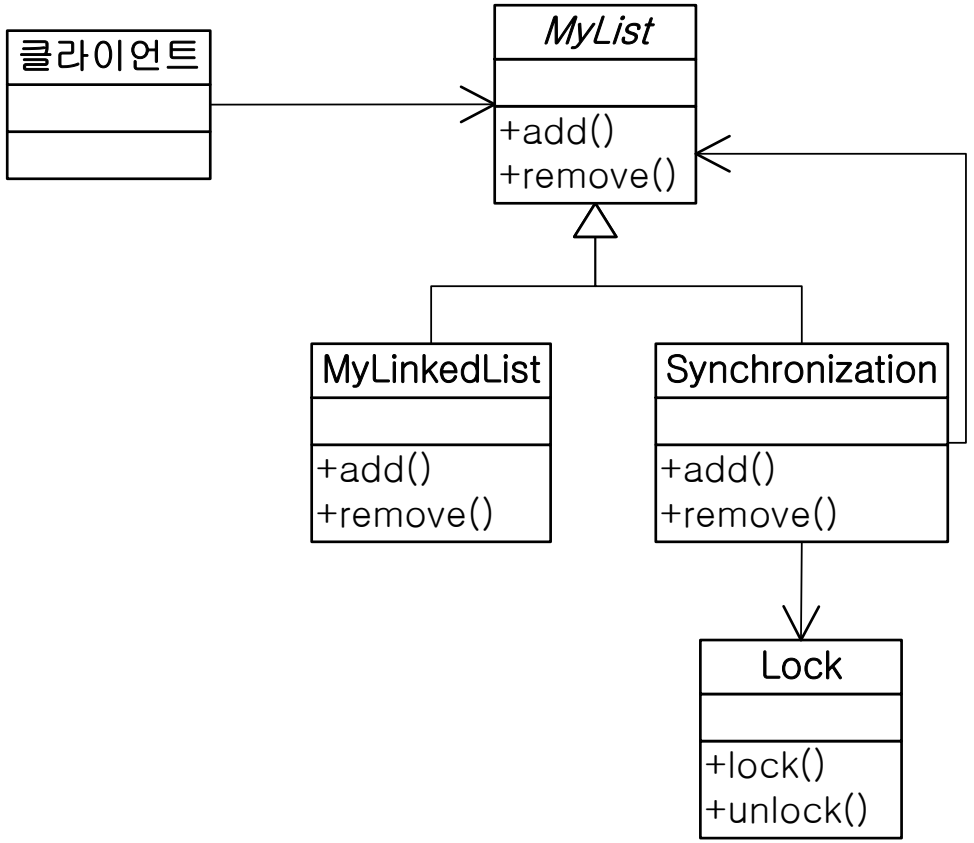
다형성 구현 #2


클라이언트
MyList
+add()
+remove()
다형성 호출 구조
Synchronization
+add()
+remove()
MyLinkedList
+add()
+remove()
Lock
+lock()
+unlock()

다형성 구현 #2

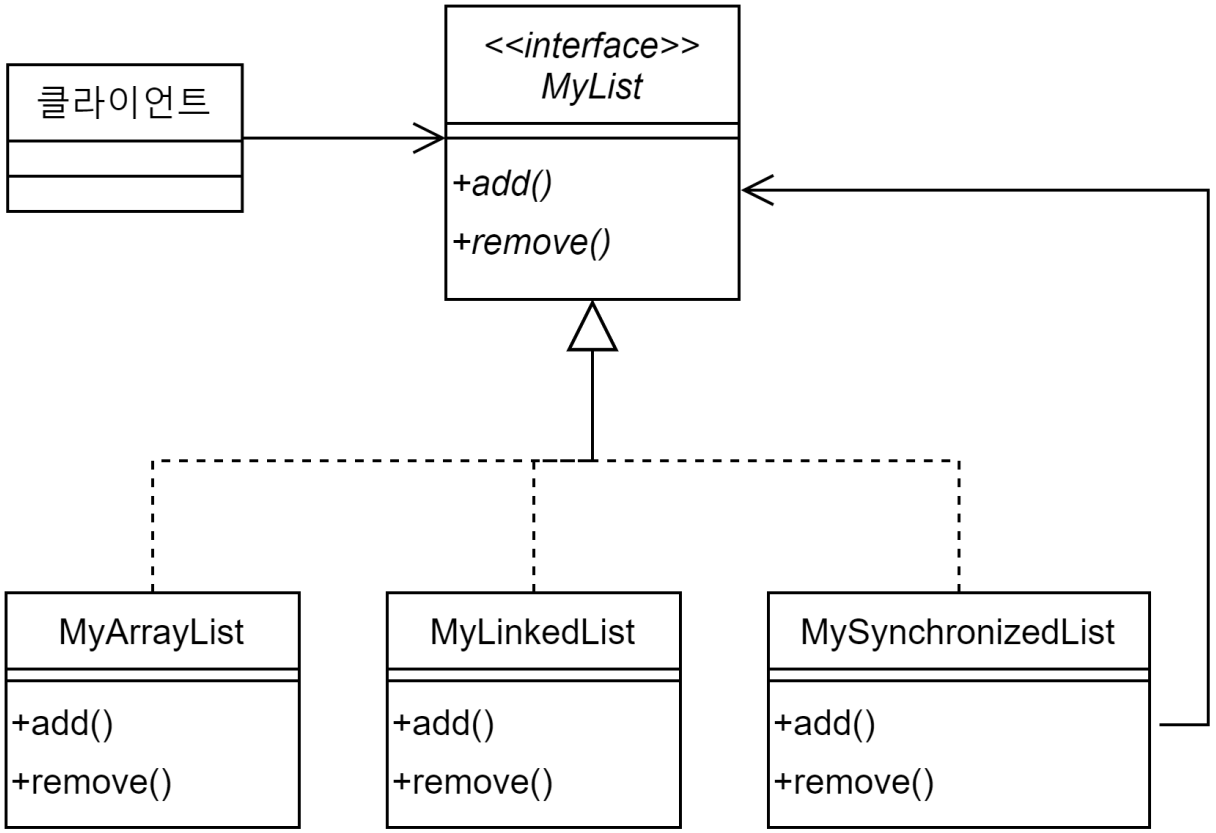

클라이언트
MyList
+add()
+remove()
MyLinkedList
+add()
+remove()
Synchronization
+add()
+remove()
Lock
+lock()
+unlock()
다형성 호출 구조가 아님

다형성 구현 #2 – Proxy 패턴

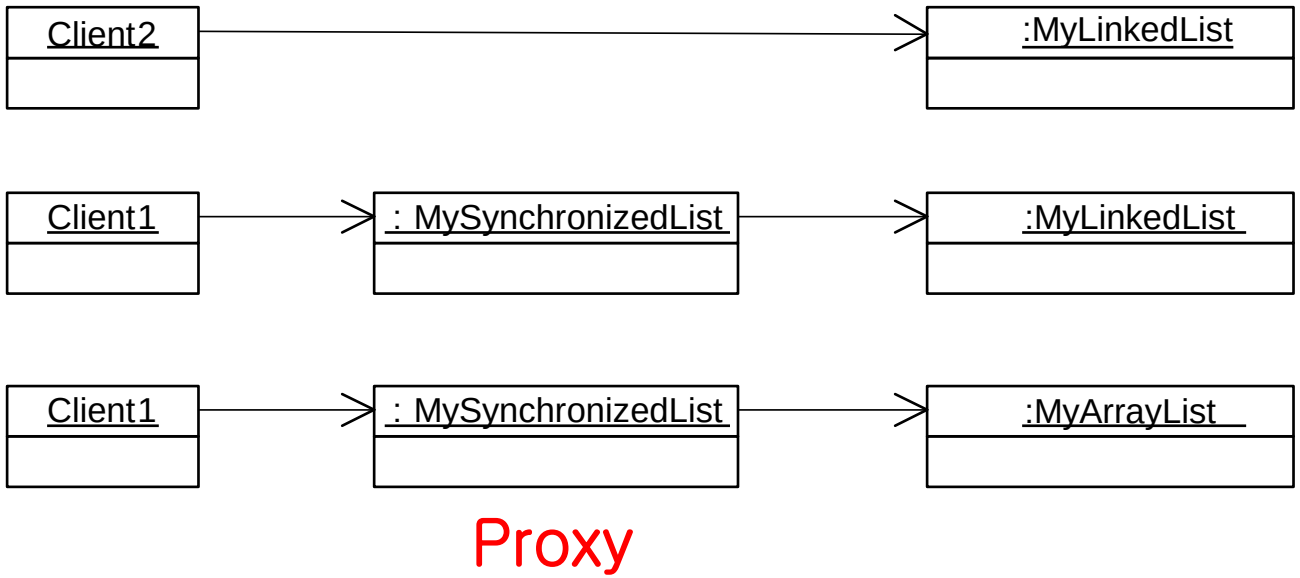


다형성 호출 구조

Proxy – 구조


클라이언트
<<interface>>
MyList
+add()
+remove()
MyArrayList
+add()
+remove()
MyLinkedList
+add()
+remove()
MySynchronizedList
+add()
+remove()

Proxy – 구조


Client2
:MyLinkedList
Client1
: MySynchronizedList
:MyLinkedList
Client1
: MySynchronizedList
:MyArrayList
Proxy

예제 코드 분석

- proxy.e1 Example1A.java
  - MyArrayList/MyLinkedList 목록 선두에 999 를 추가하고 제거하기를 1000번 반복
  - 목록에 변화 없다

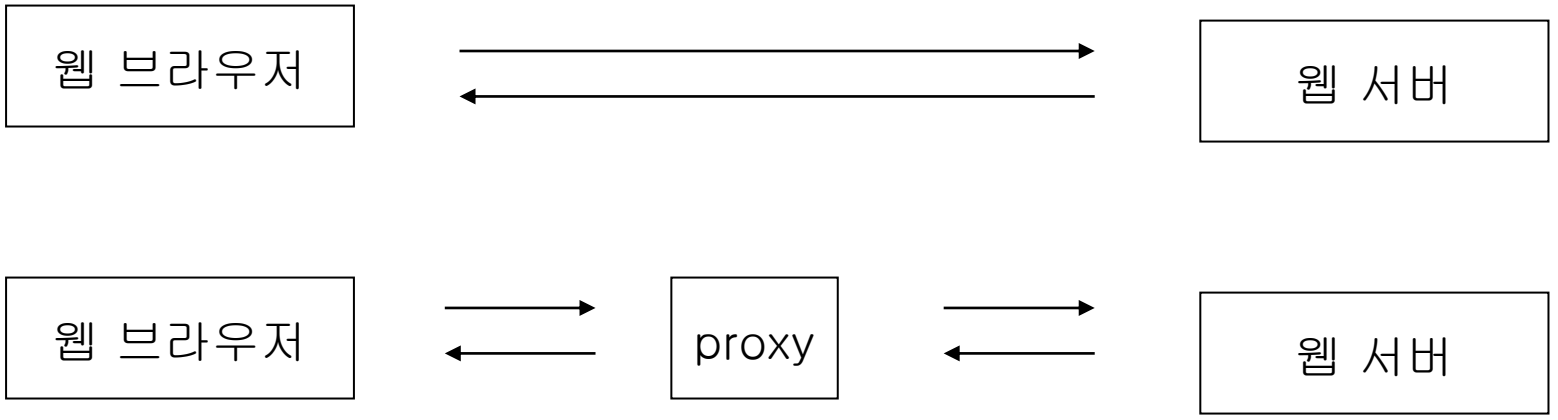
- proxy.e1 Example1B.java
  - 위 작업을 multi thread 실행 → 충돌 발생

- proxy.e2 proxy 패턴 적용 → 충돌 해결함

- proxy e1, e2 예제 소스코드를 분석하시오

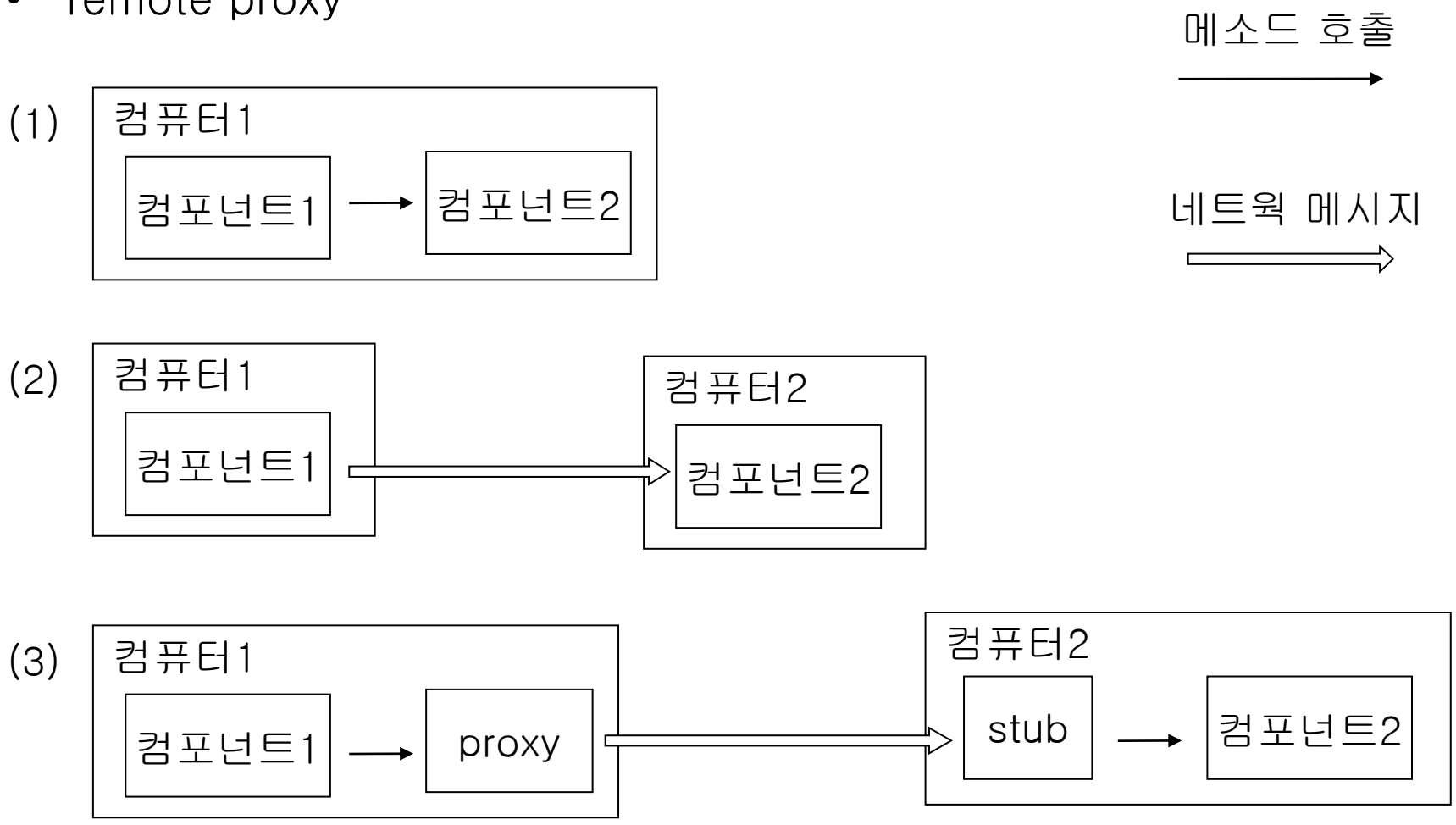
Proxy의 예

- 웹 proxy 서버
  - 캐쉬의 역할
  - 방화벽의 역할



Proxy의 예

- remote proxy



Proxy의 예

- remote proxy

- 컴포넌트1, 컴포넌트2의 구현
  (1) 는 메소드 호출로 구현. 쉽다.
  (2) 는 네트웍 메시지 전달을 구현해야함. 어렵다.
  (3) 은 메소드 호출로 구현. 쉽다

- proxy 와 stub 은 저절로 생성되므로 구현할 필요 없음.
- proxy 는 컴포넌트2와 같은 인터페이스를 상속받음
- 컴포넌트1을 구현할 때 proxy는 보이지 않고
  직접 컴포넌트2의 메소드를 호출하는 것처럼 구현됨.

Types of Proxies

- Remote Proxy – Provides a reference to an object located in a different address space on the same or different machine

- Virtual Proxy – Allows the creation of a memory intensive object on demand. The object will not be created until it is really needed.

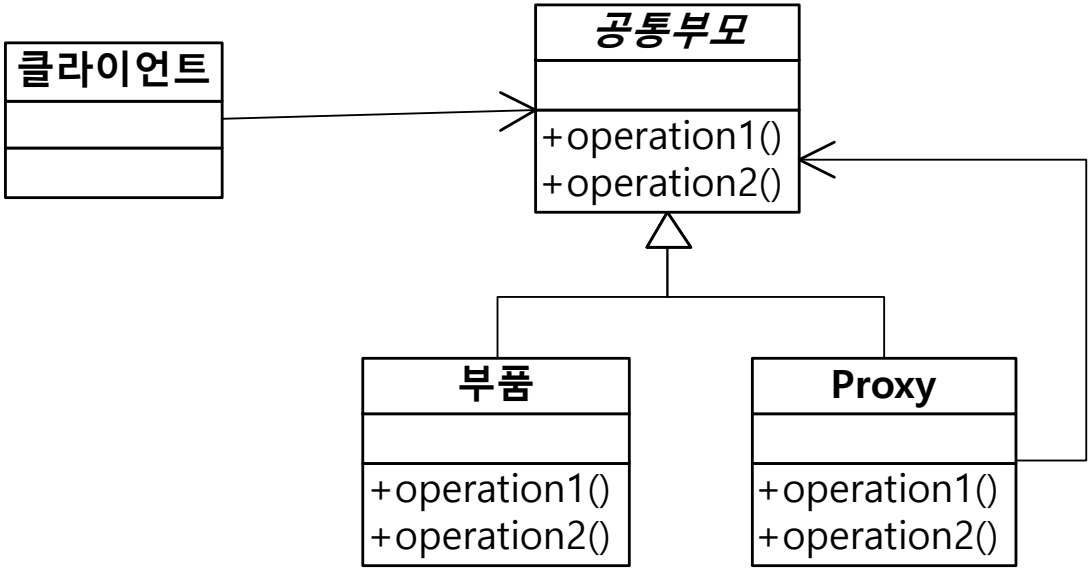
- Copy-On-Write Proxy – Defers copying (cloning) a target object until required by client actions. Really a form of virtual proxy.

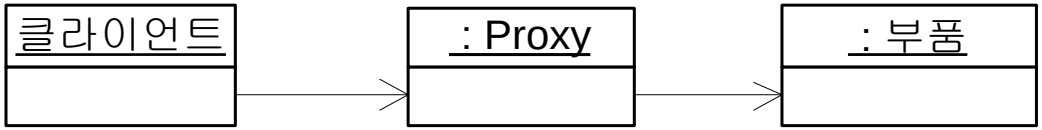
- Protection (Access) Proxy – Provides different clients with different levels of access to a target object

Types of Proxies

- Cache Proxy – Provides temporary storage of the results of expensive target operations so that multiple clients can share the results
- Firewall Proxy – Protects targets from bad clients (or vice versa)
- Synchronization Proxy – Provides multiple accesses to a target object
- Smart Reference Proxy – Provides additional actions whenever a target object is referenced such as counting the number of references to the object

Proxy 패턴 요약


클라이언트
공통부모
+operation1()
+operation2()
부품
+operation1()
+operation2()
Proxy
+operation1()
+operation2()


클라이언트
: Proxy
: 부품

Proxy 패턴 요약

- 클라이언트가 부품을 사용하고 있는 중간에, proxy 객체가 사이에 끼어 들어가서, 클라이언트의 부품에 대한 접근을 통제한다.

- proxy 객체와 부품은 다형성이 구현되어서, 중간에 proxy 객체의 존재 유무와 무관하게 클라이언트는 동일한 코드로 부품을 사용한다.

Read Write Lock

- 읽기와 쓰기를 구분해서 lock
  - 읽기 전에는 읽기 lock
  - 쓰기 전에는 쓰기 lock

- 읽기 lock 걸려 있으면
  - 새 읽기 lock 허용, 쓰기 lock 금지

- 쓰기 lock 걸려 있을 때
  - 모든 lock 금지

Read Write Lock - java

```
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

lock.readLock().lock();
try {
    읽기
} finally {
    lock.readLock().unlock();
}

lock.writeLock().lock();
try {
    쓰기
} finally {
    lock.writeLock().unlock();
}
```

Read Write Lock – cpp

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);



// C++ read write lock 표준 클래스는 std::shared_mutex 인데,
// 아직 g++은 C++ 17 표준을 지원하지 않는다
```

Read Write Lock – cpp

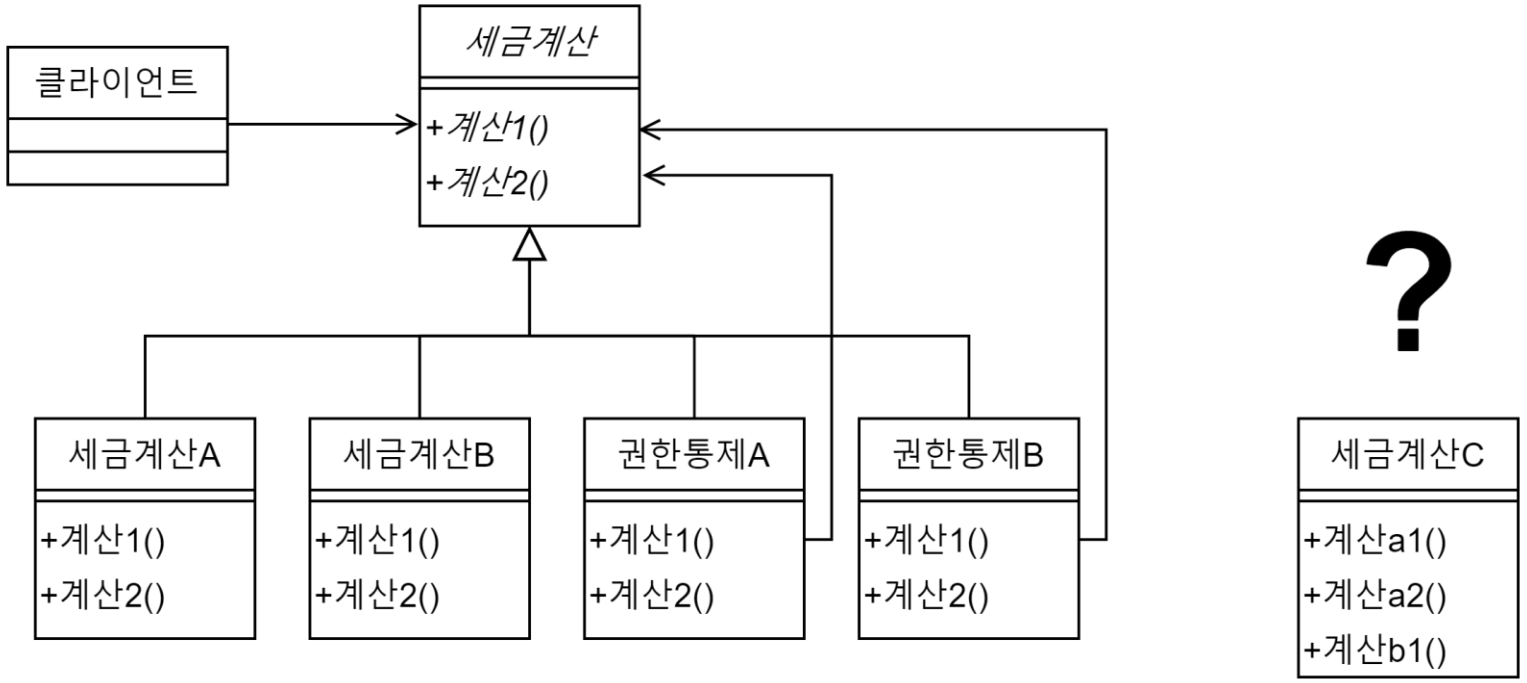
```
pthread_rwlock_t lock;
pthread_rwlock_init(&lock, NULL);

pthread_rwlock_rdlock(&lock);
읽기
pthread_rwlock_unlock(&lock);


pthread_rwlock_wrlock(&lock);
쓰기
pthread_rwlock_unlock(&lock);


pthread_rwlock_destroy(&lock);
```

구현 실습

- proxy e2 예제에
  read write lock 을 활용한 synchronized list 를 구현하시오

Adapter

Adapter – 문제

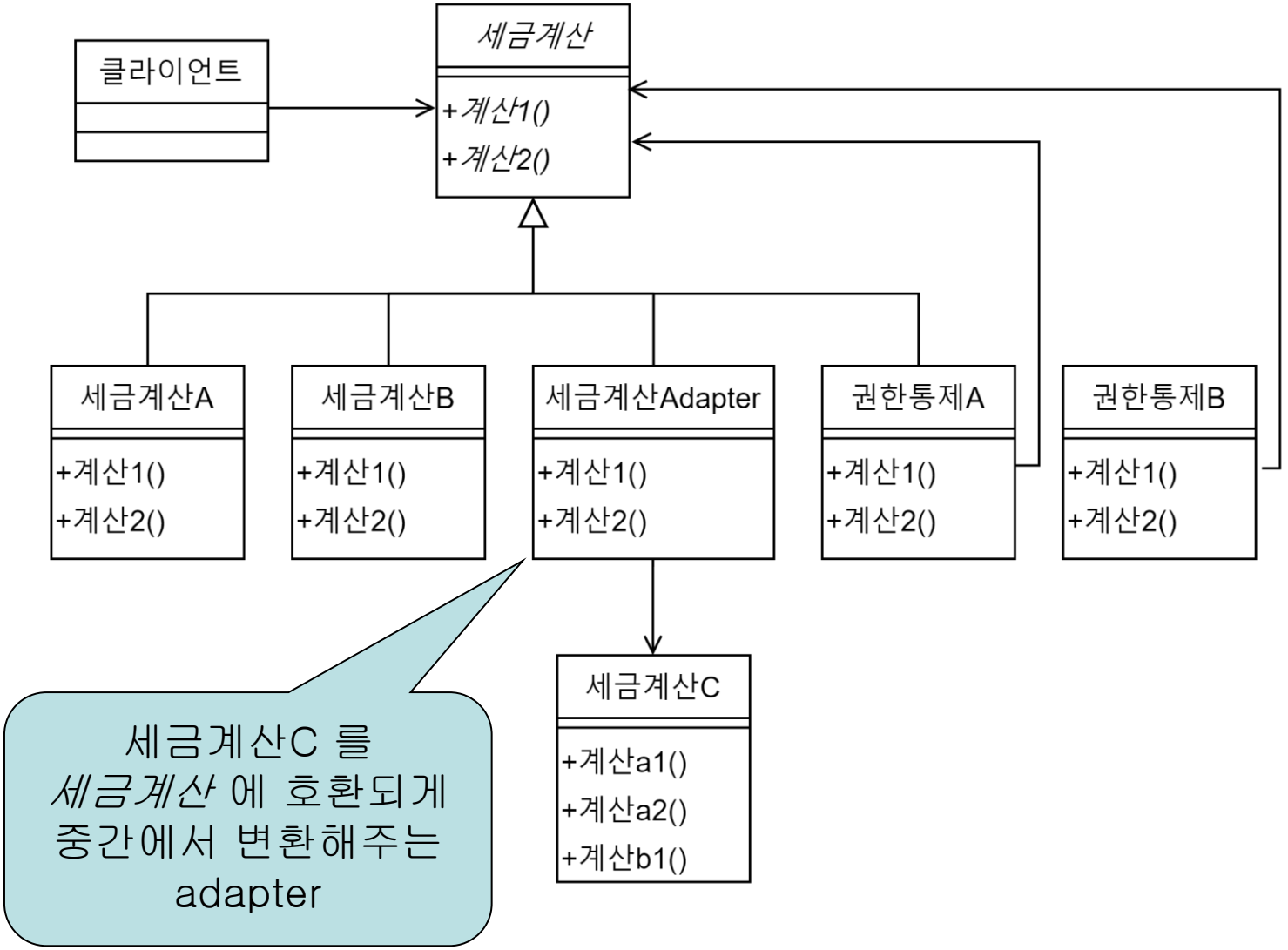

클라이언트
세금계산
+계산1()
+계산2()
세금계산A
+계산1()
+계산2()
세금계산B
+계산1()
+계산2()
권한통제A
+계산1()
+계산2()
권한통제B
+계산1()
+계산2()
?
세금계산C
+계산a1()
+계산a2()
+계산b1()

Adapter - 문제

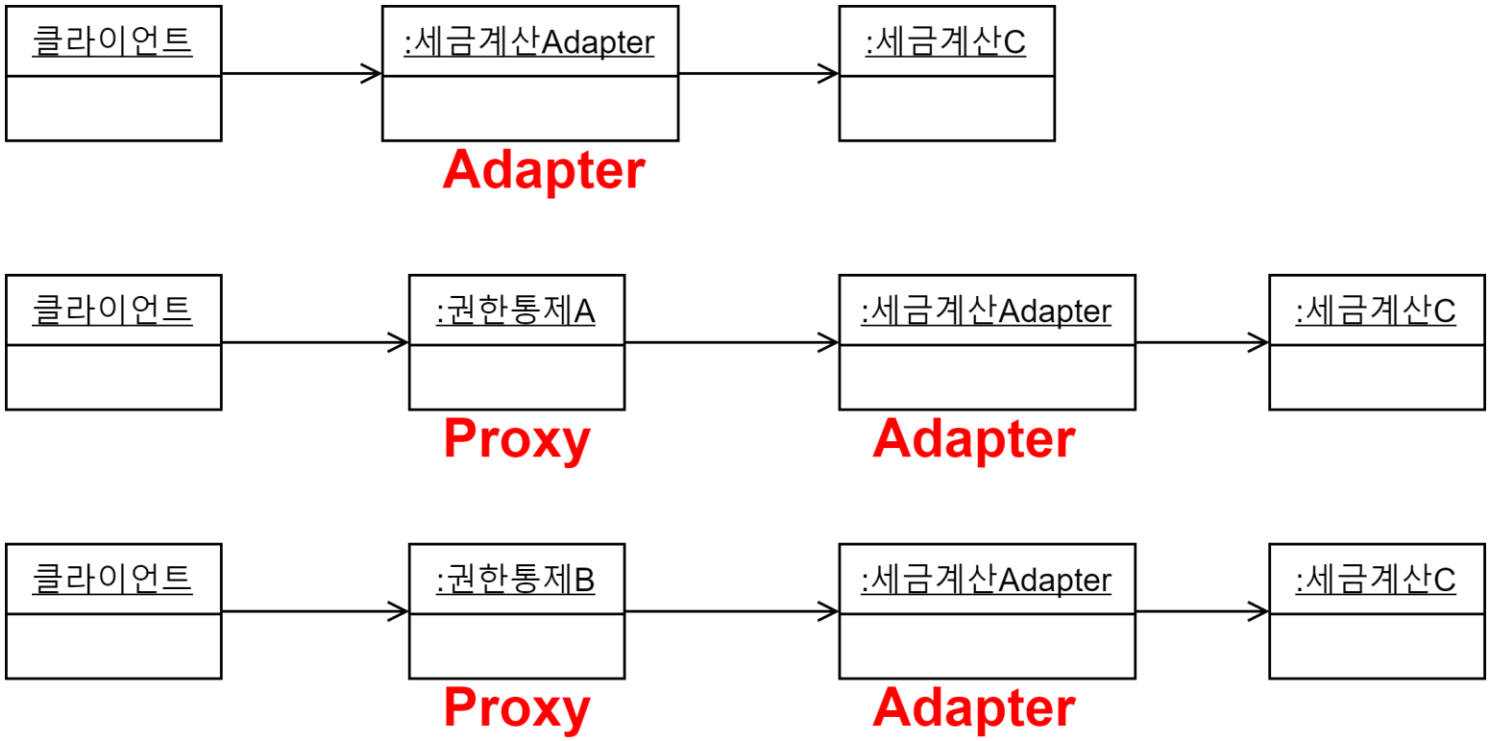
- 새 세법 세금계산 구현
  - 세법이 변해서 세금계산 클래스를 새로 구현해야 한다.
  - 새 세법이 잘 구현된 세금계산C 클래스를 구해왔다
  - 세금계산C 클래스를 재사용하려고 한다
  - 그런데 *세금계산* 인터페이스와 일치하지 않는다

- *세금계산* 인터페이스에 일치하도록 수정하는 방법은?
  - 세금계산C 클래스 수정?
  - 세금계산C 클래스를 상속하여 자식 클래스 구현?
  - adapter 구현?

Adapter – 해결


클라이언트
세금계산
+계산1()
+계산2()
세금계산A
+계산1()
+계산2()
세금계산B
+계산1()
+계산2()
세금계산Adapter
+계산1()
+계산2()
권한통제A
+계산1()
+계산2()
권한통제B
+계산1()
+계산2()
세금계산C
+계산a1()
+계산a2()
+계산b1()
세금계산C 를
세금계산 에 호환되게
중간에서 변환해주는
adapter

Adapter – 해결

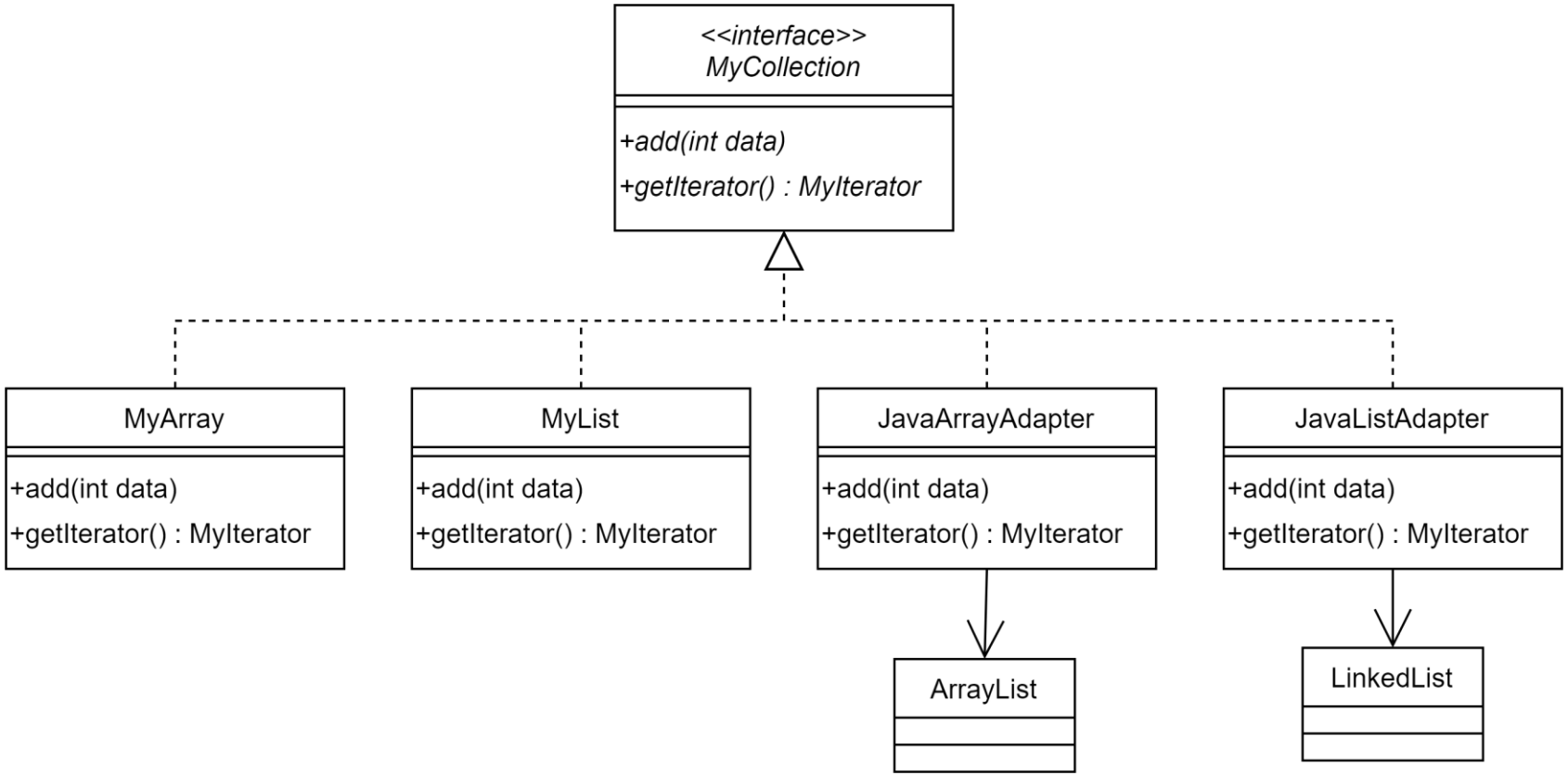

클라이언트
:세금계산Adapter
:세금계산C
Adapter
클라이언트
:권한통제A
:세금계산Adapter
:세금계산C
Proxy
Adapter
클라이언트
:권한통제B
:세금계산Adapter
:세금계산C
Proxy
Adapter

Adapter – 문제

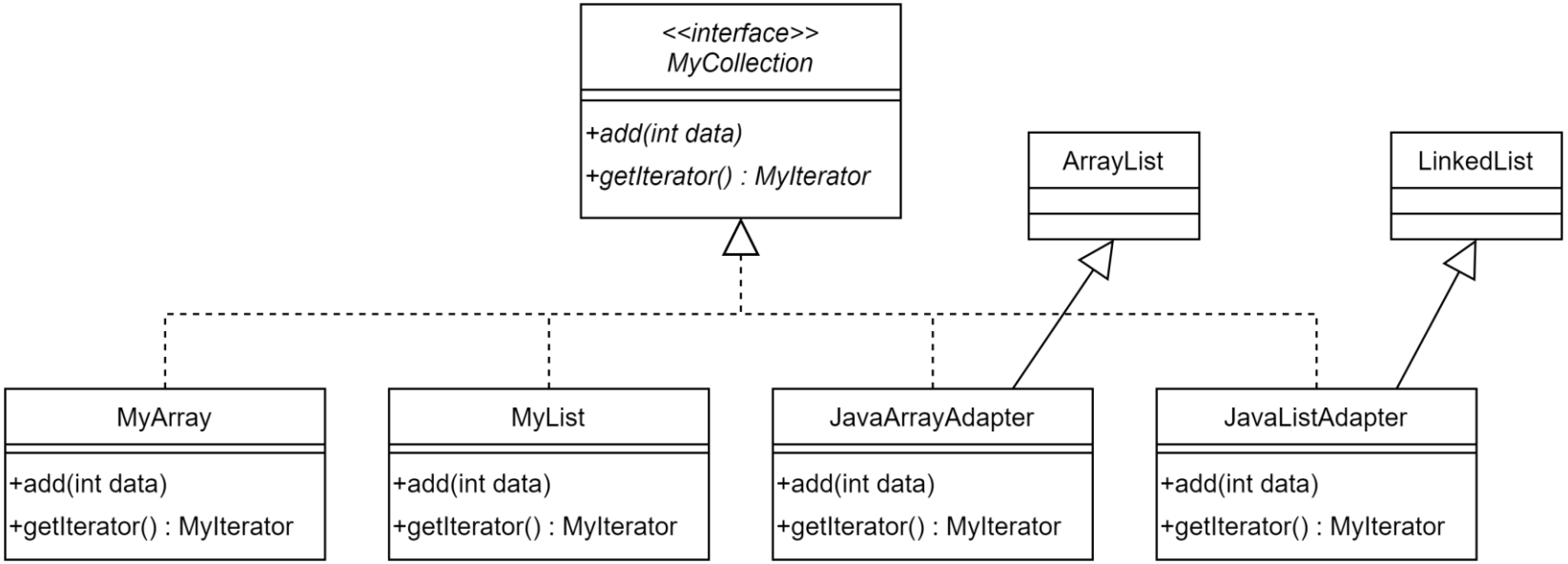
- Java의 ArrayList, LinkedList 소스 코드를 수정하지 않고
  이들을 MyCollection, MyIterator 인터페이스에 맞추는 방법은?

- → 인터페이스를 맞추기 위한 목적의 adapter 클래스를 생성

Adapter – 위임으로 구현



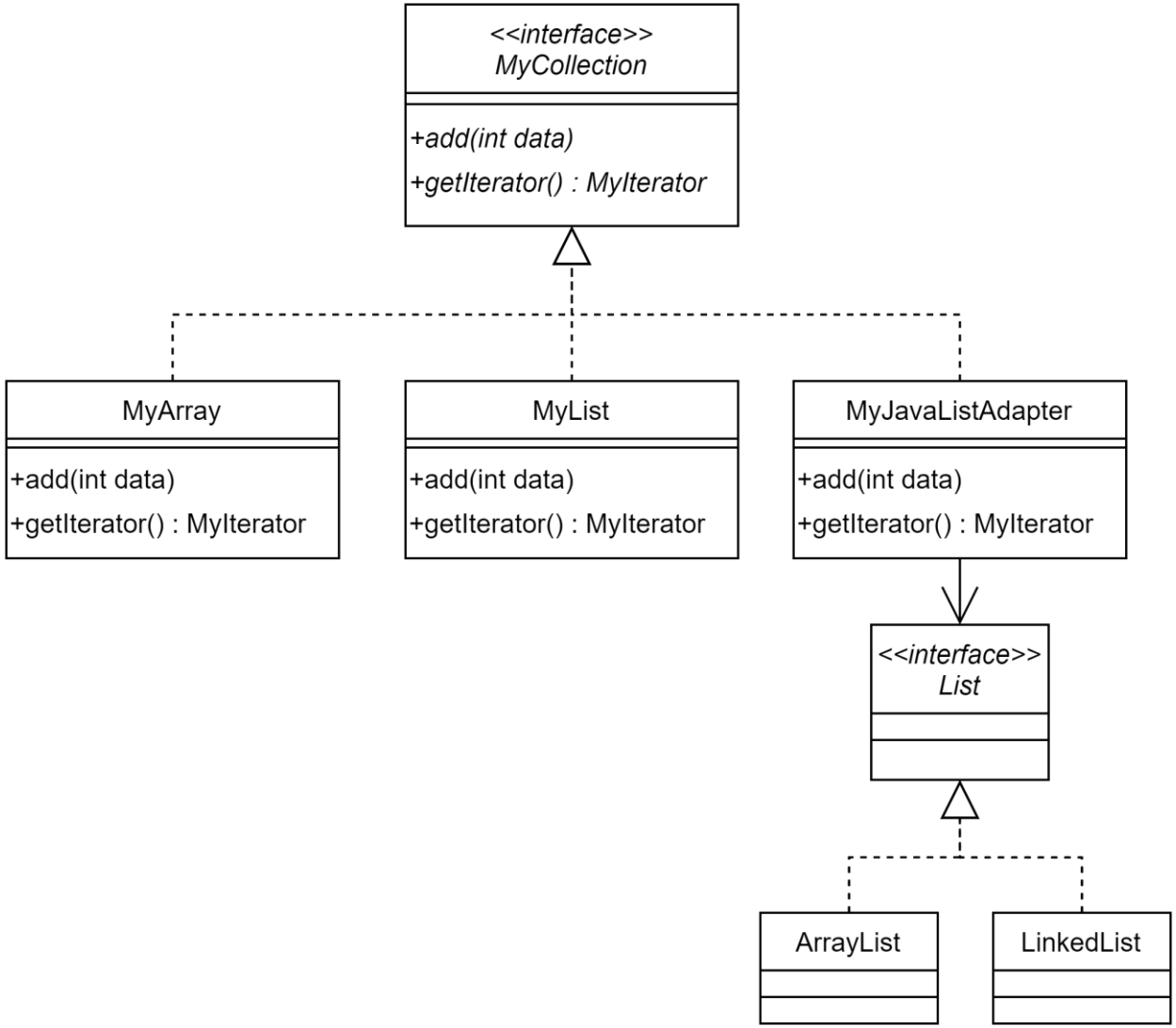
Adapter – 상속으로 구현



Adapter – 위임과 상속

- 상속으로 구현
  - protected member 도 사용 가능

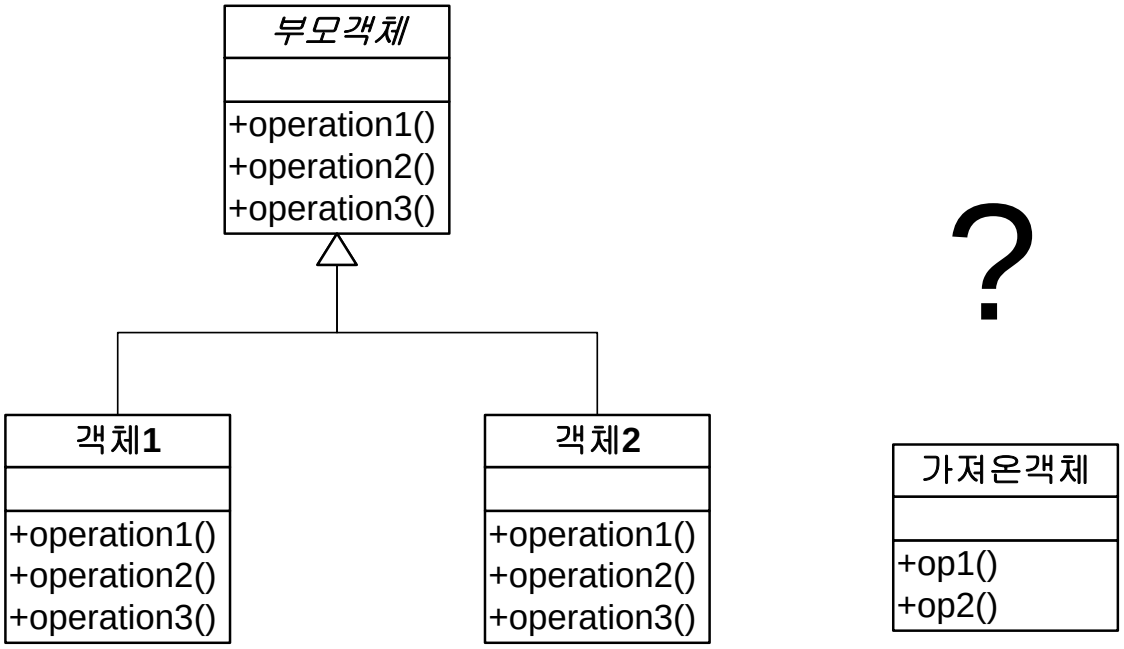
- 위임으로 구현
  - adapter class와 adaptee class 사이에 polymorphism 가능


<<interface>>
MyCollection
+add(int data)
+getIterator() : MyIterator
MyArray
+add(int data)
+getIterator() : MyIterator
MyList
+add(int data)
+getIterator() : MyIterator
MyJavaListAdapter
+add(int data)
+getIterator() : MyIterator
<<interface>>
List
ArrayList
LinkedList

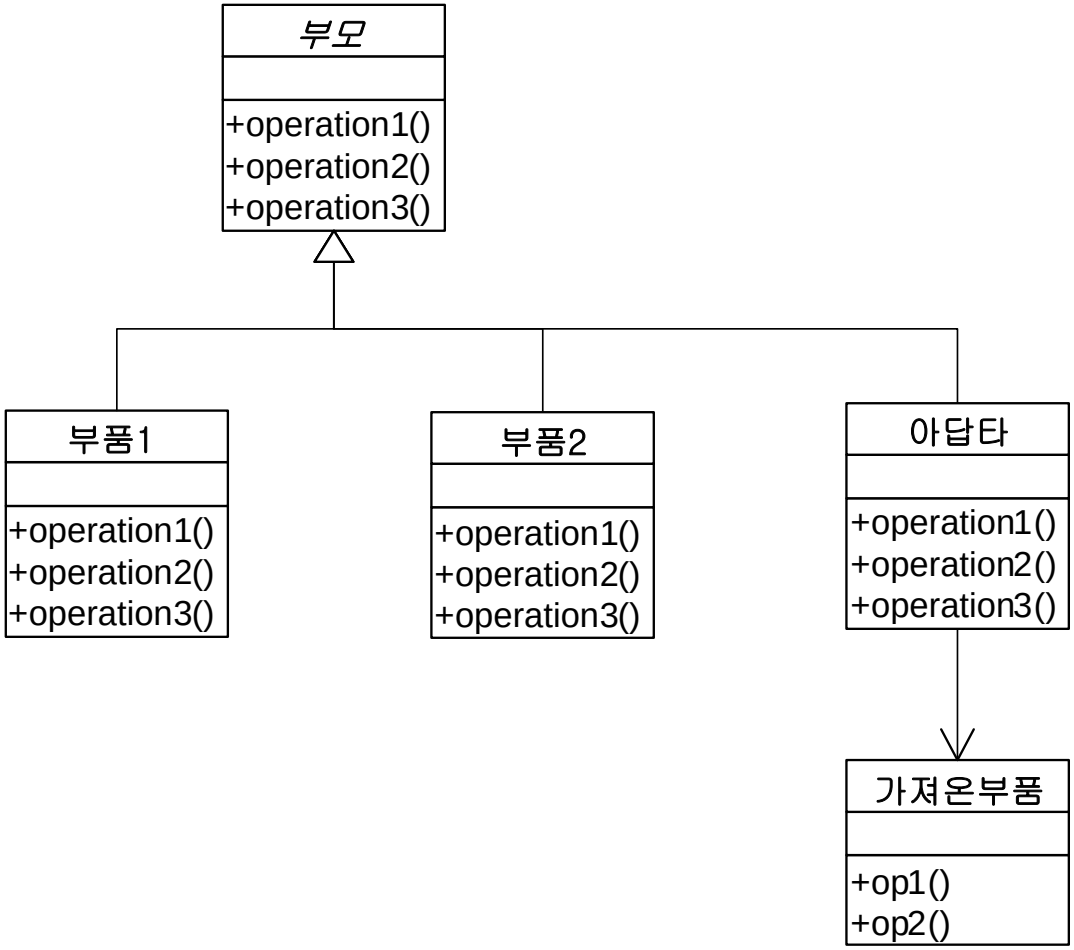
예제 분석

- proxy.e2 예제에 adapter 패턴을 적용
  위임으로 구현 → adapter.e2 예제
  상속으로 구현 → adapter.e3 예제

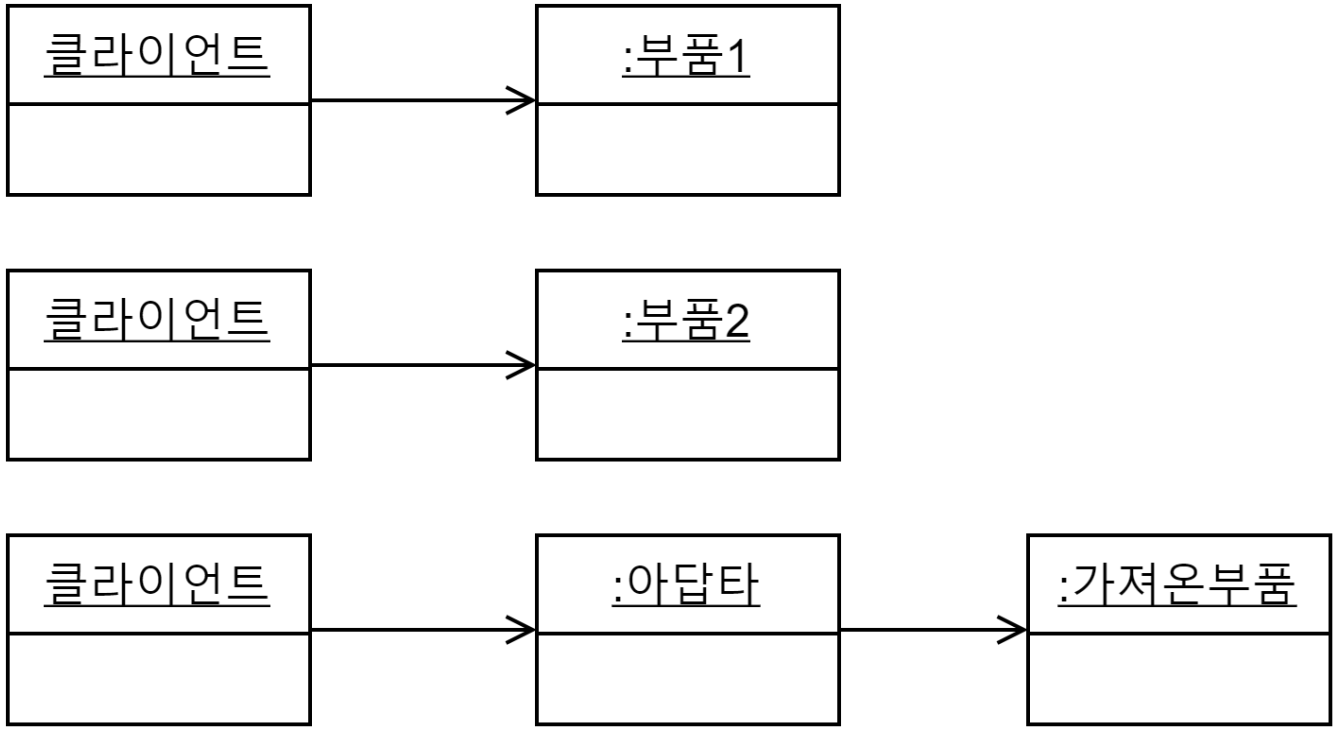
Adapter 요약


부모객체
+operation1()
+operation2()
+operation3()
객체1
+operation1()
+operation2()
+operation3()
객체2
+operation1()
+operation2()
+operation3()
?
가져온객체
+op1()
+op2()

Adapter 요약


부모
+operation1()
+operation2()
+operation3()
부품1
+operation1()
+operation2()
+operation3()
부품2
+operation1()
+operation2()
+operation3()
아답타
+operation1()
+operation2()
+operation3()
가져온부품
+op1()
+op2()

Adapter 요약



Adapter

Observer

Observer – 의도

- 객체의 내부 상태 변경을, 관련된 여러 객체들에 통보
- 객체들간 의존성 최소화
- 상태 변경 통보를 위한 abstraction

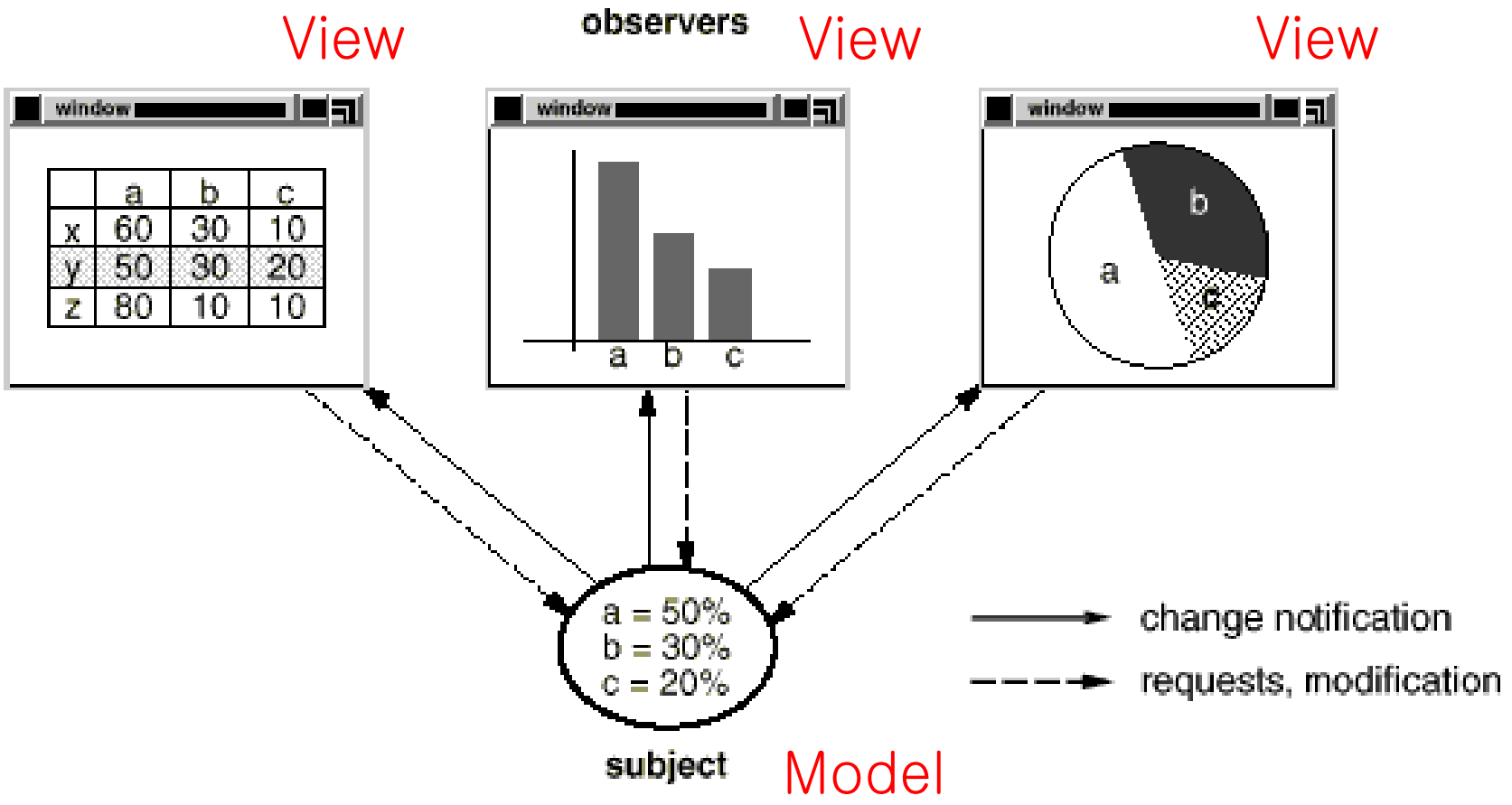
참조 방향 선택

- 서로 참조하는 양방향 참조는 바람직하지 않다

- 바람직한 참조 방향은?
  - observer → observable
  - observable → observer

- 직접 참조하지 않고 통보/호출하는 방법은?

Observer – 의도


View
observers
View
View
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a
b
c
a = 50%
b = 30%
c = 20%
change notification
requests, modification
subject
Model

Observer – 의도

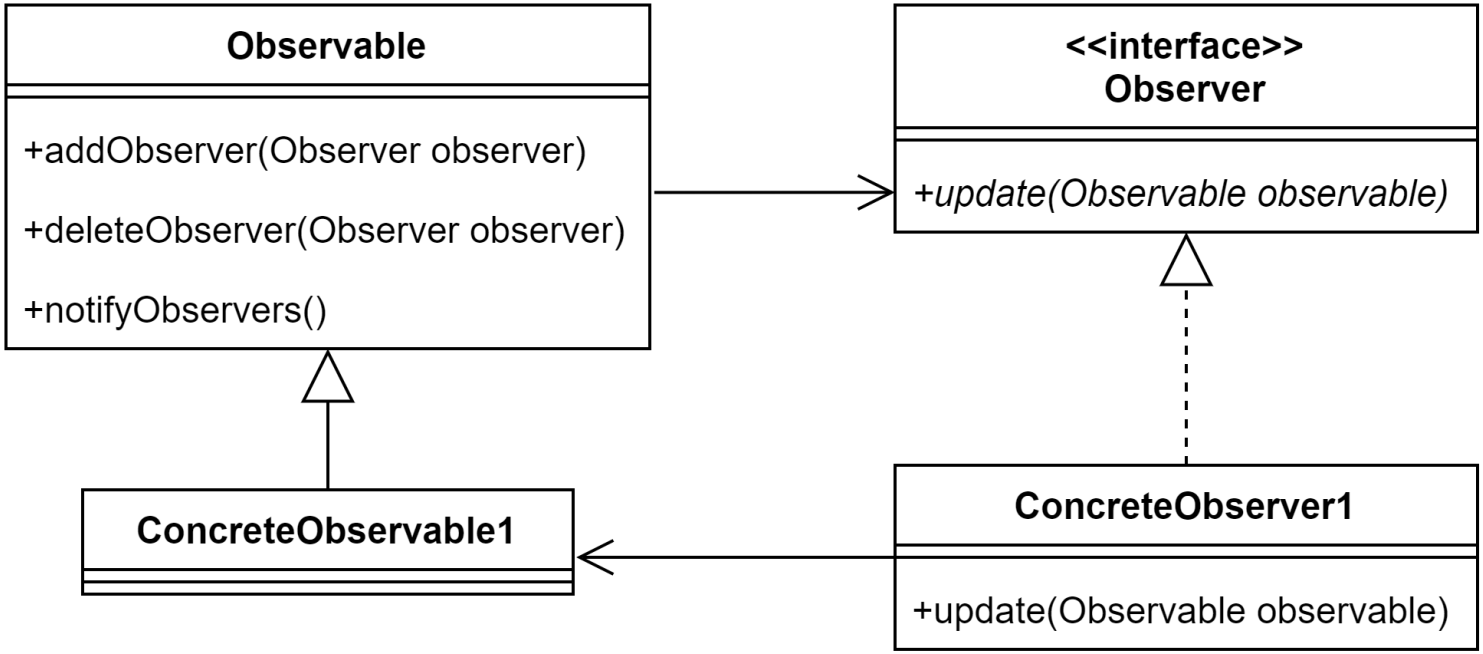
- 참조 방향
  - View → Model

- 통보 방향
  - Model → View

Observer – 의도

- 통보하는 가장 간단한 방법은 메소드를 호출하는 것
- 직접 참조하지 않고 통보하는 방법은?
- → Observer 패턴

Observer – 구조


Observable
+addObserver(Observer observer)
+deleteObserver(Observer observer)
+notifyObservers()
<<interface>>
Observer
+update(Observable observable)
ConcreteObservable1
ConcreteObserver1
+update(Observable observable)

예제 분석

- observer.e1 예제
  - Document와 View 서로 양방향 참조
  - 바람직하지 않은 구조

- observer.e2 예제
  - Observer 패턴 적용

- observer e1 e2 예제 소스 코드를 분석하시오

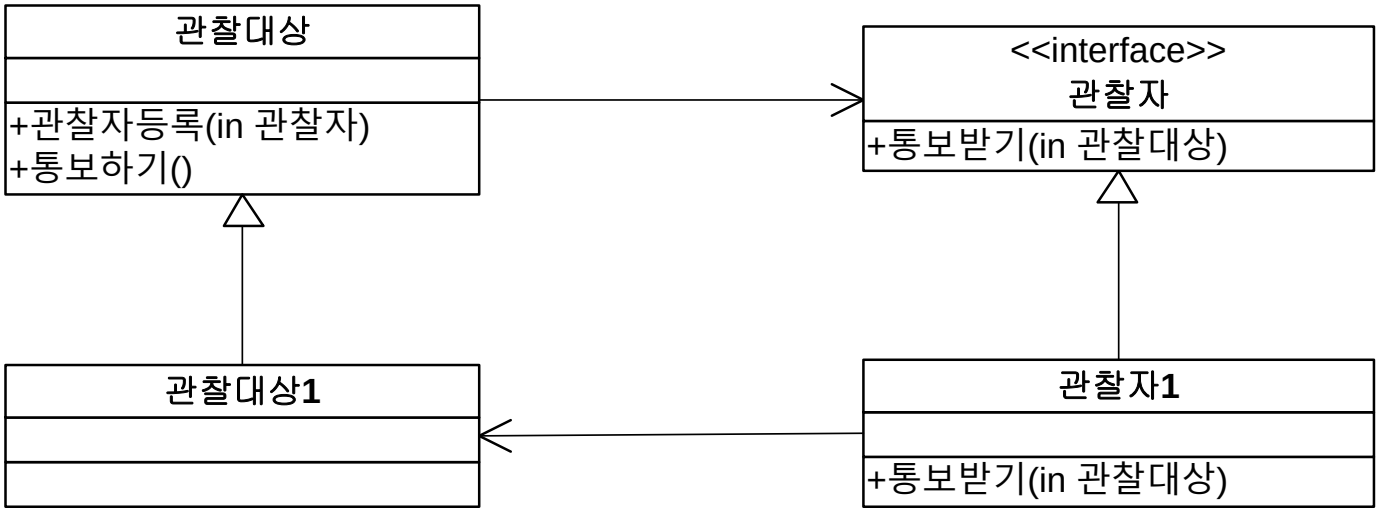
구현 실습

- MyTextView, MyGraphView 클래스에 유사한 코드가 중복된다

- 비슷한 유형의 클래스들에서 중복되는 코드는
  부모 클래스로 추출하여 공유하는 것이 바람직하다

- observer.e2 예제에서
  중복되는 코드를 부모 클래스로 추출하시오.
  (MyView, MyDocument)

Observer 패턴 요약



Mediator

Mediator – 의도

- 여러 객체들간 상호작용을 하나의 객체에 몰아서 구현한다
- 나머지 객체들이 단순해져서 유지보수성/재사용성이 증가한다
- 상호작용이 한 곳에 모아져 있으므로 이해하고 수정하기 좋다

Mediator – 사례

- 파일 선택 대화상자
  - 파일을 선택할 수 있는 리스트 박스
  - 파일 이름을 입력할 수 있는 텍스트 박스
  - 확인 버튼

- 텍스트 박스에 파일 이름을 입력하면
  - 알파벳이 눌러질 때마다
    리스트 박스는 저절로 스크롤 되어 해당 파일이 선택된다
  - 확인 버튼은 활성화된 상태가 된다

Mediator – 사례

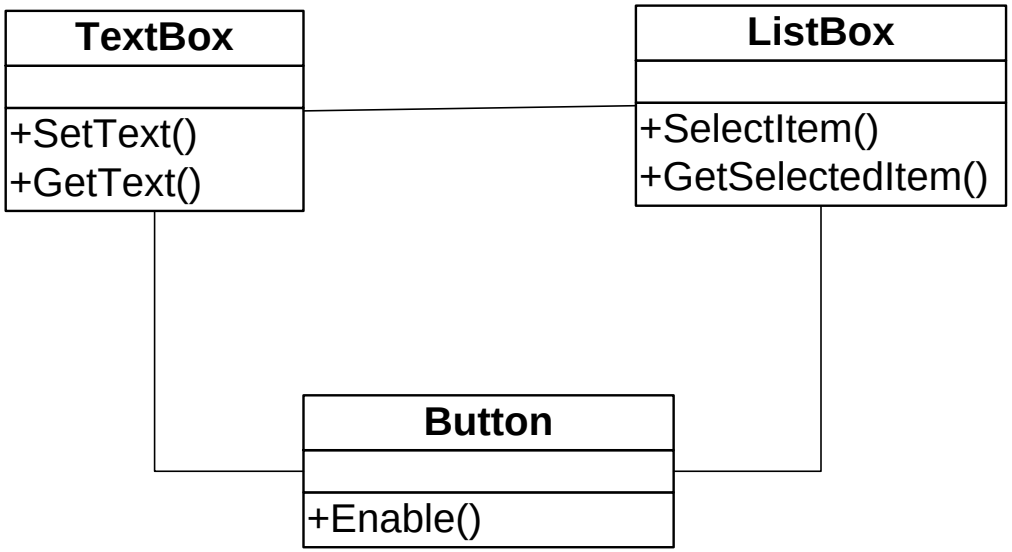
- 리스트 박스에서 파일을 선택하면
  - 텍스트 박스에 그 파일 이름이 저절로 입력된다
  - 확인 버튼은 활성화된 상태가 된다

Mediator – 사례

- 이 기능을 어디에 구현하는 것이 좋을까?

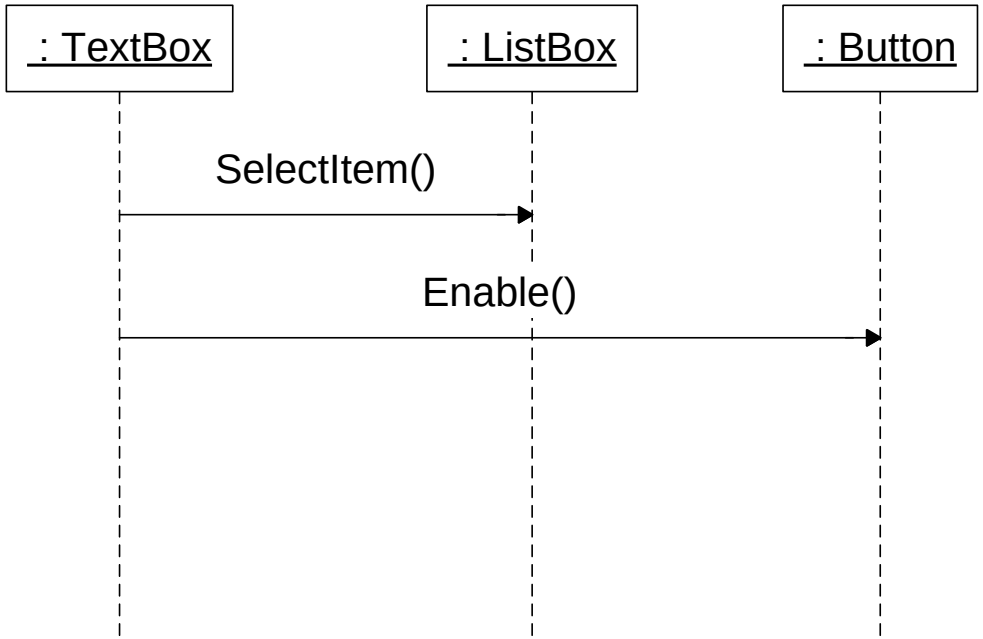
- 텍스트 박스가 직접
  - 리스트 박스의 항목을 선택하고
  - 버튼을 활성화 해야하나?

구조1


TextBox
+SetText()
+GetText()
ListBox
+SelectItem()
+GetSelectedItem()
Button
+Enable()

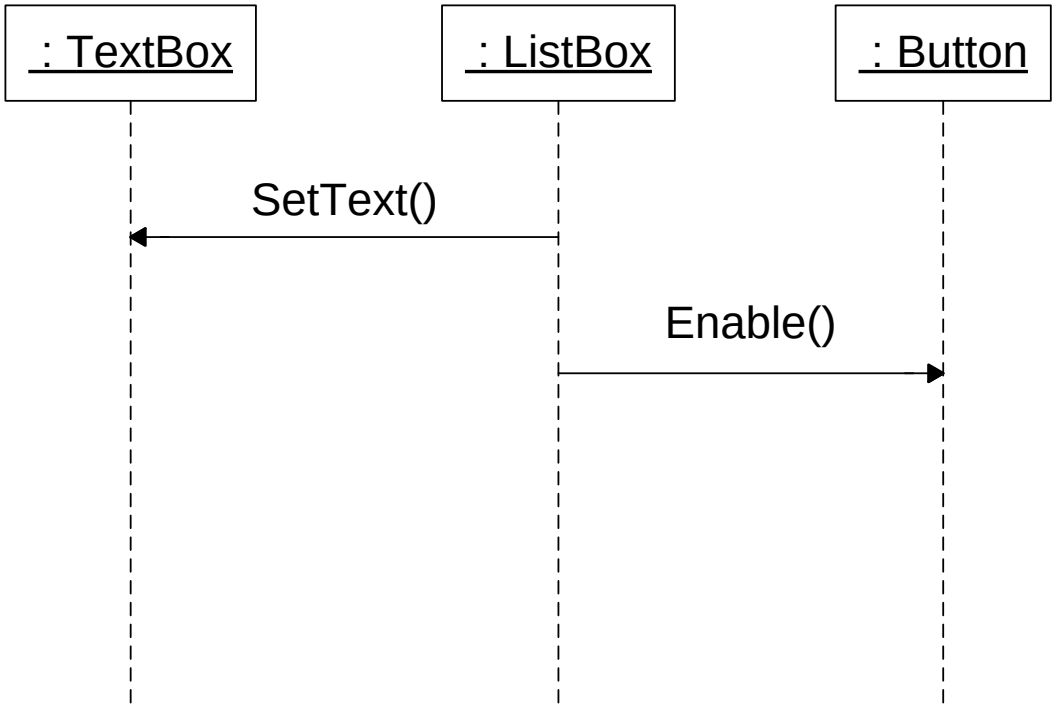
구조1

- 텍스트 박스가 변경된 경우

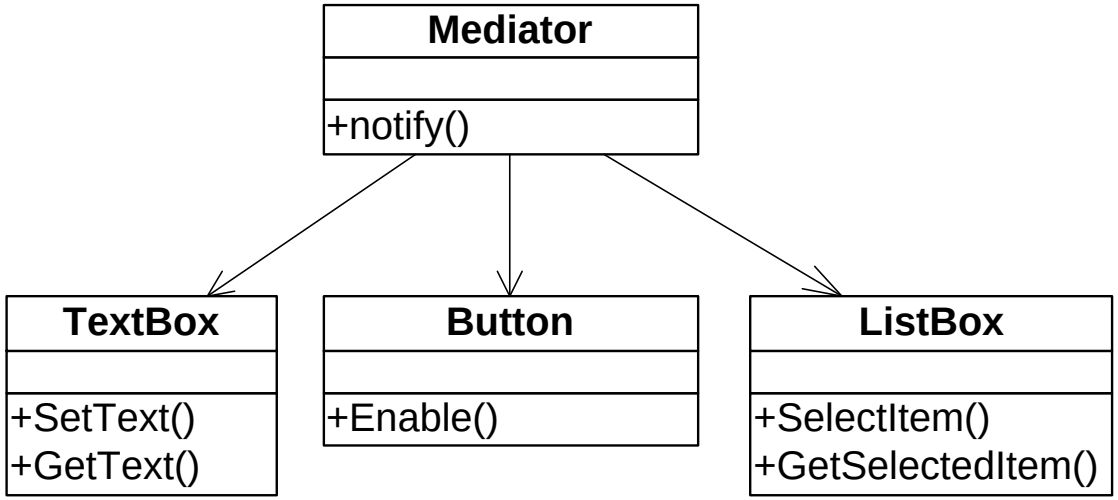


구조1

- 리스트 박스가 변경된 경우

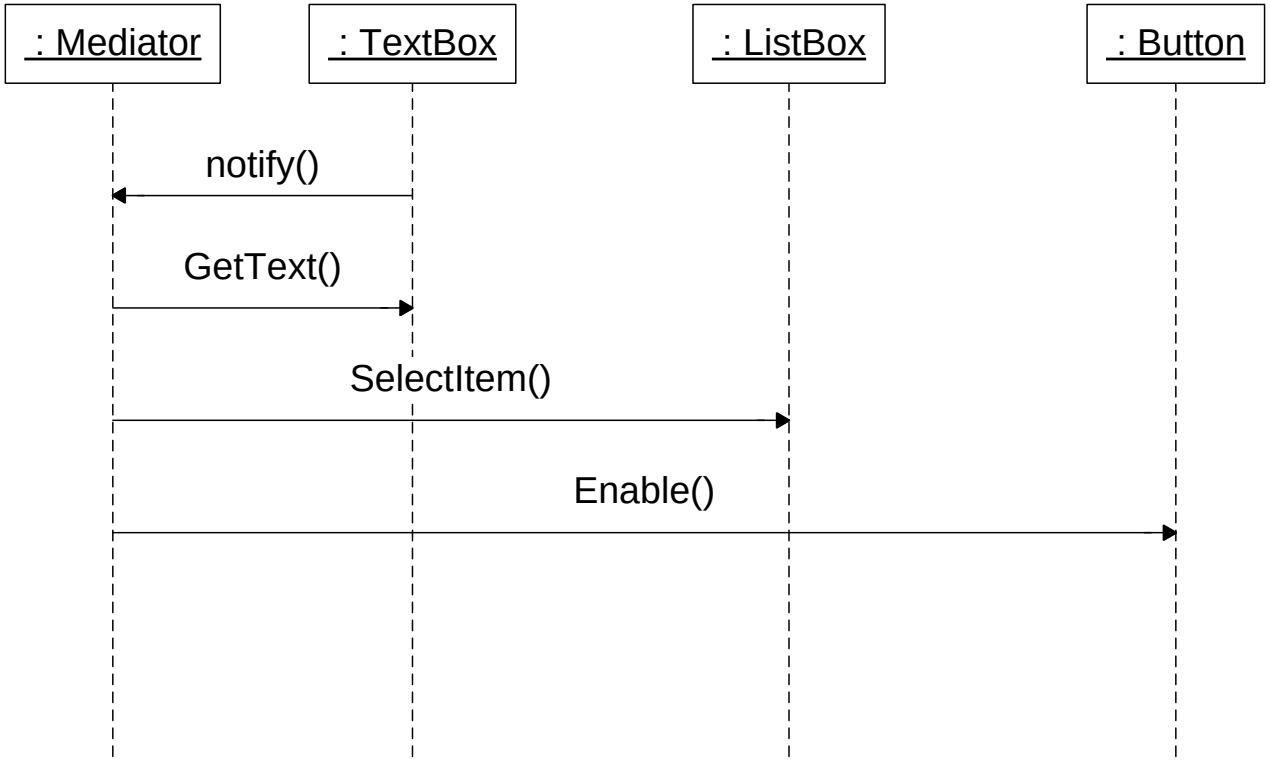


구조2


Mediator
+notify()
TextBox
+SetText()
+GetText()
Button
+Enable()
ListBox
+SelectItem()
+GetSelectedItem()

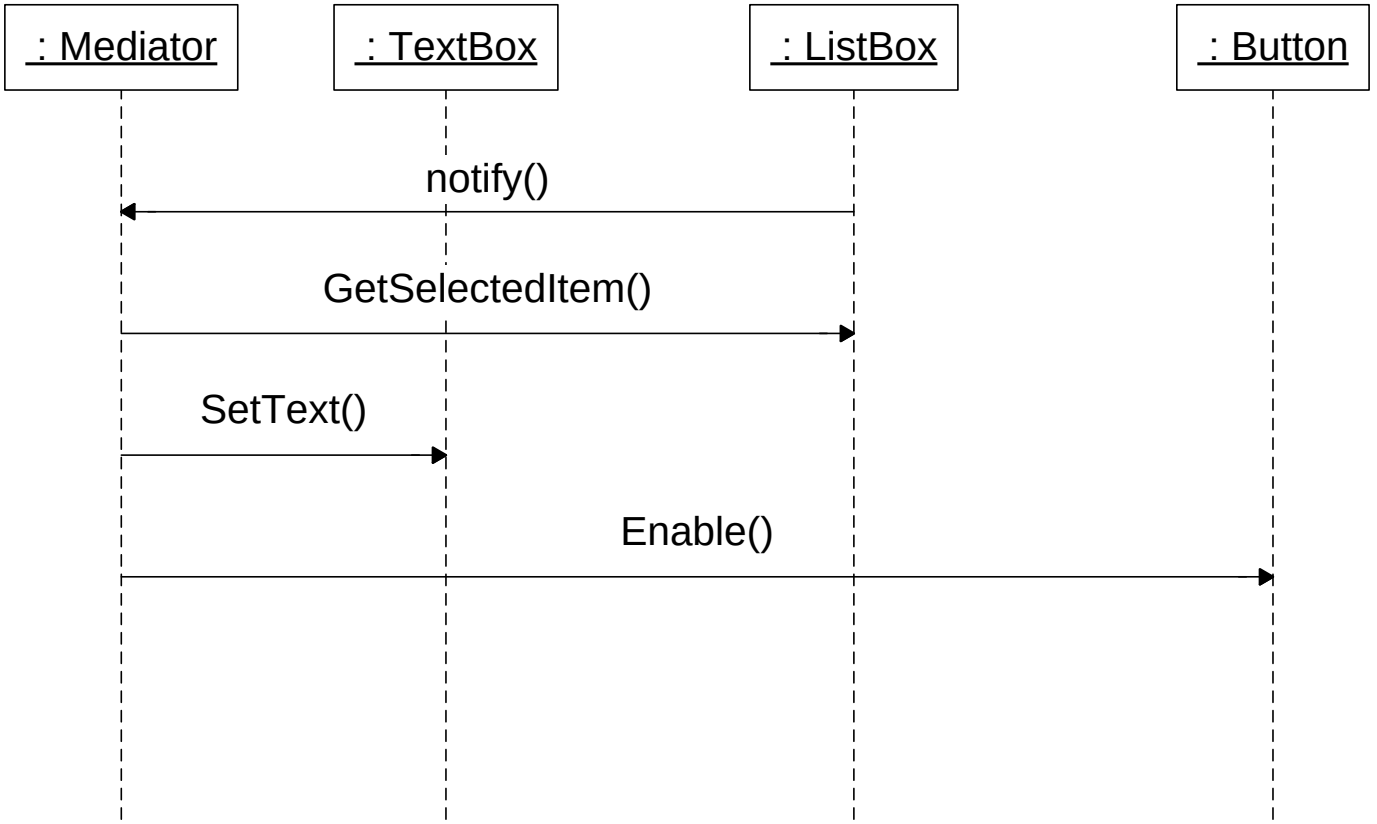
구조2

- 텍스트 박스가 변경된 경우



구조2

- 리스트 박스가 변경된 경우



예제 분석

- 구조1 → mediator.e1
- 구조2 → mediator.e2 (mediator 패턴)
- mediator e1 e2 예제 소스코드를 비교하시오

구현 실습

- TextBox, ListBox, Button 클래스에 유사한 코드가 중복된다

- 비슷한 유형의 클래스들에서 중복되는 코드는
  부모 클래스로 추출하여 공유하는 것이 바람직하다

- mediator.e2 예제에서
  TextBox, ListBox, Button 클래스에
  중복되는 코드를 부모 클래스로 추출하시오.
  → Widget
  (모든 mediator들의 부모 클래스인 Mediator 클래스도 필요)

- template method 패턴 적용

구현 실습

- mediator.e2 예제에 observer 패턴 적용
  - TextBox, ListBox, Button 상태가 변경되면 mediator에 통보가 되고 (childChanged 메소드) 상호작용이 시작되어야 한다.
  - TextBox, ListBox, Button이 Mediator를 참조하지 않고 상태 변화 통보 가능 (observer 패턴)

Visitor

Visitor – 의도

- 복잡한 자료구조의 탐색을 구현

- 자료구조 탐색과 작업을 분리해서 구현

Visitor – 의도

- 컴파일러 내부에는 복잡한 자료구조가 있다

- abstract syntax tree

- nodes of an abstract syntax tree
  - assignment statements
  - variable accesses
  - arithmetic expressions…

Visitor – 의도

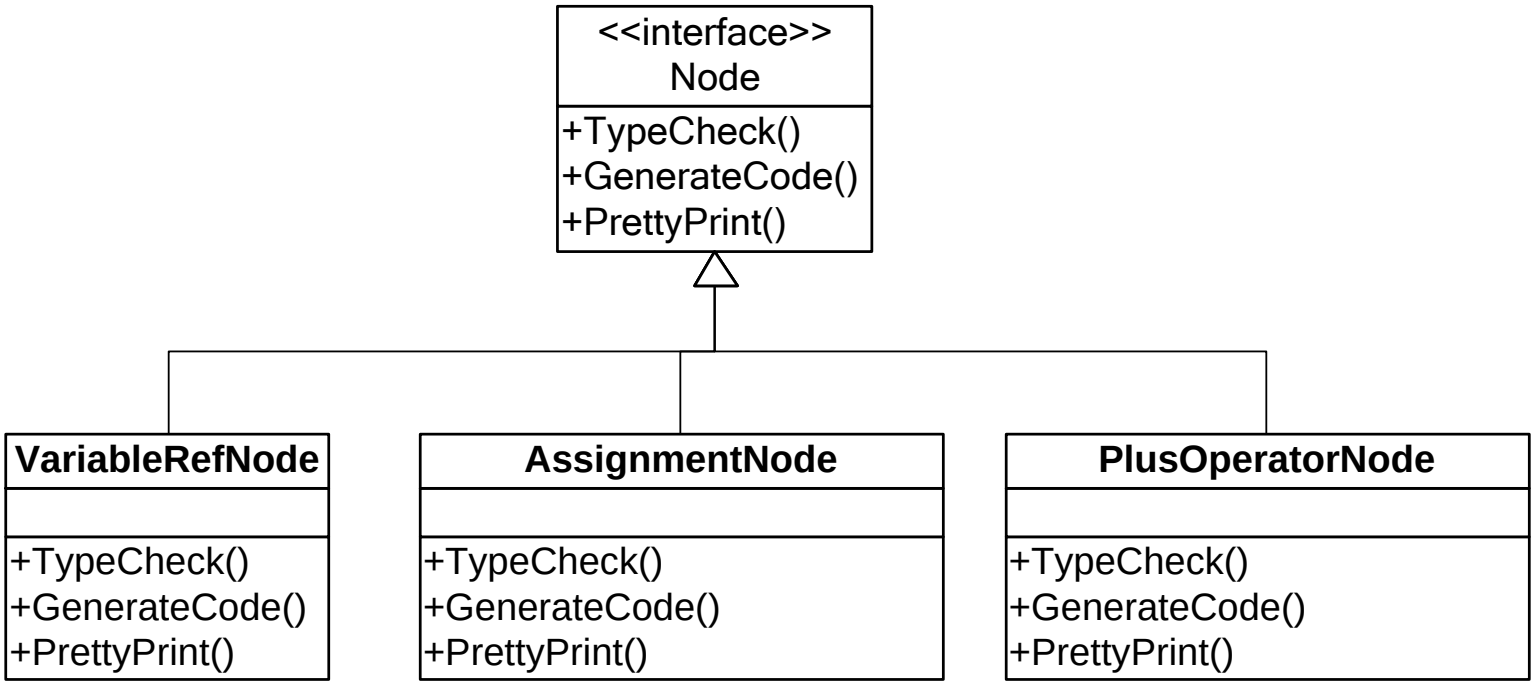
- abstract syntax tree 의 각 노드를 탐색하며 여러 작업을 해야 한다

- operations on abstract syntax tree
  - type-checking
  - code-generation
  - pretty-printing…

- 위 작업들을 어떻게 구현할 것인가?

Visitor – 구현방식1

- 다음과 같이 구현하는 것은 어떤가?



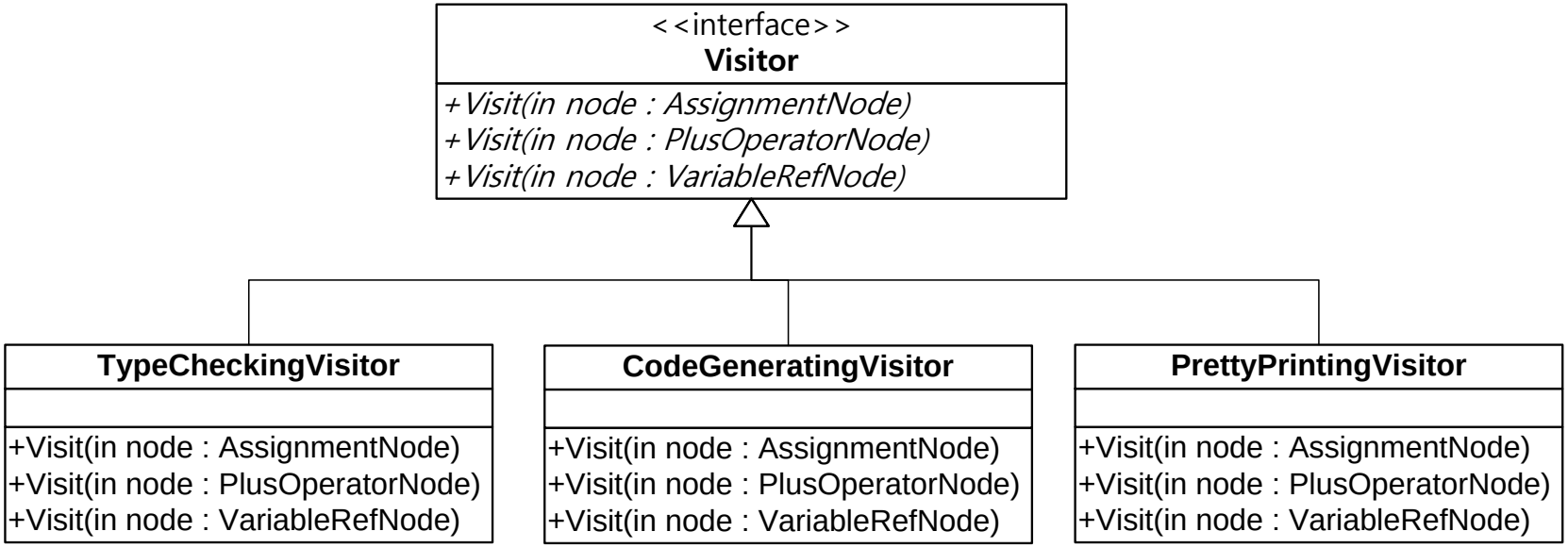
Visitor – 의도

- 앞 슬라이드의 구현은 바람직하지 않다
- 새 작업을 추가하려면 모든 노드 클래스를 수정해야 한다
- 작업이 여러 노드에 조금씩 분산되어서, 이해하고 수정하기 어렵다

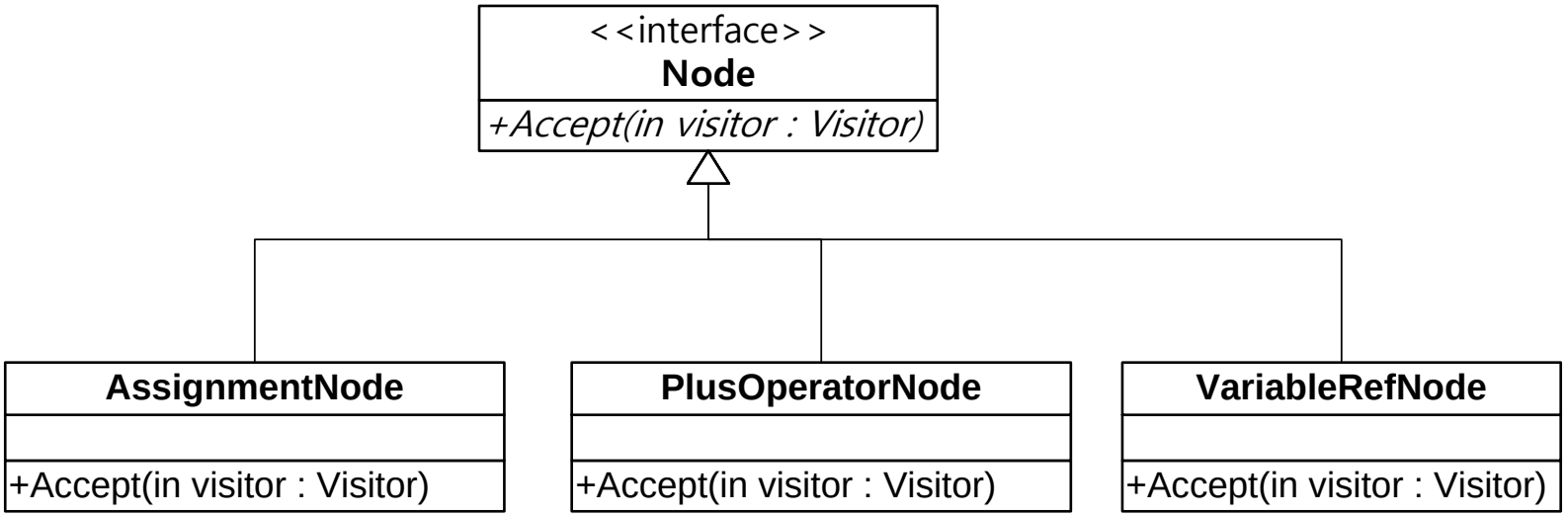
Visitor – 해결

- 작업을 유지보수 해야 한다면, 자료구조로부터 작업을 분리
- 작업 클래스 → Visitor 클래스
- 각 작업마다 하나의 Visitor 클래스가 만들어진다.
  - TypeCheckingVisitor
  - CodeGeneratingVisitor
  - PrettyPrintingVisitor

작업 클래스


<<interface>>
Visitor
+Visit(in node : AssignmentNode)
+Visit(in node : PlusOperatorNode)
+Visit(in node : VariableRefNode)
TypeCheckingVisitor
+Visit(in node : AssignmentNode)
+Visit(in node : PlusOperatorNode)
+Visit(in node : VariableRefNode)
CodeGeneratingVisitor
+Visit(in node : AssignmentNode)
+Visit(in node : PlusOperatorNode)
+Visit(in node : VariableRefNode)
PrettyPrintingVisitor
+Visit(in node : AssignmentNode)
+Visit(in node : PlusOperatorNode)
+Visit(in node : VariableRefNode)

자료구조 클래스

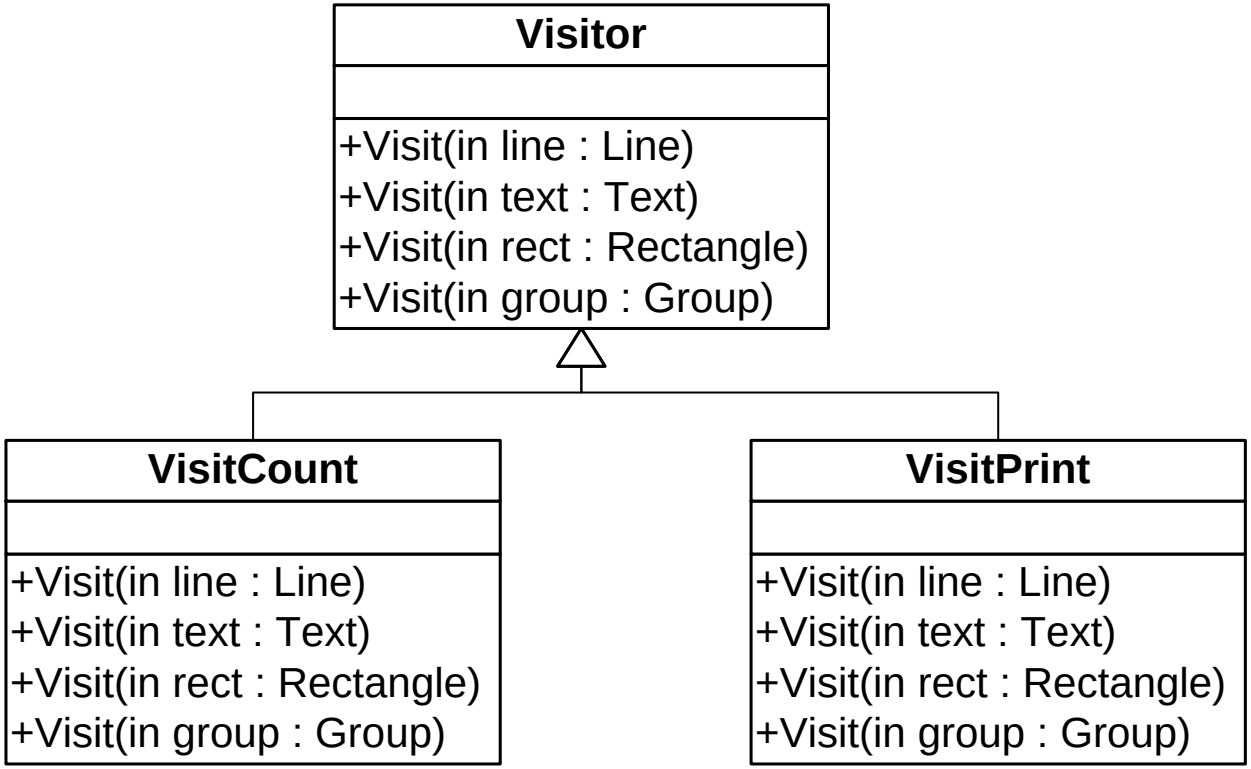

<<interface>>
Node
+Accept(in visitor : Visitor)
AssignmentNode
+Accept(in visitor : Visitor)
PlusOperatorNode
+Accept(in visitor : Visitor)
VariableRefNode
+Accept(in visitor : Visitor)

Visitor – 결과

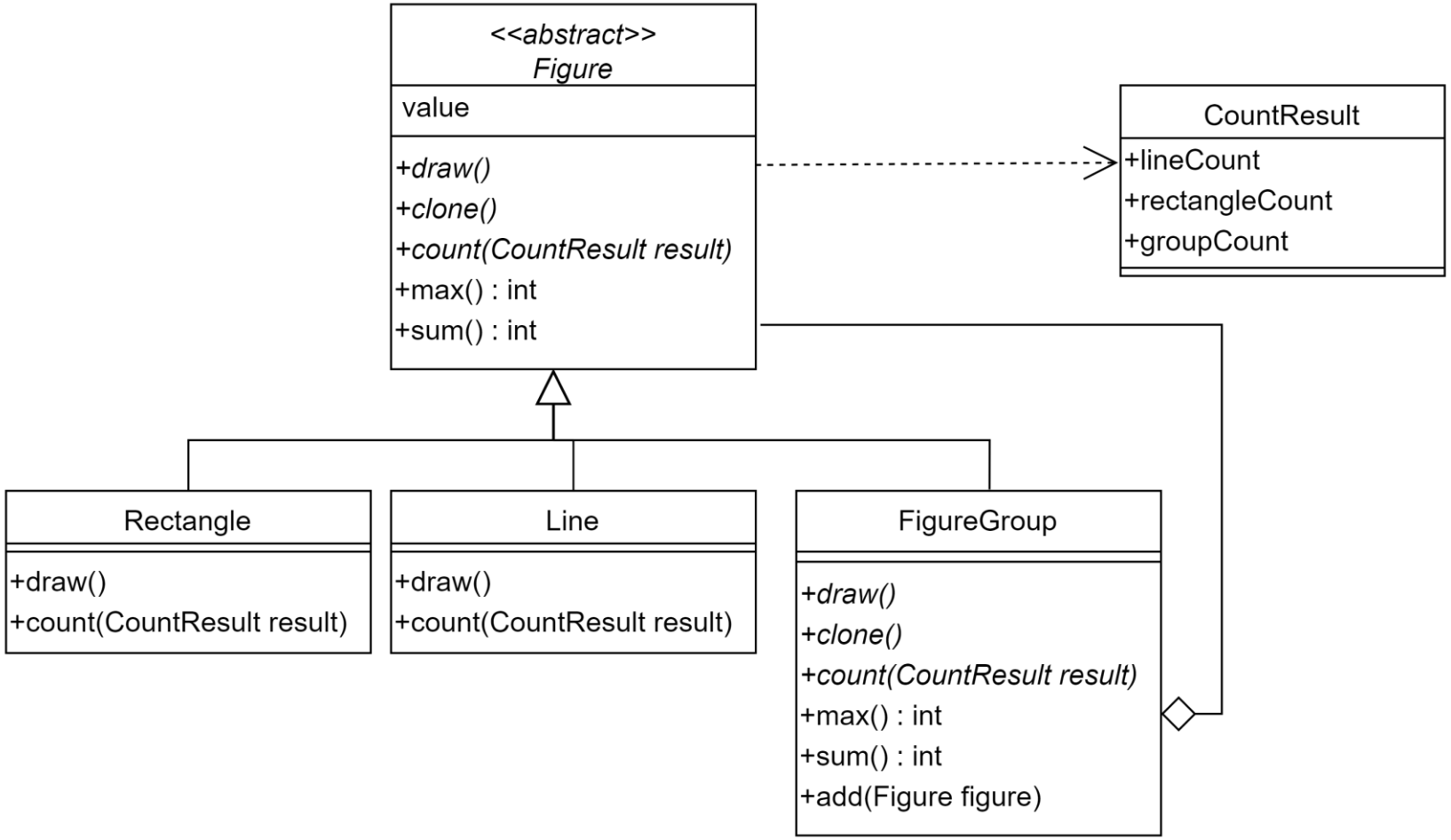
- 새 작업을 구현하려면?
  - Visitor 클래스만 추가 구현

- 새 노드가 추가되면?
  - 모든 Visitor 클래스가 수정되어야 함

Visitor – 사례


Visitor
+Visit(in line : Line)
+Visit(in text : Text)
+Visit(in rect : Rectangle)
+Visit(in group : Group)
VisitCount
+Visit(in line : Line)
+Visit(in text : Text)
+Visit(in rect : Rectangle)
+Visit(in group : Group)
VisitPrint
+Visit(in line : Line)
+Visit(in text : Text)
+Visit(in rect : Rectangle)
+Visit(in group : Group)

예제 코드 분석 e1

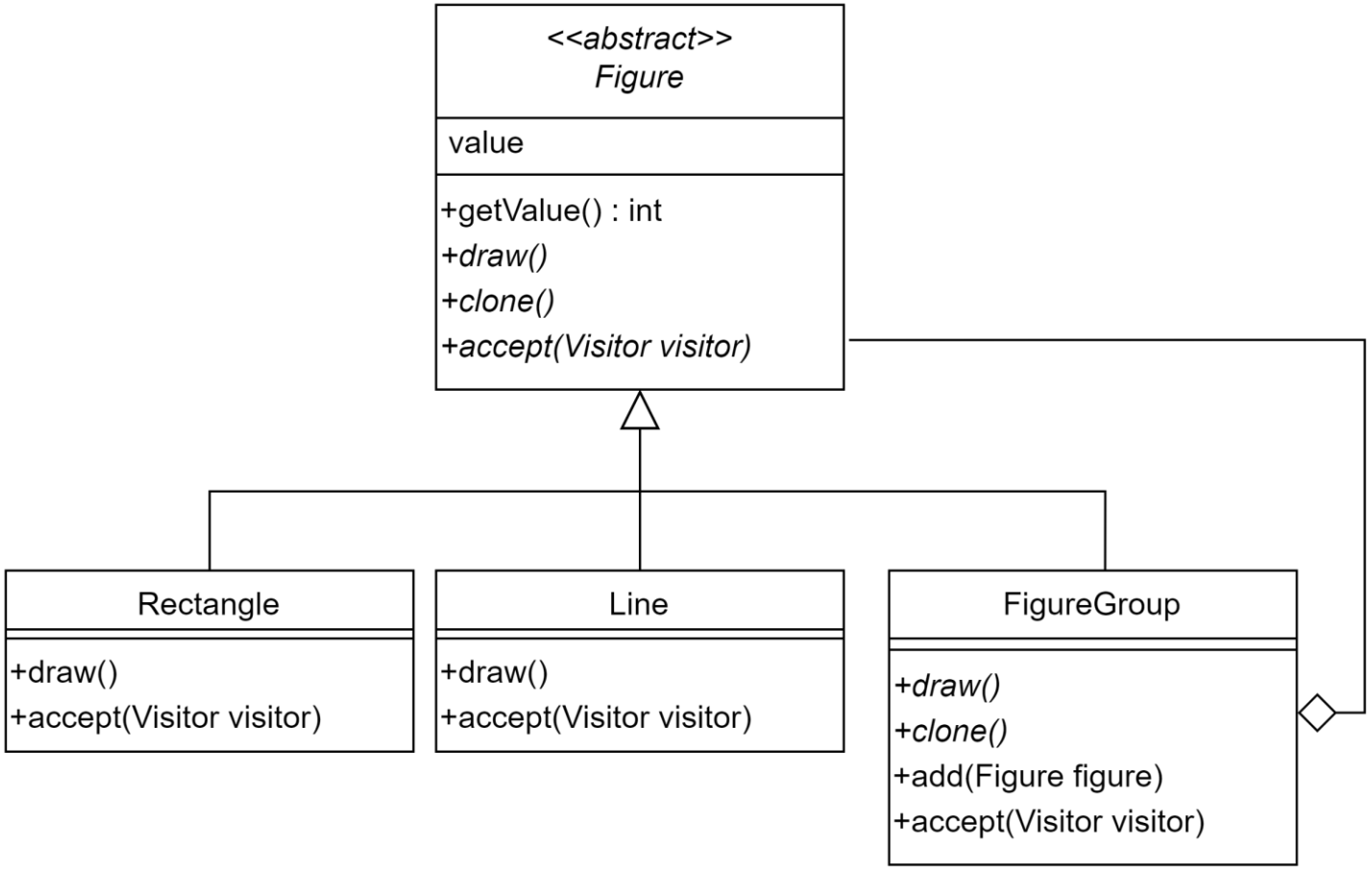

<<abstract>>
Figure
value
+draw()
+clone()
+count(CountResult result)
+max() : int
+sum() : int
CountResult
+lineCount
+rectangleCount
+groupCount
Rectangle
+draw()
+count(CountResult result)
Line
+draw()
+count(CountResult result)
FigureGroup
+draw()
+clone()
+count(CountResult result)
+max() : int
+sum() : int
+add(Figure figure)

예제 코드 분석 e1

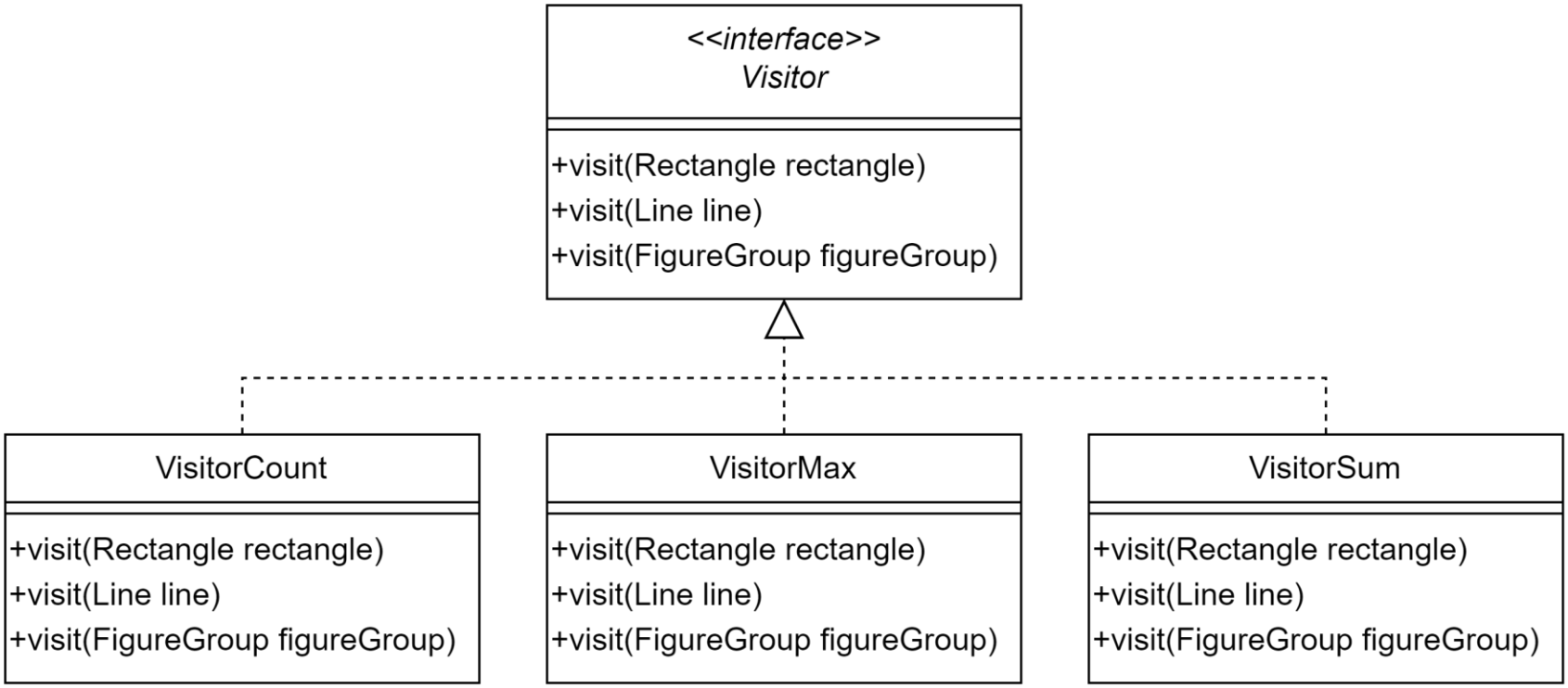
- 도형(Figure)에는 int value 가 있다.
- composite 패턴이므로 도형의 트리 구조가 생성된다

- visitor.e1 예제는 아래 메소드들을 도형 클래스에 구현함
  - max 메소드는 도형 트리에서 value의 최대값을 구한다
  - sum 메소드는 도형 트리에서 value 합계를 구한다.
  - count 메소드는 도형의 수를 센다
    그 결과는 CountResult 객체에 저장한다

예제 코드 분석 e2


<<abstract>>
Figure
value
+getValue() : int
+draw()
+clone()
+accept(Visitor visitor)
Rectangle
+draw()
+accept(Visitor visitor)
Line
+draw()
+accept(Visitor visitor)
FigureGroup
+draw()
+clone()
+add(Figure figure)
+accept(Visitor visitor)

예제 코드 분석 e2


<<interface>>
Visitor
+visit(Rectangle rectangle)
+visit(Line line)
+visit(FigureGroup figureGroup)
VisitorCount
+visit(Rectangle rectangle)
+visit(Line line)
+visit(FigureGroup figureGroup)
VisitorMax
+visit(Rectangle rectangle)
+visit(Line line)
+visit(FigureGroup figureGroup)
VisitorSum
+visit(Rectangle rectangle)
+visit(Line line)
+visit(FigureGroup figureGroup)

예제 코드 분석

- visitor.e2
  - visitor.e1 예제에 visitor 패턴을 적용함

구현 실습:

- 이진트리(binary tree)에 Visitor 패턴 구현
  - Binary Tree와 Node에 accept 메소드를 구현해야 함
  - accept 메소드에서 Visitor의 visit 메소드를 호출해야함
  - 그리고 자식 노드들의 accept 메소드를 재귀적 호출해야함
  - Node 클래스가 한 개 뿐이므로 visit 메소드도 한 개

- visitor.e3 예제에 Visitor 패턴을 구현하시오
  - PrintVisitor: 이진트리에 저장된 정수 출력 작업
  - SumVisitor: 이진트리에 저장된 정수 합계 작업

구현실습 - group sum 계산 작업

- visitor.e1 예제와 visitor.e2 예제에 다음 작업을 구현해 보자

- 계산도형 클래스 그룹의 value 합계 구하기 (group sum)
  - Rectangle 객체들의 value 합계
  - Line 객체들의 value 합계
  - FigureGroup 객체들의 value 합계

- 예제의 구조만 이해하면, 구현은 매우 간단함
  visitor.e1에서 구현은 CountResult 구현을 참고
  - visitor.e2에서 구현은 VisitorCount 구현을 참고

- 이 group sum 작업 구현을 기준으로
  visitor.e1와 visitor.e2의 유지보수성을 평가해 보자.