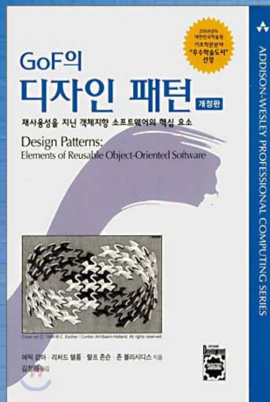
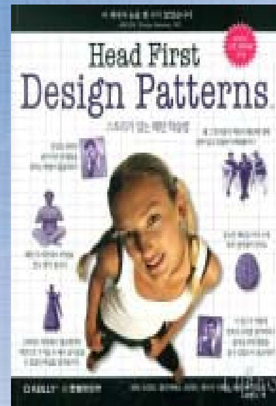
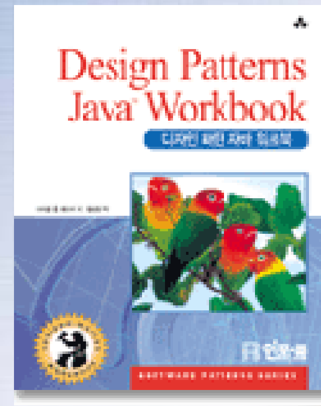
LECTURE

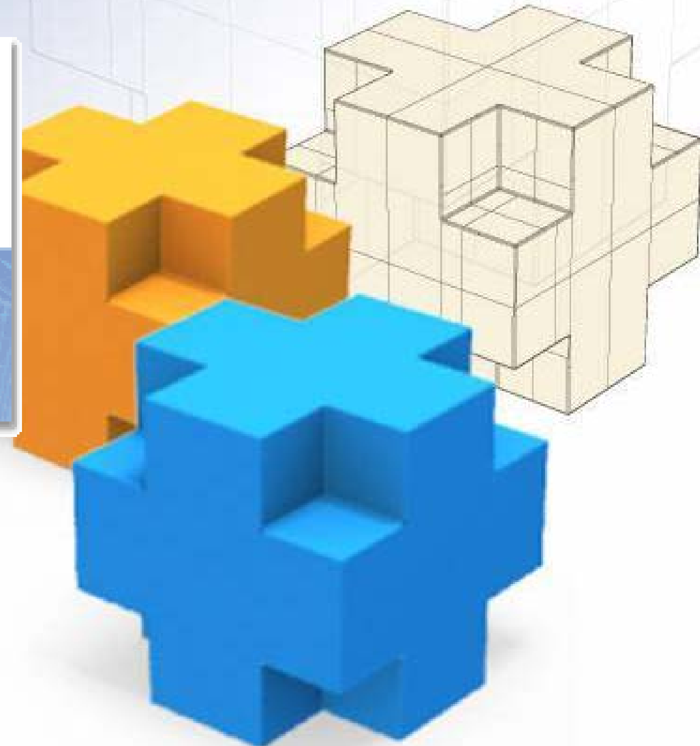
11

디자인 패턴









최은만, CSE 4039 소프트웨어 공학

설계 작업에 대한 도전

- 소프트웨어 설계는 어려운 일
  - 문제를 잘 분할하고
  - 유연하고 잘 모듈화 된 우아한 디자인이 되어야 함
- 설계는 시행 착오(trial and error)의 결과
- 성공적인 설계가 존재
  - 두 설계가 똑 같은 일은 없음
  - 반복되는 특성

디자인 패턴

*디자인 패턴*은 공통된 소프트웨어 문제에 오래 동안 사용되어 검증된 솔루션

- 디자인 작업에 사용되는 공통 언어. 의사소통을 향상 시키며 구현, 문서화에 도움
- 패턴은 전문 기술을 담고 있고 그것을 전수시킬 수 있음

참고 자료

- 디자인 패턴 참고 웹 사이트
  - http://sourcemaking.com/design_patterns

Gangs of Four(GoF) 패턴

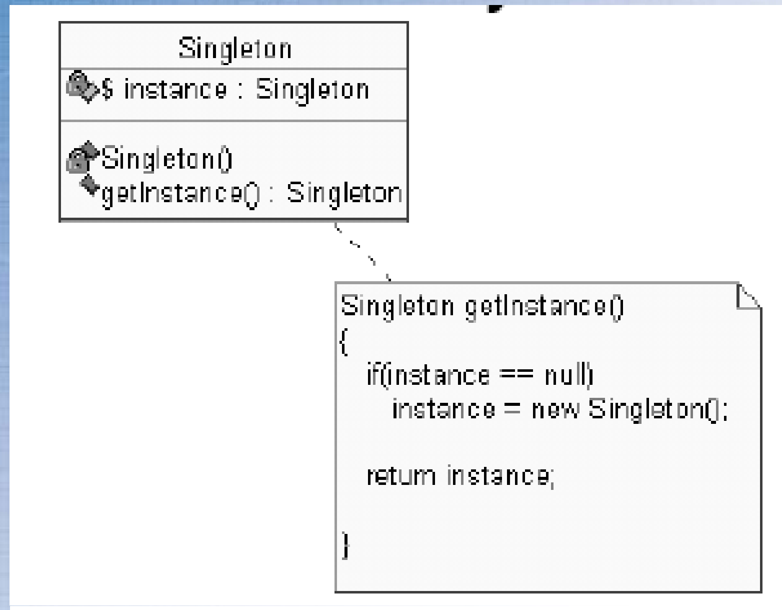
- 생성 패턴(추상 객체 인스턴스화)
  - 추상 팩토리, 팩토리, 빌더, 프로토타입, 싱글톤
- 구조 패턴(객체 결합)
  - 어뎁터, 브리지, 컴포지트, 데코레이터, 퍼싸드, 플라이웨이트, 프록시
- 행위 패턴(객체 간 커뮤니케이션)
  - 책임 체인, 커맨드, 인터프리터, 반복자(iterator), 중재자, 메멘토, 옵서버, 상태, 전략(stretegy), 템플릿 메소드, 비지터

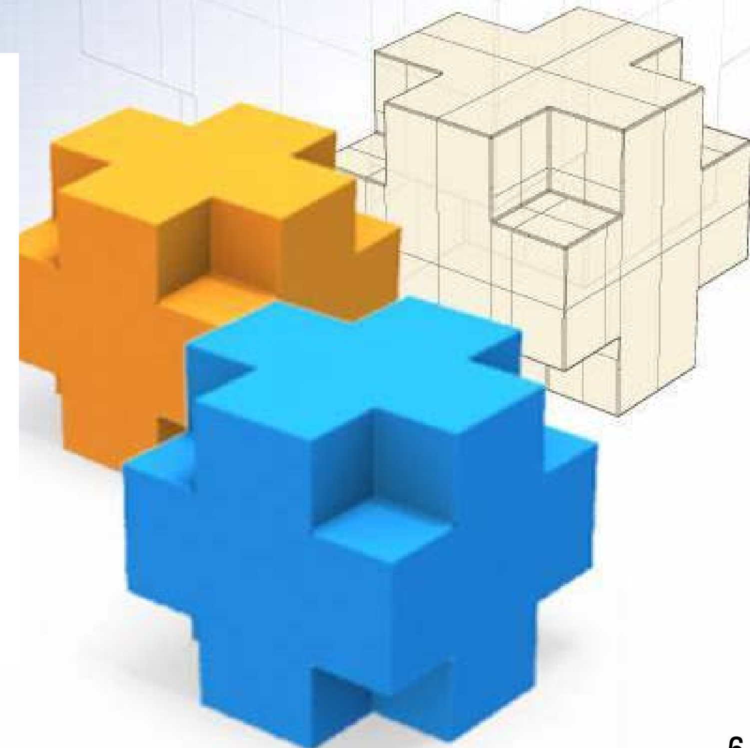
LECTURE

11

싱글톤 패턴

클래스가 하나의 객체만을 생성





객체의 생성을 제한

- 문제: 클래스의 객체를 하나만 만들어야 하는 경우가 있음
  - 두 개 이상 만들면 안되게 하고 싶을 때
  - 예: 키보드 리더, 프린터 스풀러, 점수기록표

- 고려해야 하는 이유
  - 객체를 많이 만드는 것은 시간이 많이 걸림
  - 잉여 객체는 메모리 낭비
  - 여러 가지 다른 객체가 메모리에 떠 도는 것은 유지보관에 골치 덩어리

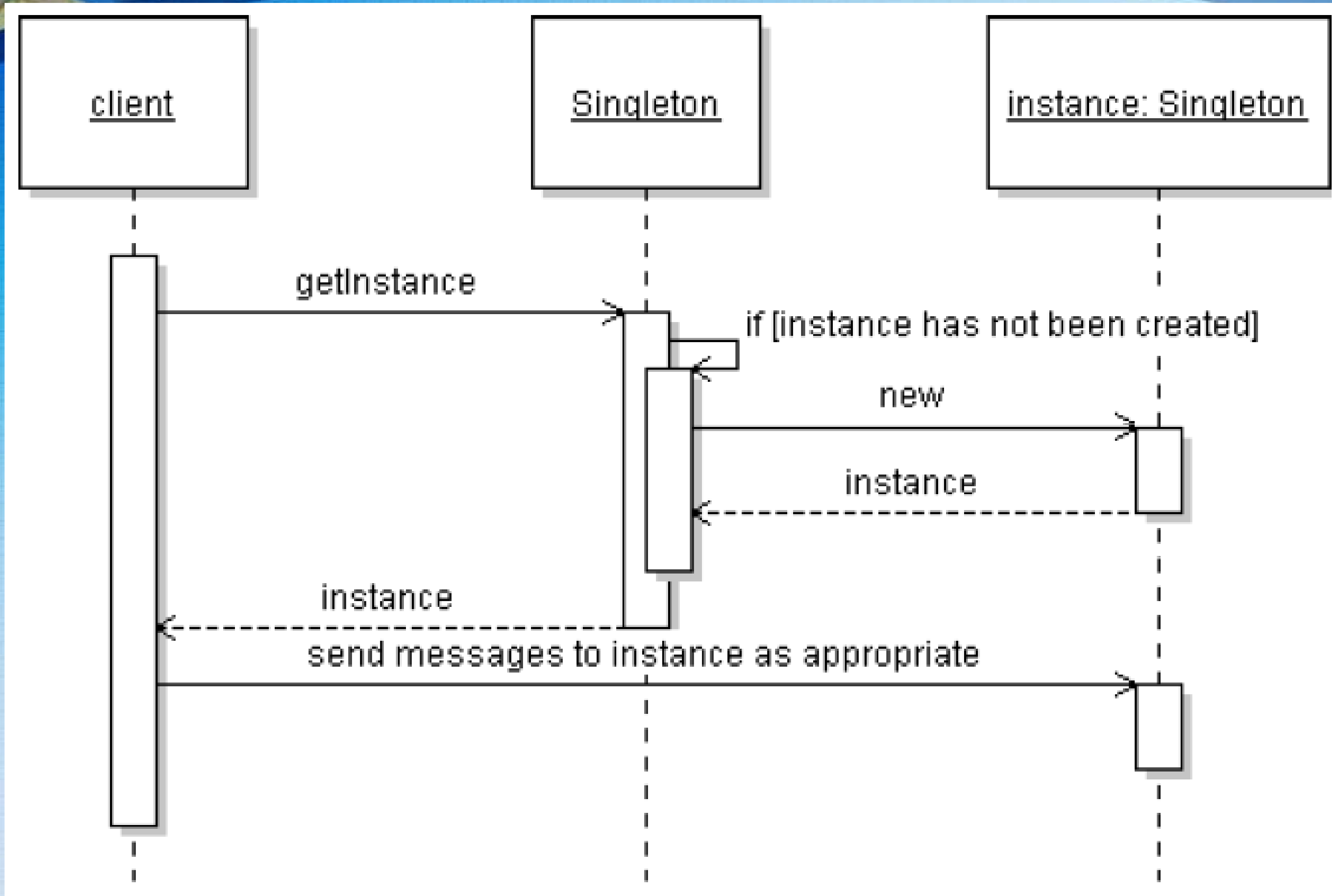
싱글톤 패턴

- 싱글톤: 그 타입의 유일한 객체
  - 많아야 하나의 인스턴스를 가짐을 보장
  - 인스턴스에 대하여 어디서든 접근할 수 있게 하여야
  - 프로그래머가 인스턴스를 없애버리는(또는 더 생성할) 관리 책무는 빼앗음
  - 사용자에게 유일한 인스턴스를 접근할 수 있는 메소드를 제공
  - 많이 알려진 디자인 패턴 중의 하나

싱글톤 패턴의 구현

- 생성자를 밖에서 부르지 못하도록 private으로 만든다.
- 클래스 안에 클래스의 인스턴스를 static private 으로 선언
- 단일 인스턴스를 접근할 수 있는 public getInstance() 나 유사 메소드를 둔다.
  - 이 메소드는 다중 스레드로도 실행될 수 있기 때문에 보호되어야 하고 동기화 되어야

싱글톤 순서 다이어그램



싱글톤 예

- 난수를 만들어 내는 RandomGenerator를 싱글톤으로 만들어 보자

```
public class RandomGenerator {
    private static RandomGenerator gen = new
            RandomGenerator();
    public static RandomGenerator getInstance() {
            return gen;
    }
    private RandomGenerator() {}
    ...
}
```

이 프로그램의 문제점은?

싱글톤 사례 2

- 필요할 때까지는 객체를 만들지 않는다.

```
public class RandomGenerator {
    private static RandomGenerator gen = null;
    public static RandomGenerator getInstance() {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }
    ...
}
```

이 버전의 문제점은?

싱글톤 사례 3

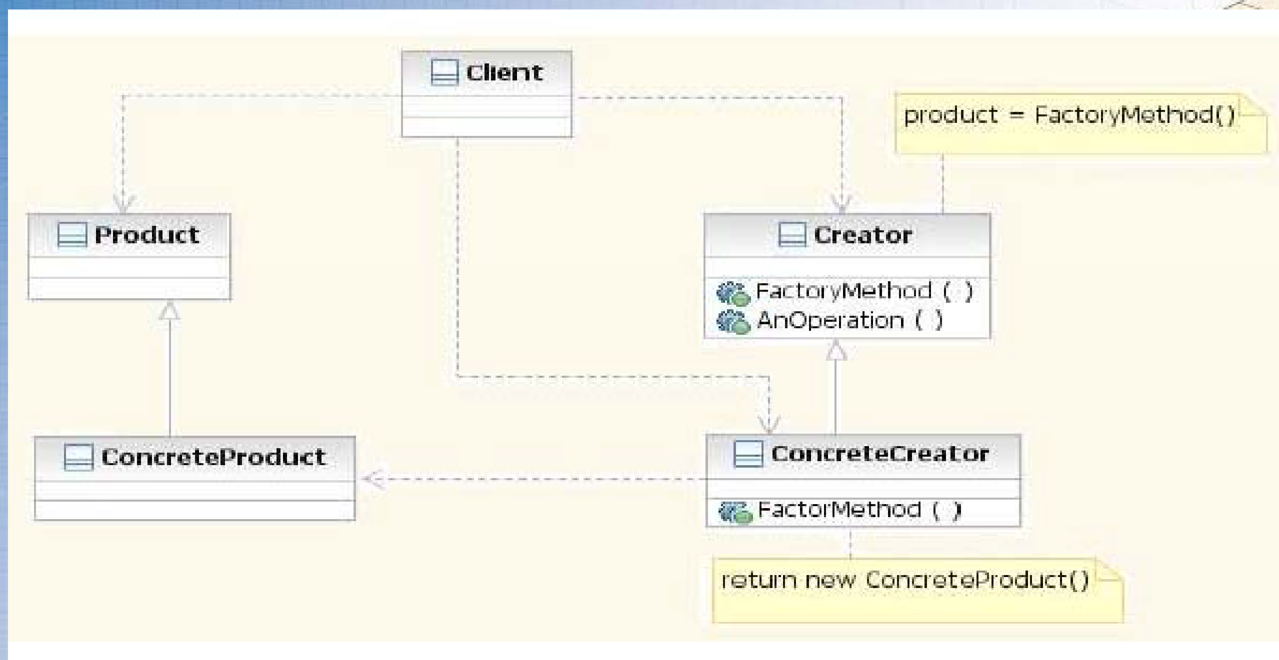
- Locking으로 병렬처리의 문제점 해결

```
public class RandomGenerator {
        private static RandomGenerator gen = null;
        public static synchronized RandomGenerator
                                                getInstance() {
            if (gen == null) {
                gen = new RandomGenerator();
            }
            return gen;
        }
        ...
}
```

LECTURE

11 팩토리 패턴 (팩토리 메소드, 추상 팩토리)

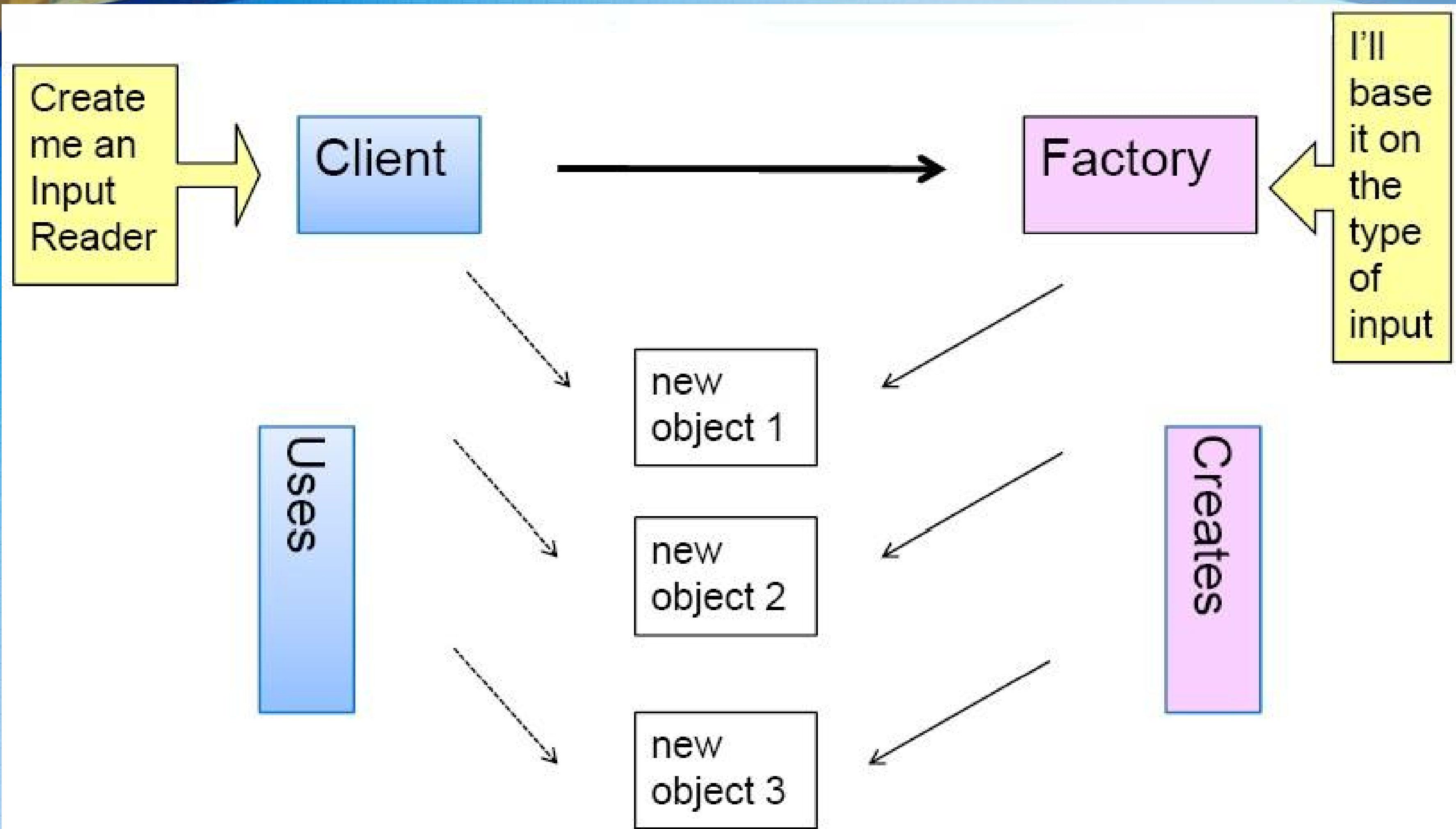
객체를 쉽게 생성하는 클래스나 메소드



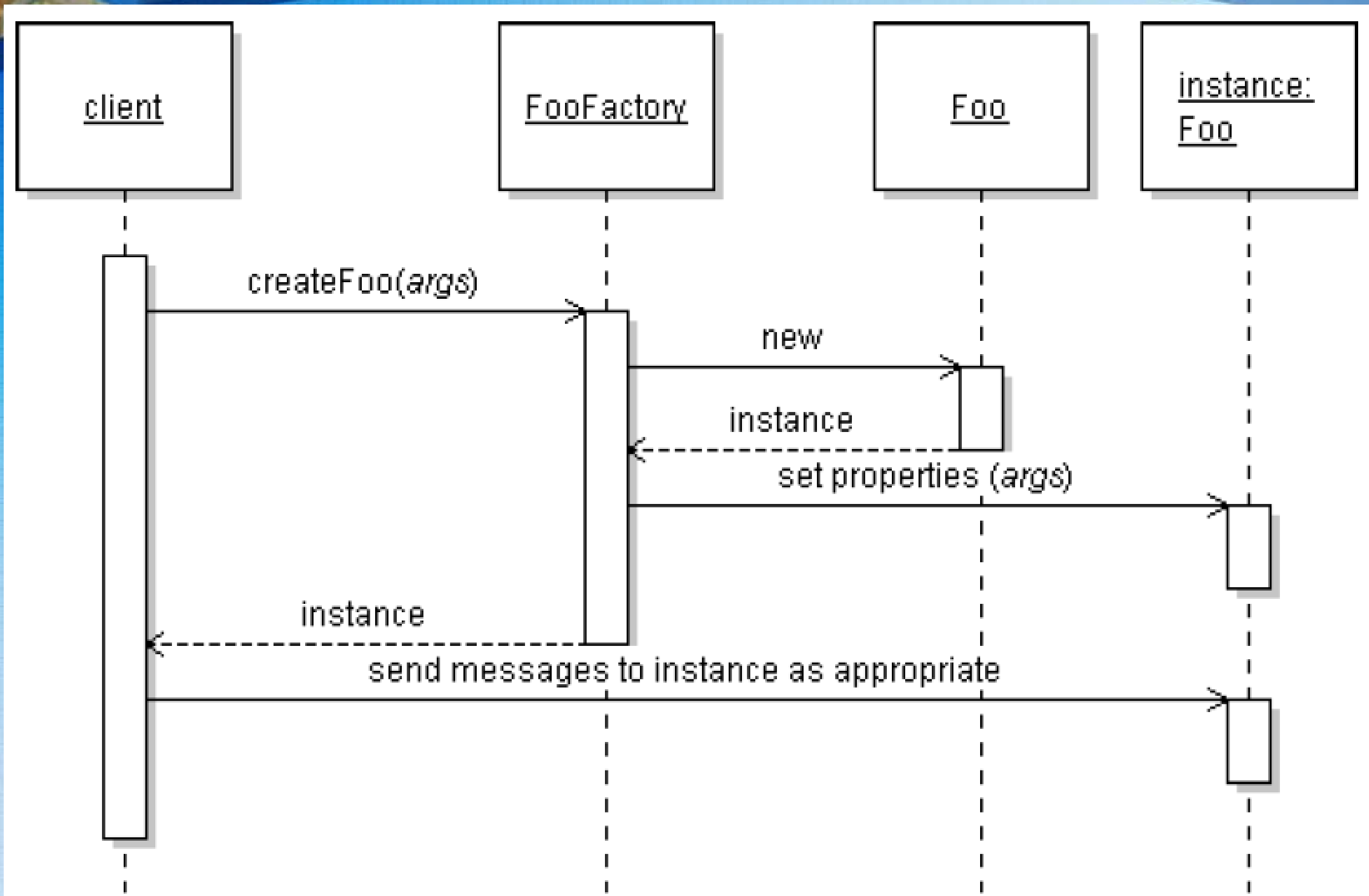
팩토리 패턴

- 팩토리: 다른 클래스의 인스턴스를 쉽게 생성하고 리턴하는 임무를 가진 클래스
  - 생성자를 부르는 대신 팩토리 클래스의 정적 메소드를 사용하여 객체를 셋업
  - 구축 정보를 사용 정보에서 분리(응집을 높이고 결합을 약하게 하기 위하여)하여 객체의 생성과 관리를 쉽게
  - 서브 클래스의 인스턴스화를 지연하는 효과

사용과 생성을 분리



팩토리 순서 다이어그램



팩토리 구현

필요한 팩토리를 만드는 방법

- 팩토리 자체를 인스턴스로 만들어야
  - Private 생성자로
- 팩토리는 컴포넌트를 생성할 때 static 메소드를 사용
- 팩토리는 인터페이스를 가능하면 간단하게 만들어 클라리언트가 쉽게 부르도록 해야

팩토리 사례

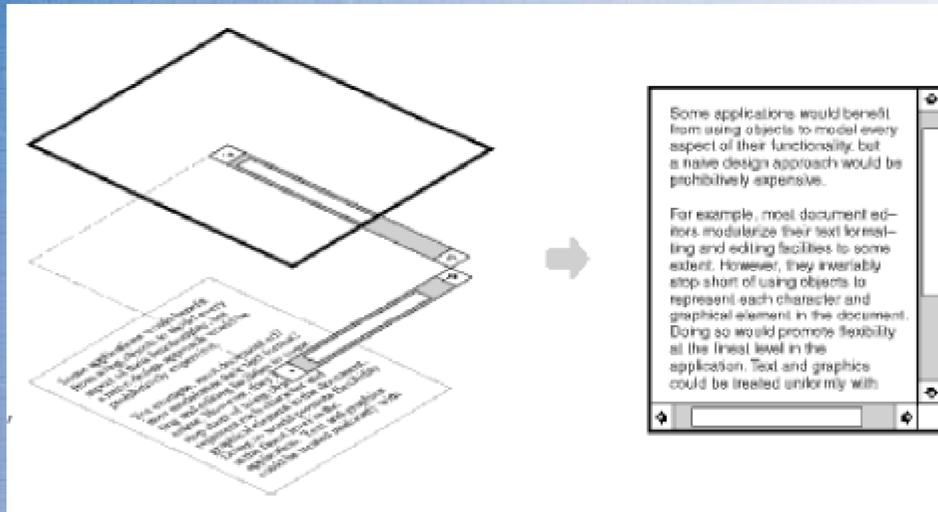
```
public class ImageReaderFactory
{
    public static ImageReader createImageReader(
            InputStream is ) {
            int imageType = figureOutImageType( is );
            switch( imageType ) {
                case ImageReaderFactory.GIF:
                    return new GifReader( is );
                case ImageReaderFactory.JPEG:
                    return new JpegReader( is ); // etc.
            }
    }
}
```

LECTURE

11

데코레이터 패턴

유용한 기능을 추가하기 위하여 다른 객체를
싸고 있는 객체



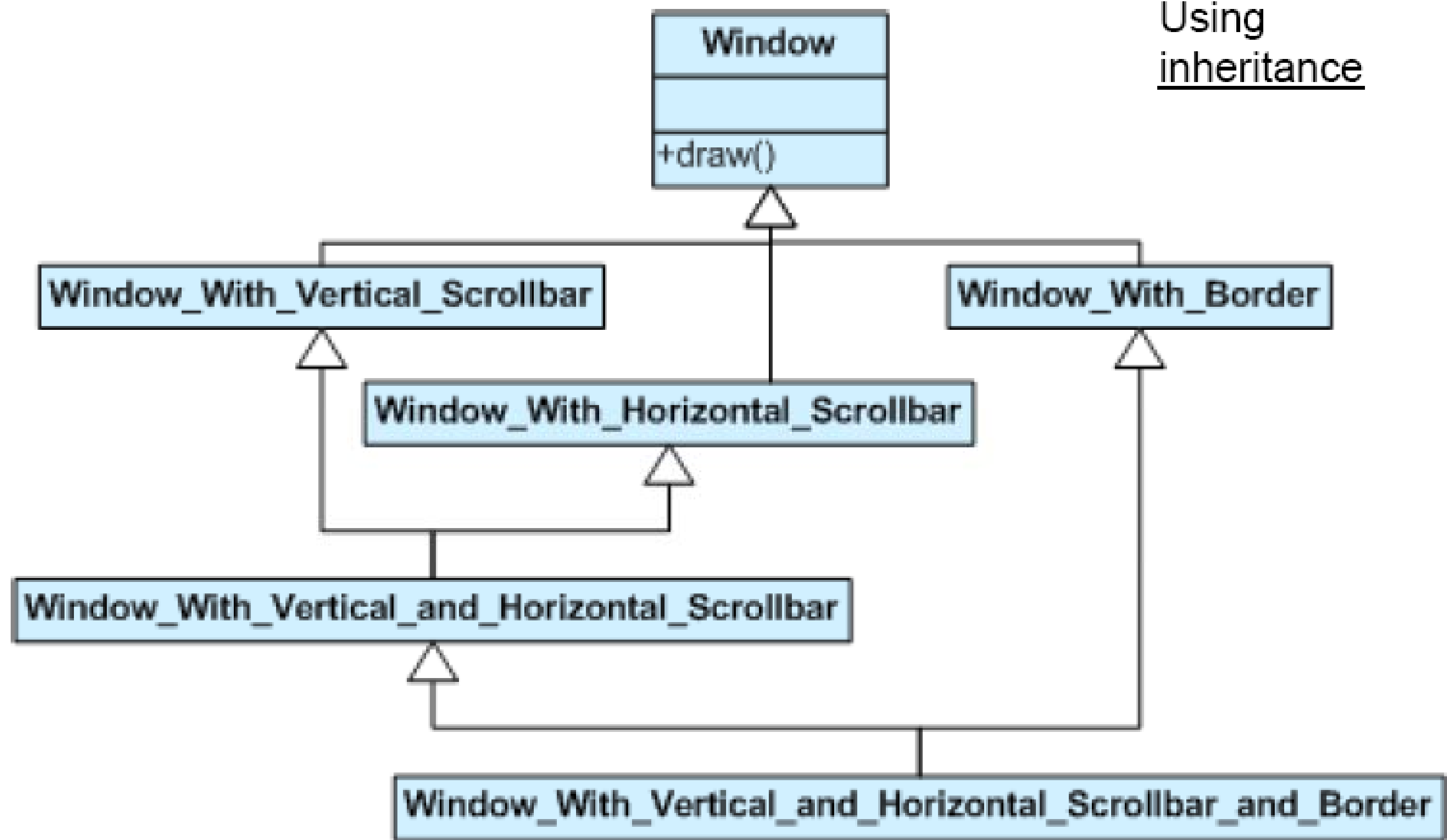
최은만, CSE 4039 소프트웨어 공학

데코레이터 패턴

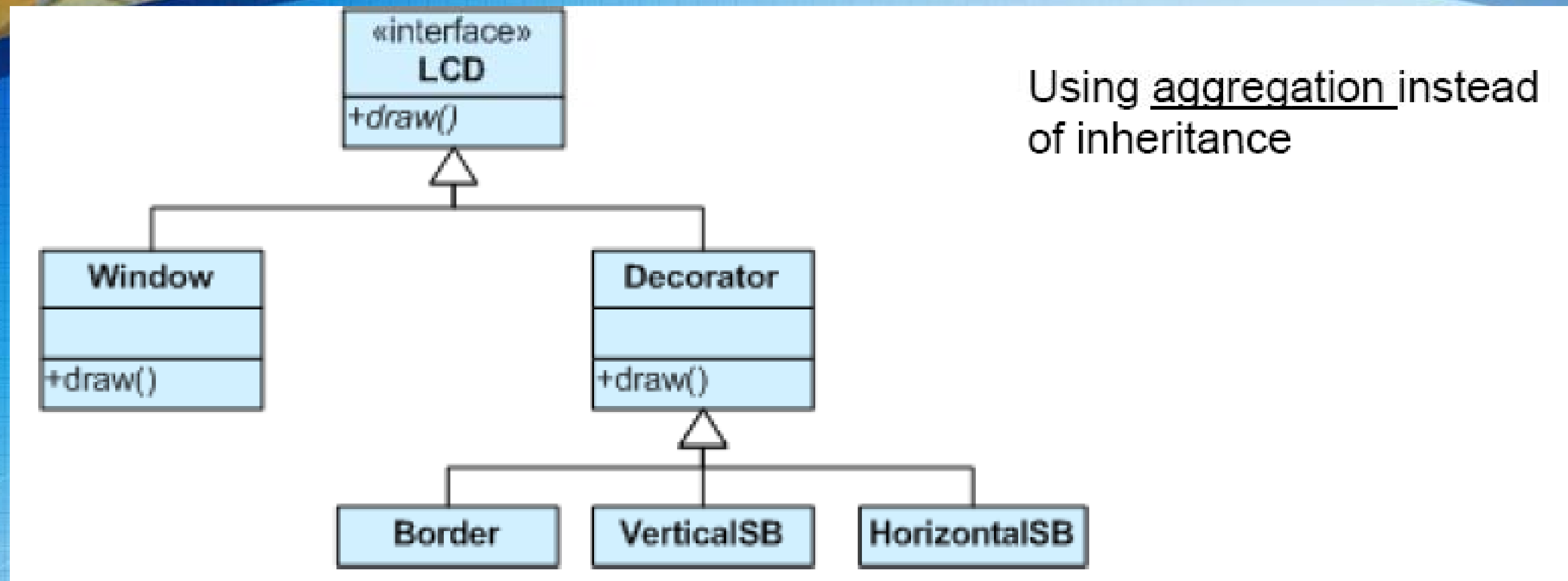
데코레이터: 다른 객체의 행위를 바꾸거나 기능을 추가하는 객체

- 객체에 동적으로 책무를 추가
- 데코레이션 당하는 객체가 데코이터를 알지 못함
- 데코레이터는 랩핑하려는 객체에 통일된 인터페이스를 제공해야

데코레이터 사례: GUI



데코레이터 객체의 사용




```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
  new VerticalScrollBarDecorator(
  new Window(80, 24))));
aWidget->draw();
```

LECTURE

11

퍼싸드 패턴

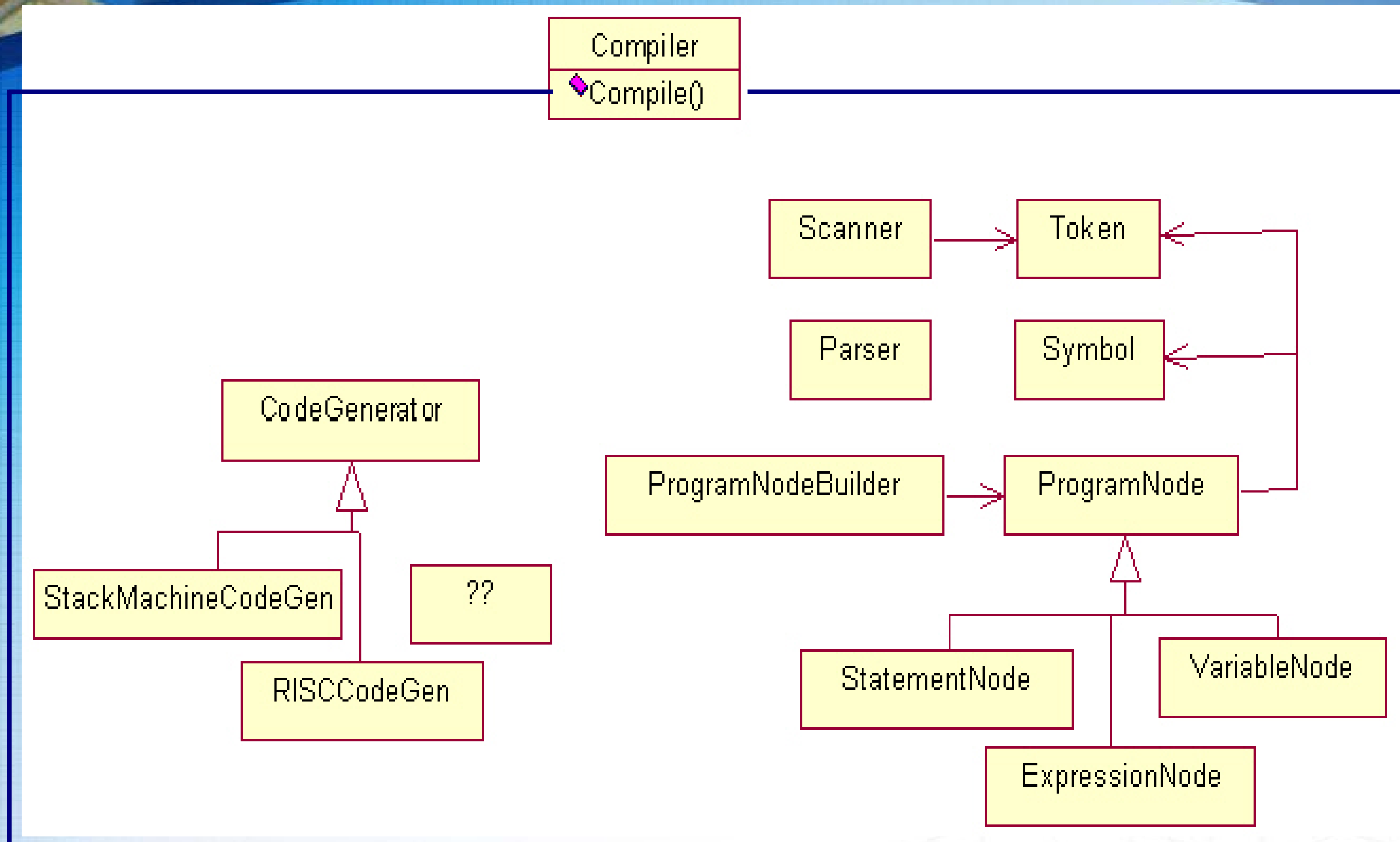
서로 다른 인터페이스 위에 통일된 인터페이스
또는
복잡한 인터페이스 위에 간단한 인터페이스를
씌움

퍼싸드 패턴

문제: 현재 인터페이스가 너무 복잡하여 쉽게 사용할 수 없거나 서브 시스템을 사용하는 데 너무 많은 선택이 있어

- 퍼싸드: 서로 다른 인터페이스 위에 통일된 인터페이스 또는 복잡한 인터페이스 위에 간단한 인터페이스를 제공하는 객체

퍼싸드 사례



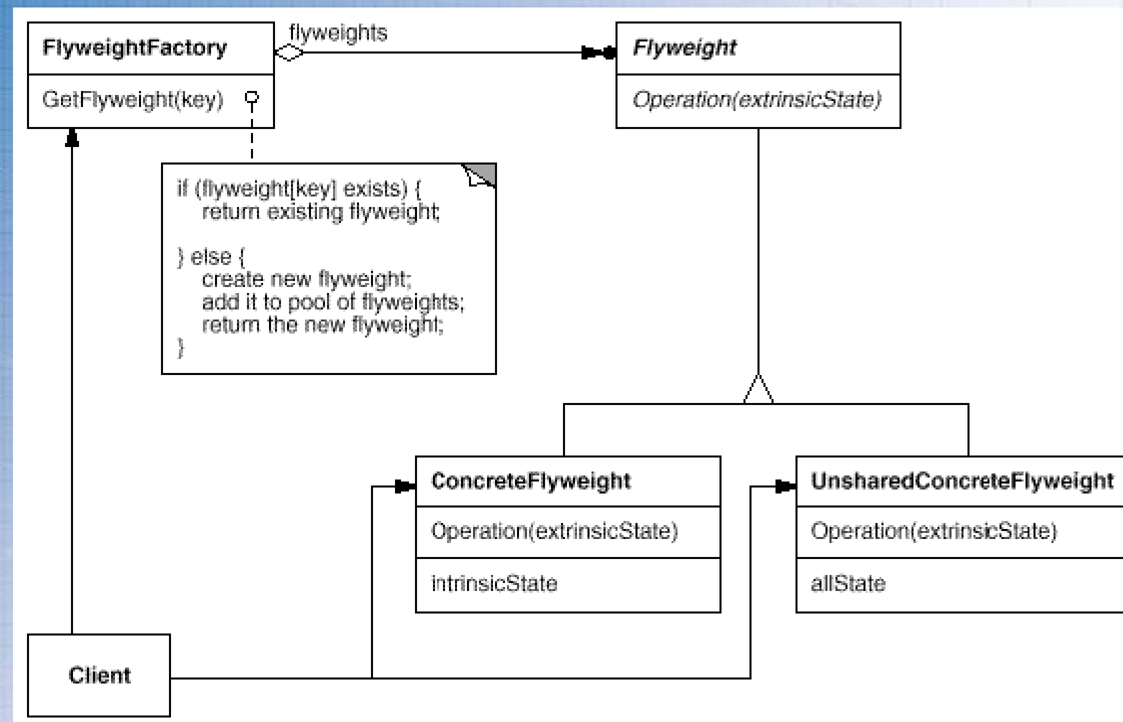
Compiler 퍼싸드

```
class Compiler {
public:
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

LECTURE

11

플라이웨이트 패턴



중복되는 객체 문제

- 문제: 중복되는 객체는 비효율적임
  - 여러 객체들이 같은 상태를 가짐
  - 예: 문서 편집기나 오류 메시지에서 사용하는 텍스트나 스트링 구조, 게임에 나오는 수많은 캐릭터들
  - 예: 디스크에 있는 같은 파일을 나타내는 파일 객체
    - new File(“note.txt”)
    - new File(“note.txt”)
    - new File(“note.txt”)
  - 그리드에 점을 나타내는 객체
    - new Point(x, y)
    - new Point(5.23432423, 3.14)

플라이웨이트 패턴

- 플라이웨이트: 어떤 클래스의 인스턴스도 동일한 상태를 갖지 않는다는 보장이 있어야
  - 객체 구축의 노력을 줄이기 위하여 객체들의 동일한 부분을 캐쉬로 만듦
  - 싱글톤과 유사하나 많은 인스턴스를 가지며 각 객체마다 고유한 상태를 가짐
  - 타입의 인스턴스는 많으나 각 인스턴스에 유사한 부분이 많은 경우에 유용함

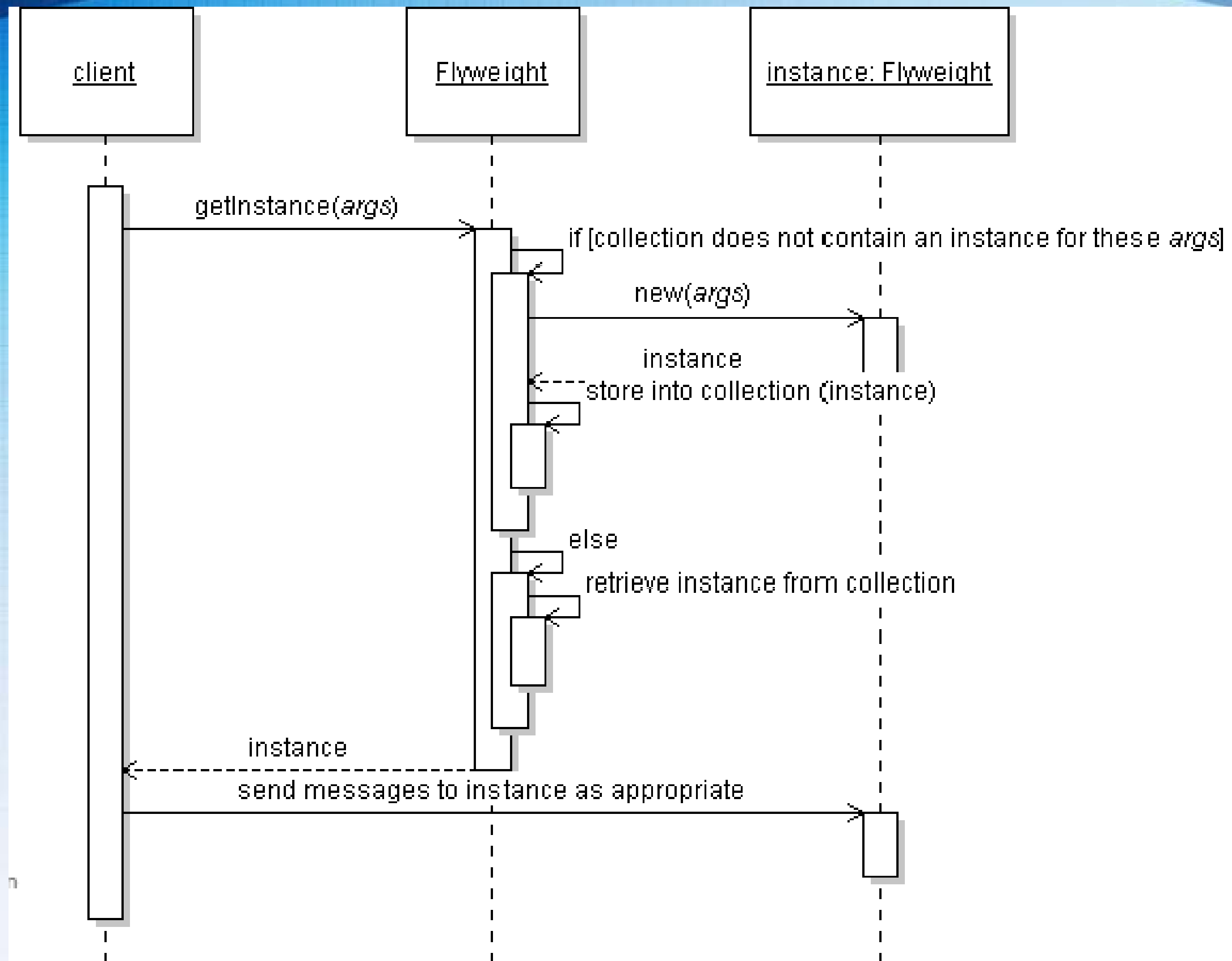
플라이웨이트의 구현

- 플라이웨이트 객체는 잘 변하지 않는 객체에 적합

의사코드

```
public class Flyweighted {
    o static collection (list) of instances
    o private constructor
    o static method to get an instance:
            if (we have created this kind of instance before),
        get it from the collection and return it
            else,
        create a new instance, store it in the collection and
return it
```

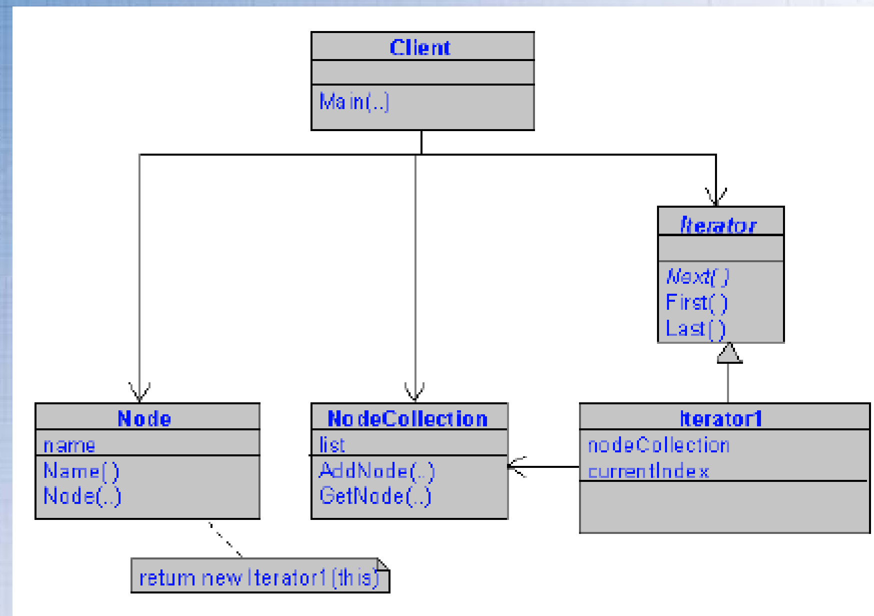
플라이웨이트 순서 다이어그램



LECTURE

11

반복자 패턴



최은만, CSE 4039 소프트웨어 공학

반복자 패턴

- 반복자: 집합에 포함된 객체를 검사하여 반복하는 일을 할 수 있도록 표준 방법을 제공하는 객체

- 장점:
  - 클라이언트 자세한 표현 방법을 몰라도 됨
  - 접근 인터페이스의 단순화

반복자의 구현

```
class List {
   public:
         int size() {…}
         boolean isEmpty() {…}
         ListElement* get(int index) {…}
}
public class ListIterator {
         int currentIndex;
   public:
         boolean hasNext() {…}
         ListElement* first() {…}
         ListElement* next() {…}
         ListElement* current() {…}
}
```

디자인 패턴의 선택

- 디자인 패턴이 주어진 문제를 어떻게 해결하고 있는지 스터디
  - 먼저 디자인 패턴을 잘 숙지
- 주어진 상황에 제일 잘 맞는 패턴이 무엇인지 숙고
  - 생성
  - 구조
  - 행위
- 시스템의 변경, 발전, 재사용 어느 측면이 유력한지 고려하여 적용