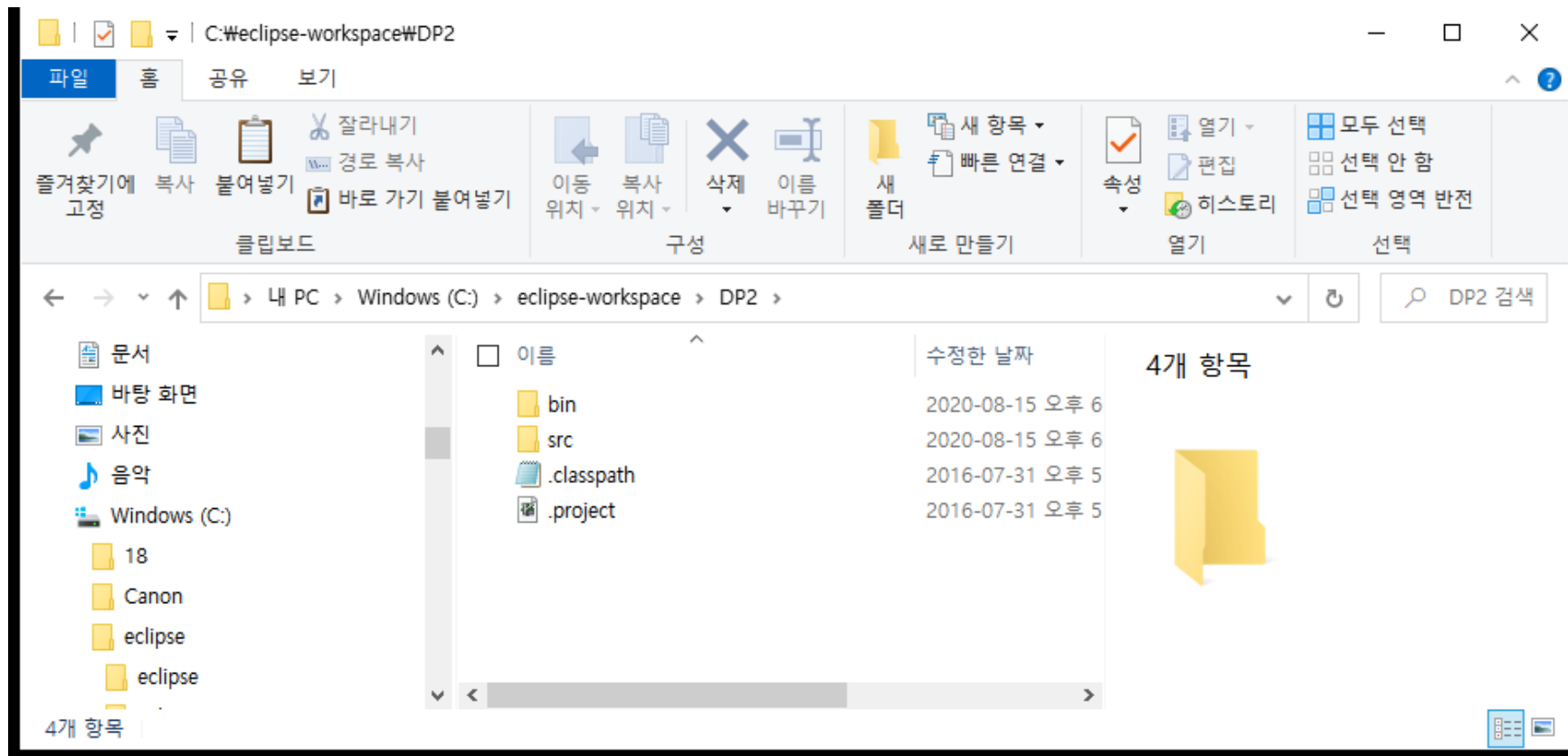


# **Design Patterns Practice**

# **0. Environment Settings**

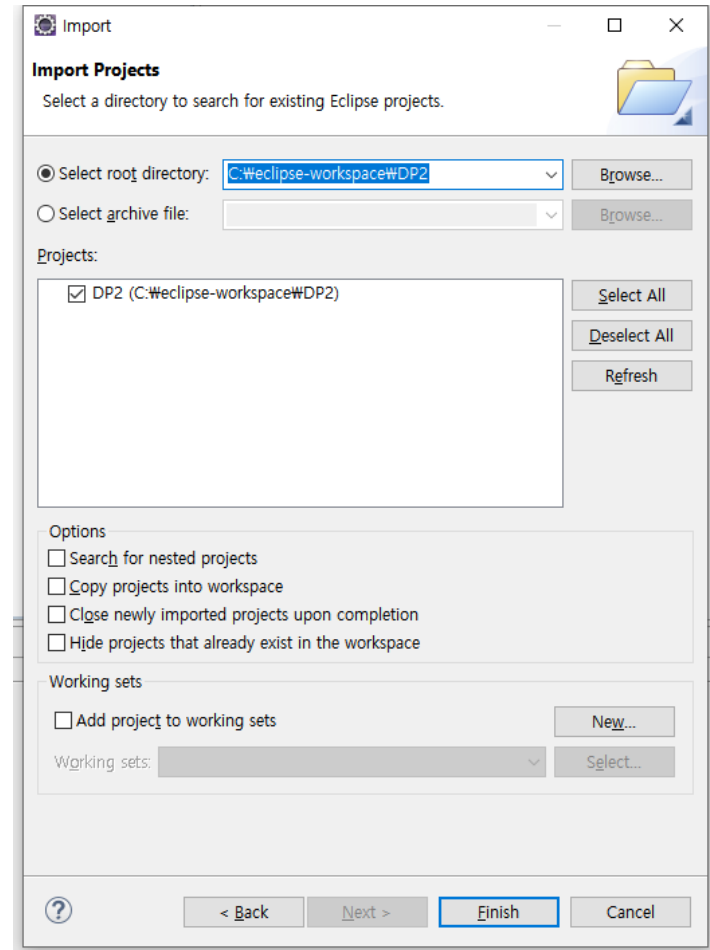
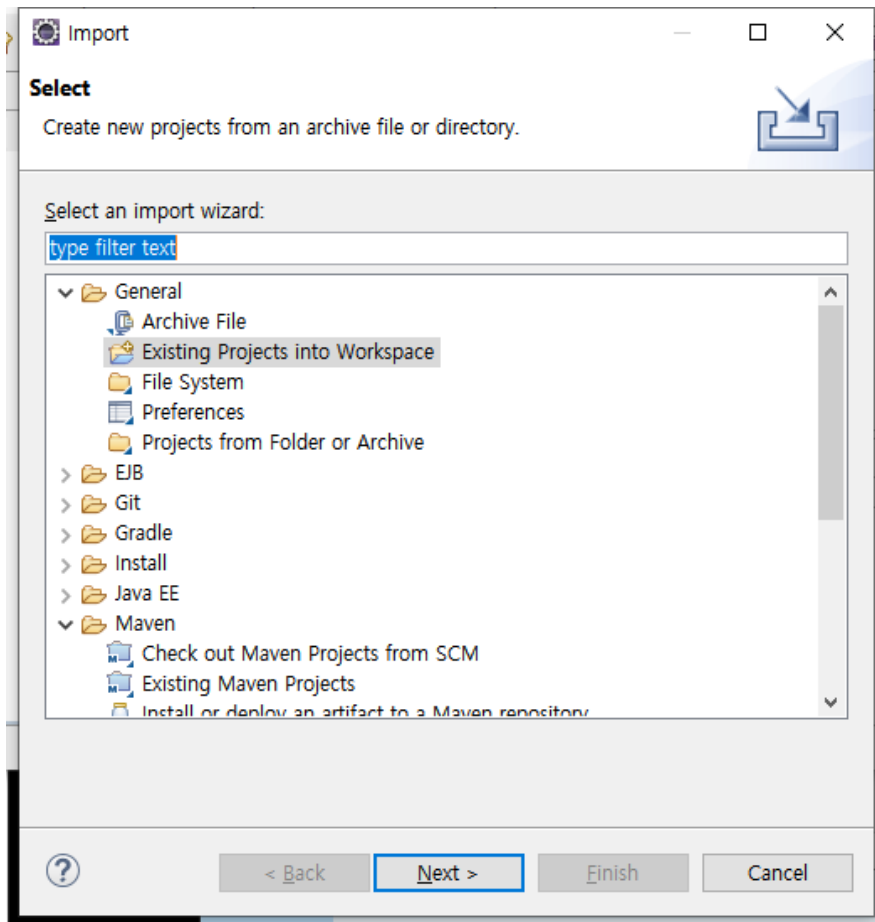
# 실습환경 구축 Step A1: 실습용 예제 복사

- 배포된 "DP2.zip" 파일을 열고, DP2 디렉토리를 포함하여 적당한 workspace 디렉토리에 압축해제



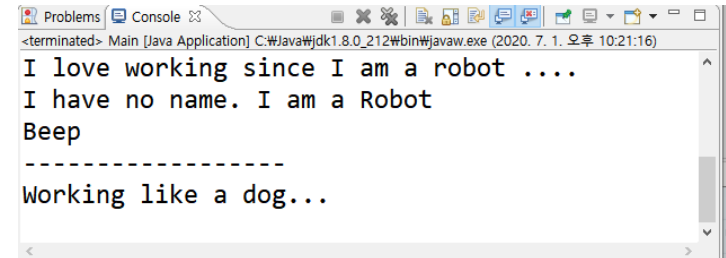
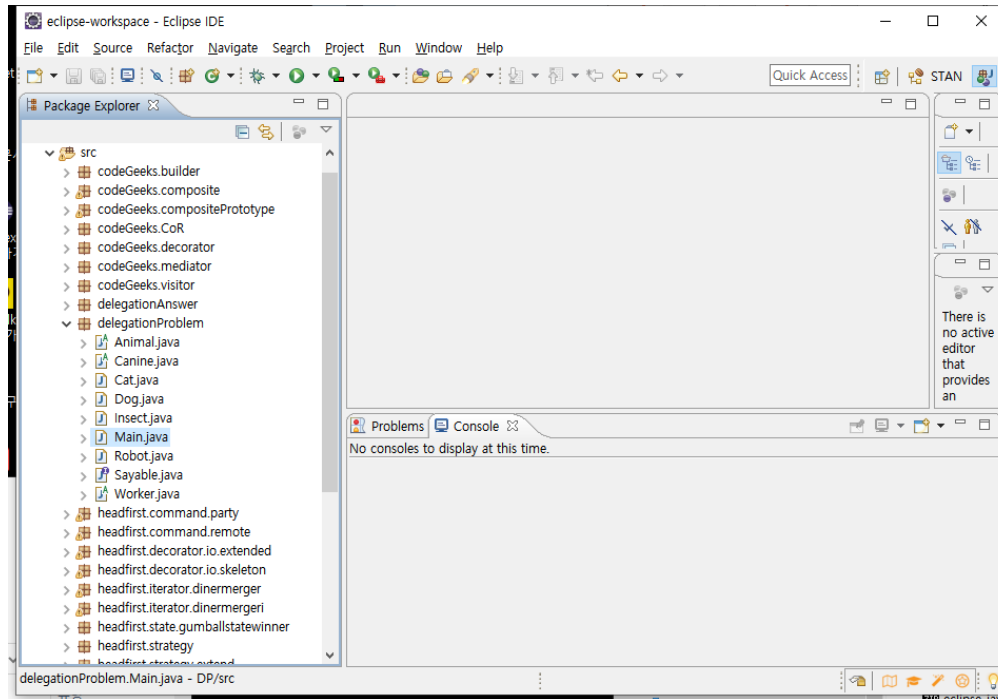
# 실습환경 구축 Step A2: 이클립스에서 импорт

- Eclipse를 구동하여 File 메뉴의 Import를 선택하고 "Existing Projects into Workspace"를 선택하여 자바 예제 프로젝트를 import



# 실습환경 구축 Step A3: delegationProblem 실행

- Package Explorer 탭에서 DP2/src/delegationProblem/Main.java 를 더블클릭 후 메뉴에서 Run>Run 선택 혹은 단축키 ctrl+F11 로 Main.java 컴파일 및 실행



A3 스텝까지 문제없이 진행되었다면,  
바로 실습 1번 문제부터 수행하시면 됩니다!

다음 페이지는 자바 and/or 이클립스가  
(재)설치 되어야하는 경우에만 따라하시기  
바랍니다.

# 실습환경 구축 Step B1: Java 8 JDK 설치

- 기존 Java 삭제 (설정/프로그램 추가-삭제)
- JDK 설치: OS에 따라 32비트용 혹은 64비트용 JDK 1.8 이상의 버전을 설치
- Path 설정
  - 내컴퓨터 -> 시스템속성 -> 환경변수
  - Path에 새로 설치된 자바 디렉토리 추가
    - C:\Program Files\Java\jdk설치된디렉토리\bin
    - C:\Program Files\Java\jre설치된디렉토리\bin
    - JAVA\_HOME 환경변수 : C:\Program Files\Java\jre설치된디렉토리\bin
- Path 확인
  - 코맨드라인(cmd) 에서, 다음의 두 명령 실행 후 버전이 동일하게 나오는지 확인
    - java -version
    - javac -version

# 실습환경 구축 Step B2: Eclipse 설치

---

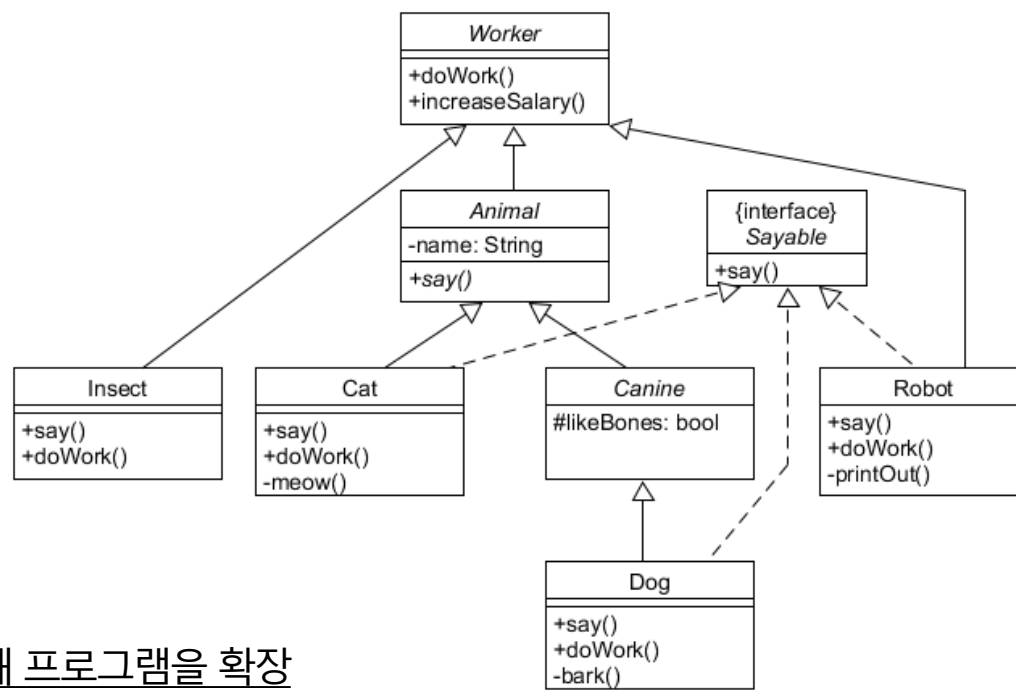
- 기존 Eclipse 삭제 (예를 들어 `Wprogram filesWeclipseW` 디렉토리 삭제)
- Eclipse 설치: OS에 따라 32비트 혹은 64비트 선택하여 Eclipse Java 개발자용 설치
  - Root 혹은 Program Files 등 적당한 디렉토리에 unzip
- 앞쪽의 Step A1 ~ A3를 따라서 실습용 자바 예제 실행

# **1. Object Composition and Delegation**



# Practice 1 : Delegation 실습

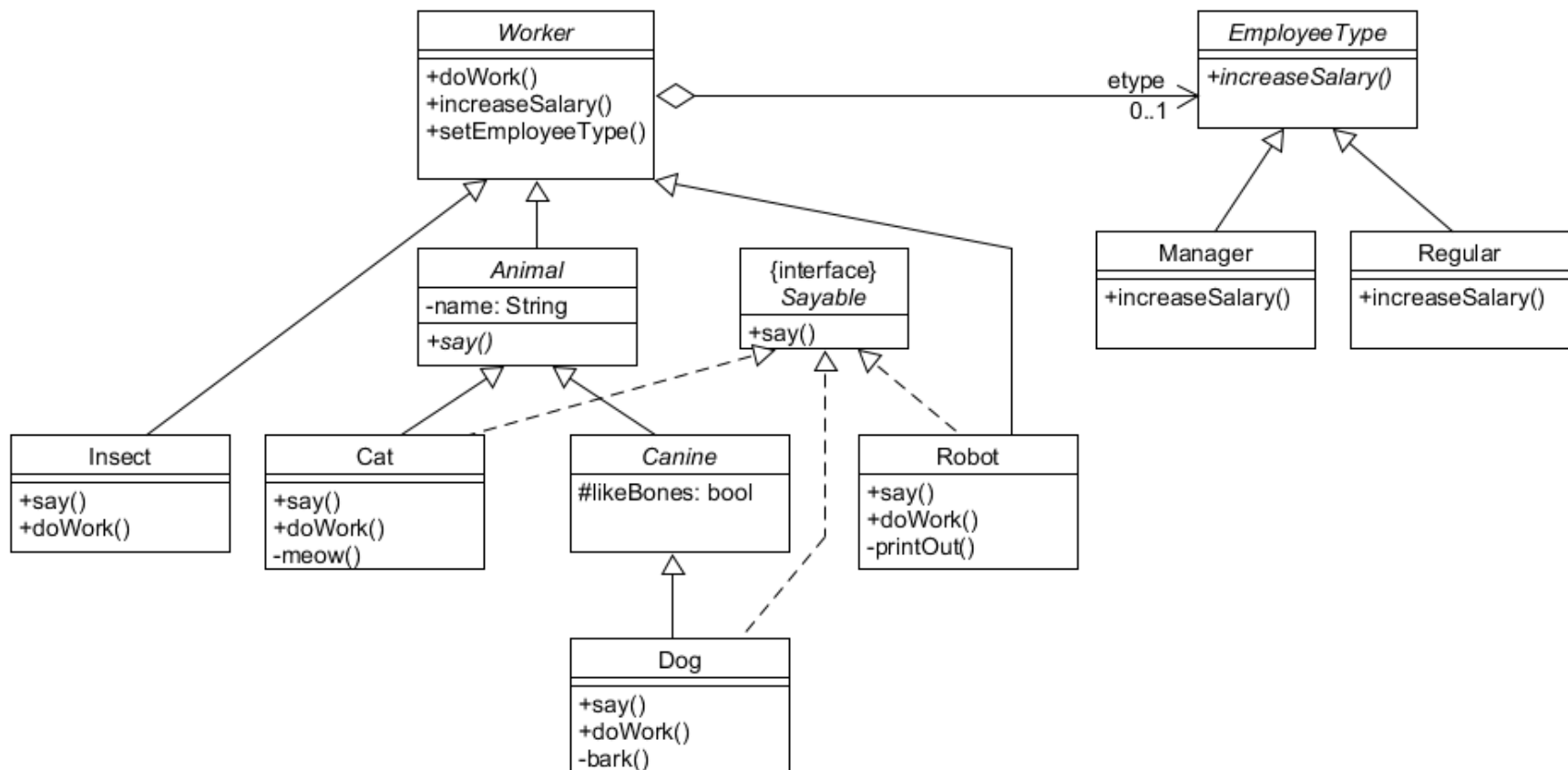
- 연습 소스: delegationProblem
- 답안 예시: delegationAnswer



- 각 Worker의 연봉을 관리하기 위해 프로그램을 확장
  - Worker의 연봉 인상 알고리즘을 결정하는 요소인 EmployeeType에는 Regular 및 Manager 가 있음
  - 각 Worker 인스턴스에 대해 EmployeeType을 동적으로 바꿀 수 있는 API를 제공해야 함
  - increaseSalary(double rate)는 Worker의 EmployeeType에 따라 다음과 같이 동작
    - Manager 인 경우: 기존 봉급에 \$10를 우선 더하고, 증가된 봉급에 rate 만큼 인상
    - Regular 인 경우: 기존 봉급에 rate 만큼 인상
  - 새로운 EmployeeType이 추후에 더 추가될 예정임을 염두에 두고 설계 구현 할 것

# Practice 1 : Delegation 실습 (해답)

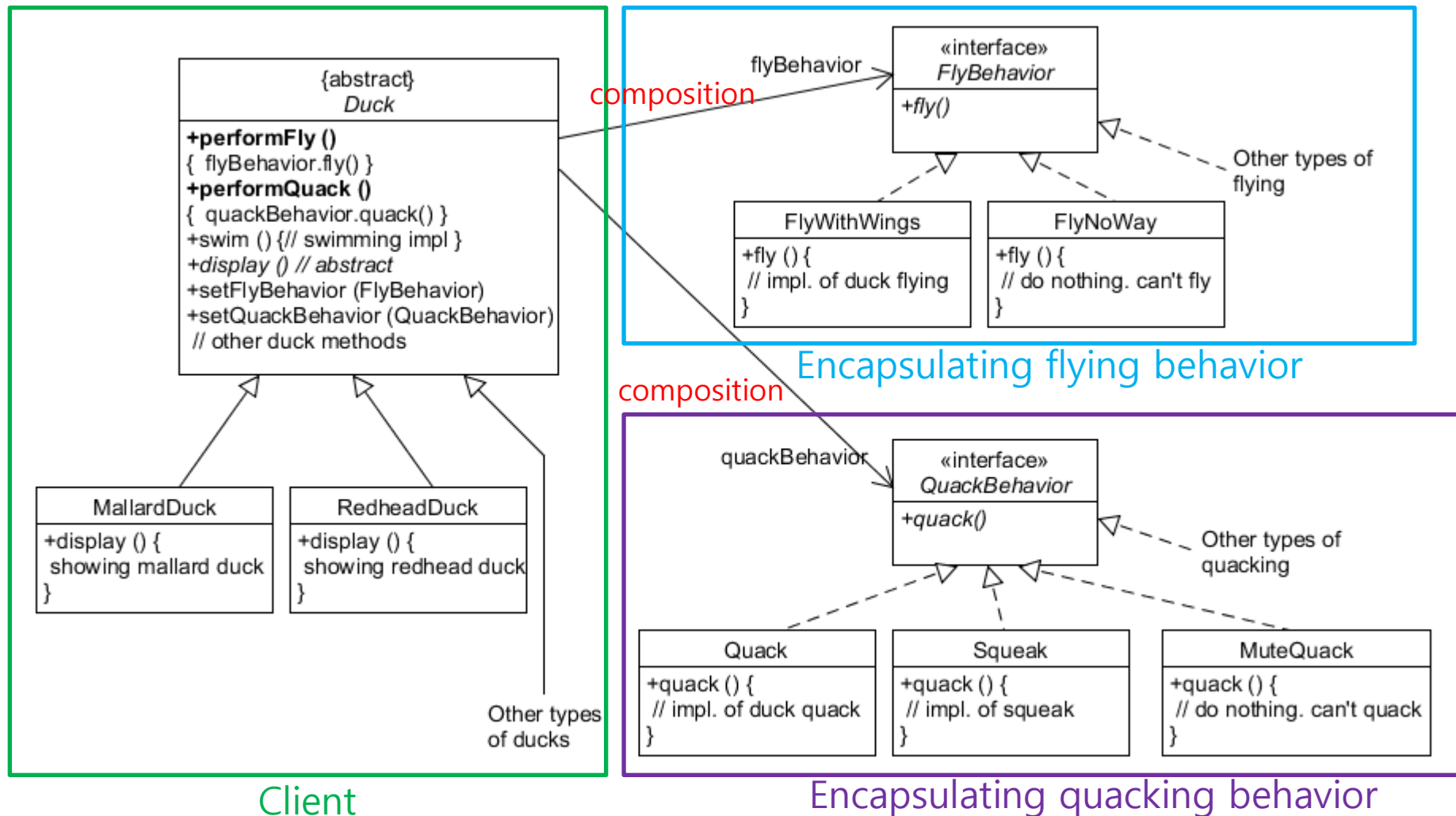
- 연습 소스: delegationProblem
- 답안 예시: delegationAnswer



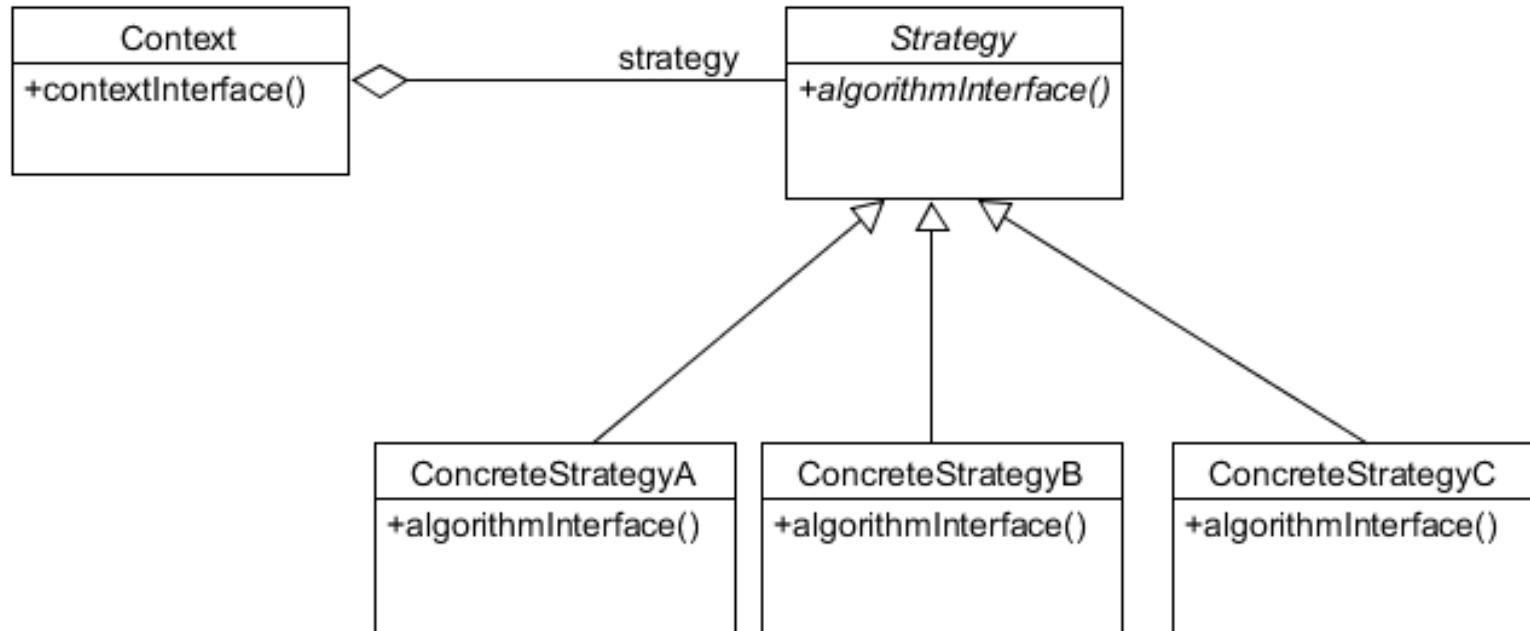
## **2. Strategy Pattern**

# Designing for Change

- Design patterns help to ensure that a system can change in specific ways → easier to change



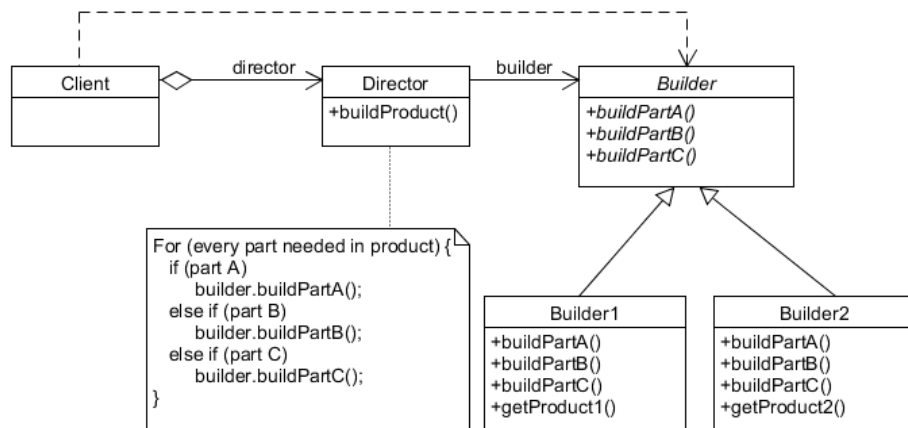
# Algorithmic Dependencies



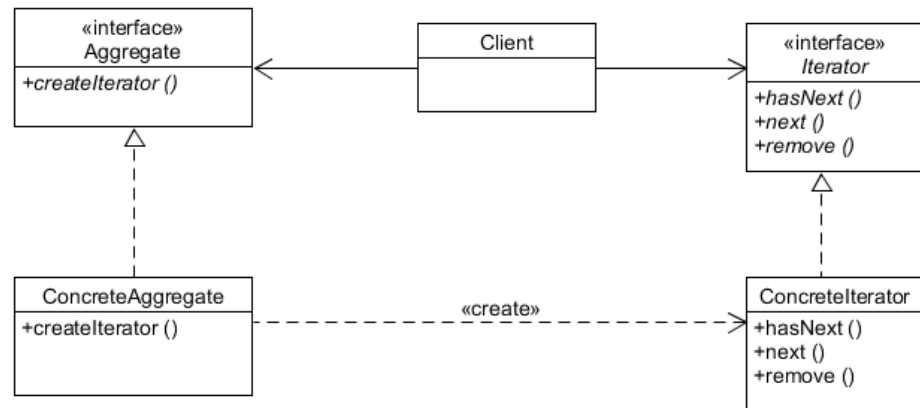
Strategy pattern

# Algorithmic Dependencies

- Algorithm changes → object changes
  - algorithm that are likely to change should be isolated
- Design patterns : Strategy, Builder, Iterator, Template Method, Visitor



Builder pattern



Iterator pattern

# Practice 1 : Quack Duck 예제 실행

---

- 연습 소스: headfirst.strategy

(1) MiniDuckSimulator.java 더블클릭 후 Ctrl-F11로 Run

(2) Duck abstract class 살펴보기

(3) MallardDuck과 RedHeaduck 클래스 살펴보기

(4) RubberDuck과 DecoyDuck 클래스 살펴보기

(5) FlyBehavior와 QuackBehavior 인터페이스 살펴보기

(6) MuteQuack, Quack, Squeak 클래스 살펴보기

(7) headfirst.strategy 패키지의 모든 클래스 및 인터페이스를 종합하여 클래스 다이어그램 그리기

# Practice 2 : 새로운 Quack Behavior와 Duck 추가

- 연습 소스: headfirst.strategy

## (1) DoubleQuack 추가

- Duck의 quack behavior의 새로운 형태인 DoubleQuack을 추가하시오.
- 해당 DoubleQuack의 quack 메소드는 다음과 같이 구현하시오.

```
public void quack() {  
    System.out.println("Quack, Quack");  
}
```

- 위의 메소드를 포함하는 DoubleQuack 클래스를 만드시오.
- MiniDuckSimulator.java 에서 ModelDuck을 이용하여 추가된 DoubleQuack을 테스트 하시오.

## (2) BadDuck 추가

- ModelDuck의 child class로 BadDuck을 추가
- BadDuck에 counter를 두고, behavior에 대한 명령 중 홀수번째만 제대로 수행하고, 짝수번째 명령은 무시하도록 만드시오.
- MiniDuckSimulator.java 에서 BadDuck을 이용하여 테스트 하시오.



# Practice 3 : Duck의 Behavior Type 확장

- 연습 소스: headfirst.strategy
- 답안 예시: headfirst.strategy.extend

## (1) EggBehavior 추가

- Duck의 새로운 behavior type으로 EggBehavior를 추가하시오. (기존 인터페이스 참고)
- setEggBehavior와 performEggBehavior를 추가하시오.
- EggBehavior를 구현하는 concrete class로 SpawnEgg와 SpawnNothing 클래스를 추가하시오.
- 모든 Duck의 기본 EggBehavior는 SpawnNothing으로 초기화되도록 코드를 작성하시오.

```
public interface EggBehavior {  
    public void spawn();  
}
```

- 테스트를 위해 MiniDuckSimulator에서는 다음과 같이 기존 소스코드를 이용하여 Duck 인스턴스의 배열을 만들고, for 문을 이용하여 각 인스턴스의 display()와 performEgg()를 호출하시오.

```
Duck ducks[] = {mallard, rubberDuckie, decoy, model};
```

- mallard 인스턴스만 SpawnEgg를 할 수 있도록 setEggBehavior를 이용하여 바꾸고, 배열의 모든 원소에 대해 display()와 performEgg()를 호출하여 검사하시오.

# Practice 4 : Duck의 Behavior 동적 복사

---

- 연습 소스: headfirst.strategy
- 답안 예시: headfirst.strategy.extend
  - (1) ModelDuck에 void CopyBehavior(Duck source) 메소드 추가
  - 해당 메소드는 source로 지정된 Duck의 flyBehavior, quackBehavior, eggBehavior를 복사해주는 기능을 구현해야 함
  - 테스트를 위해 MiniDuckSimulator에서는 CopyBehavior메소드를 이용하여 mallard 객체의 behavior를 model 객체로 복사한 후, 배열의 모든 원소에 대해 display()와 performEgg를 호출하여 행위 복사가 올바르게 되었는지 검사하시오.

# **3. Command Pattern**

# Practice 1 : HeadFirst Command Pattern

---

- 연습 소스: `headfirst.command.remote`
- 답안 예시: `headfirst.command.party` (iterator는 제외)
  - (1) 매크로 Command를 만들어서 여러 primitive Command를 포함할 수 있도록 구현
  - (2) 매크로 Command에 primitive Command의 저장을 위해 ArrayList를 사용하고, 각 primitive Command 접근 시에 iterator 패턴을 적용
  - (3) 테스트

# Practice 2 : RC Martin's ActiveObject

- 연습 소스: activeObject

- 답안 예시:

(1) RC Martin의 Command 패턴 예제인 ActiveObject 프로젝트를 run

(2) 프린트하는 숫자와 딜레이를 조정하여 실행

```
engine.addCommand(new DeLayedTyper(100, '1'));
```

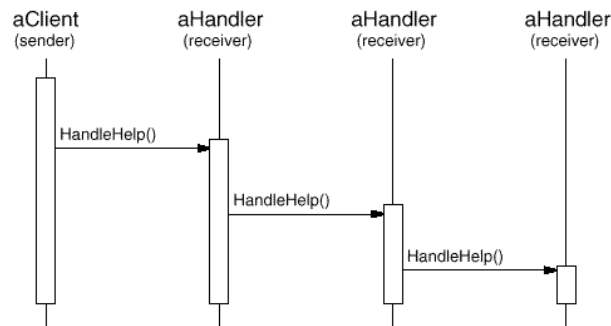
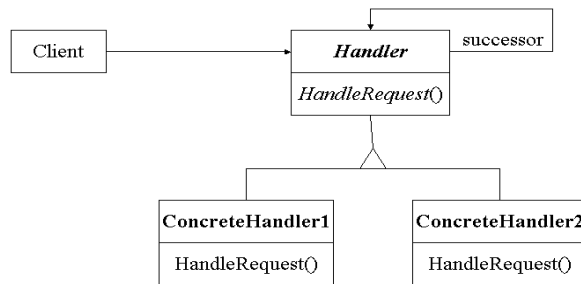
```
engine.addCommand(new DeLayedTyper(300, '3'));
```

```
engine.addCommand(new DeLayedTyper(500, '5'));
```

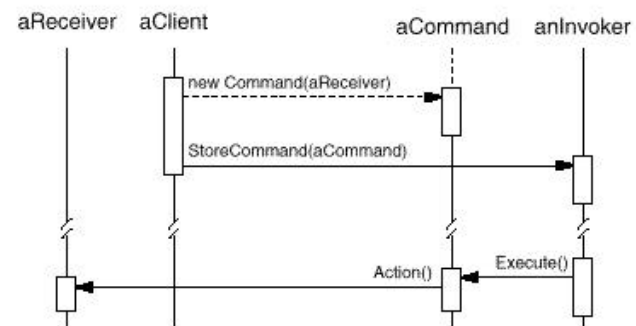
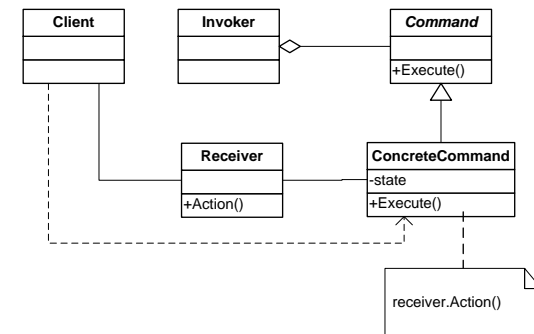
```
engine.addCommand(new DeLayedTyper(700, '7'));
```

## **4. Visitor Pattern**

- Need to avoid hard-coded requests
- Design patterns : Chain of Responsibility, Command



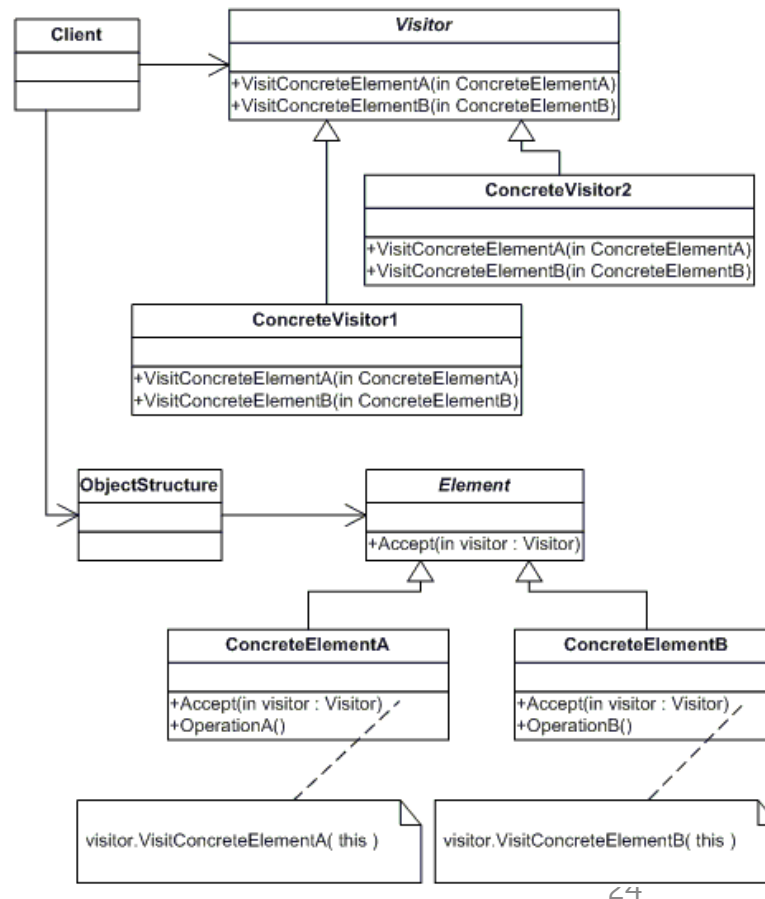
## Chain of Responsibility pattern



## Command pattern

# Algorithmic Dependencies

- algorithm changes → object changes
  - algorithm that are likely to change should be isolated
- Design patterns : Builder, Iterator, Strategy, Template Method, Visitor





# Practice 1 : accept 메소드 중복구현 피해보기

- 연습 소스: Visitor.interfaceVisitor
- 각 element 클래스에서 중복 구현되는 accept 메소드 없애기
  - ICarElement interface를 abstract class로 변경하고 accept 메소드 구현
  - 각 concrete element에서 implement를 extend로 바꾸기
  - 각 concrete element에서 accept 메소드 없애기
  - 각 visitor에서 ICarElement를 위한 visit(ICarElement) 추가 (컴파일을 위해 필요)

```
class Wheel implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
```

# Practice 1 : accept 메소드 중복구현 피해보기

- 연습 소스: Visitor.interfaceVisitor
  - ICarElement interface를 abstract class로 변경하고 accept 메소드 구현
  - 각 concrete element에서 implement를 extend로 바꾸기
  - 각 concrete element에서 accept 메소드 없애기

```
interface ICarElement {  
    void accept(ICarElementVisitor visitor);  
}
```



```
abstract class ICarElement {  
    void accept(ICarElementVisitor visitor) {  
        visitor.visit(this);  
    }  
}
```

```
class Wheel implements ICarElement {  
    public void accept(ICarElementVisitor  
        visitor)  
    {  
        visitor.visit(this);  
    }  
}
```



```
class Wheel extends ICarElement {  
    public void accept(ICarElementVisitor  
    visitor)  
    {  
        visitor.visit(this);  
    }  
}
```

# Practice 1 : accept 메소드 중복구현 피해보기

- 연습 소스: Visitor.interfaceVisitor

각 visitor에서 ICarElement를 위한 visit(ICarElement) 추가 (컴파일을 위해 필요)

```
interface ICarElementVisitor {  
    void visit(Wheel wheel);  
    void visit(Engine engine);  
    void visit(Body body);  
    void visit(Car car);  
}
```

```
class CarElementPrintVisitor implements  
ICarElementVisitor {  
    public void visit(Wheel wheel) {  
        System.out.println("Visiting " +  
wheel.getName() + " wheel");  
    }  
    public void visit(Engine engine) {  
        System.out.println("Visiting engine");  
    }  
    public void visit(Body body) {  
        System.out.println("Visiting body");  
    }  
    public void visit(Car car) {  
        System.out.println("Visiting car");  
    }  
}
```



```
interface ICarElementVisitor {  
    void visit(Wheel wheel);  
    void visit(Engine engine);  
    void visit(Body body);  
    void visit(Car car);  
    void visit(ICarElement dummy);  
}
```



```
class CarElementPrintVisitor implements  
ICarElementVisitor {  
    public void visit(Wheel wheel) {  
        System.out.println("Visiting " +  
wheel.getName() + " wheel");  
    }  
    public void visit(Engine engine) {  
        System.out.println("Visiting engine");  
    }  
    public void visit(Body body) {  
        System.out.println("Visiting body");  
    }  
    public void visit(Car car) {  
        System.out.println("Visiting car");  
    }  
    public void visit(ICarElement dummy) {  
        System.out.println("!!! Dummy !!!");  
    }  
}
```

# Practice 2 : Visitor 추가

---

- 연습 소스: Visitor.interfaceVisitor
- 답안 예시: Visitor.addVisitor
- ConcreteVisitor 추가
  - CarElementDestroyVisitor 추가
  - 수정의 범위 가늠하기

# Practice 3 : Element 추가

---

- 연습 소스: Visitor.addVisitor
- 답안 예시: Visitor.addElement
- ConcreteElement 추가
  - Trunk 추가
  - 수정의 범위 가늠하기

# Practice 4 : FileFindVisitor Visitor 추가

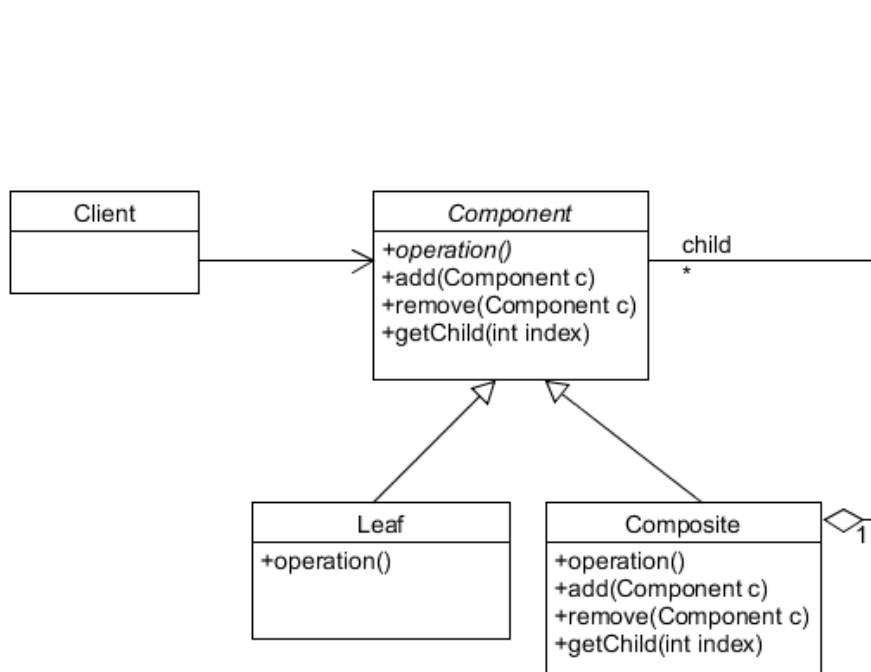
---

- 연습 소스: `hiroshi.VisitorProblem`
- 답안 예시: `hiroshi.VisitorAnswer`
  - (1) 기존 Visitor 코드를 분석하시오.
  - (2) FileFindVisitor는 생성자로 전해진 확장자를 가진 파일들을 찾아내는 기능을 제공한다. 해당 비지터를 새로 만들고 기존 디렉토리와 파일에 적용하시오.

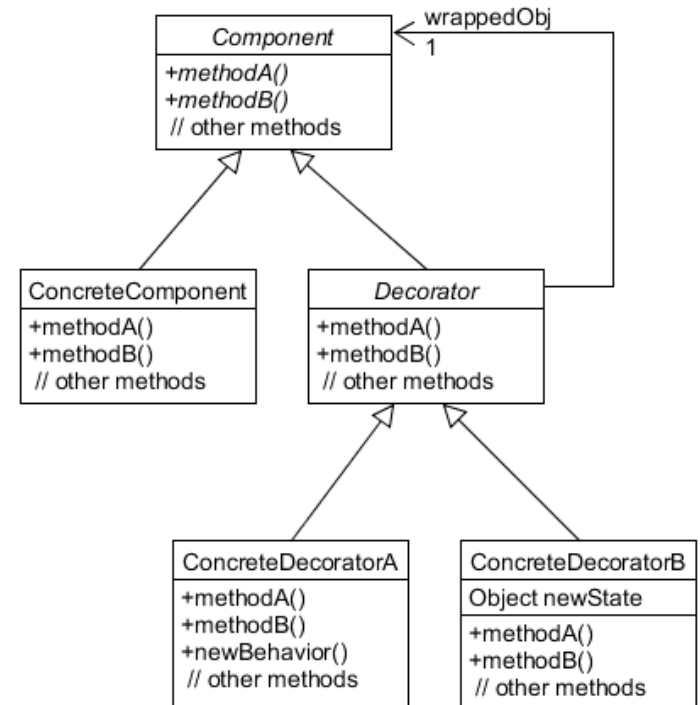
# **5. Decorator, Composite Pattern Practice**

# Avoiding Sub-classing for Func. Extension

- Subclassing can lead to explosion of subclasses
  - object composition and delegation provide flexible alternatives



Composite pattern



Decorator pattern



# Practice 1 : Composite Pattern

---

- 연습 소스: `hiroshi.directoryCompositeProblem`
- 답안 예시: `hiroshi.directoryCompositeAnswer`
  - (1) `Main.java`에 있는 여러 구조를 변경해 보면서 임의의 구조를 만들어 보시오.
  - (2) `Entry`에 새로운 method인 `getFullName()`을 추가하여, 루트로부터의 full path 정보를 돌려주도록 확장하십시오.
  - (3) `Main.java`에 적절한 테스트 코드를 넣어 시험해보시오.

# Practice 2 : StarbuzzCoffee 연습 (Decorator)

---

- 연습 소스: `headfirst.decorator.starbuzz`
- 답안 예시:
  - (1) HouseBlend 커피에 Soy 1번, Mocha 2번, Whip 2번의 순서로 추가하여 가격 계산
  - (2) HouseBlend 커피에 위의 역순으로 추가하여 가격 계산
  - (3) HouseBlend 커피를 DarkRoast로 꾸미기 시도
  - (4) Mocha를 Beverage로 하여 Whip 추가 시도

# Practice 3 : 새로운 스트림 필터 개발

- 연습 소스: [headfirst.decorator.io.skeleton](http://headfirst.decorator.io/skeleton)
- 답안 예시: [headfirst.decorator.io.extended](http://headfirst.decorator.io/extended)
- 영문자에 대해 circular shift를 지원하는 `ShiftInputStream` 클래스를 구현한 후 기존 스트림 필터와 조합 테스트

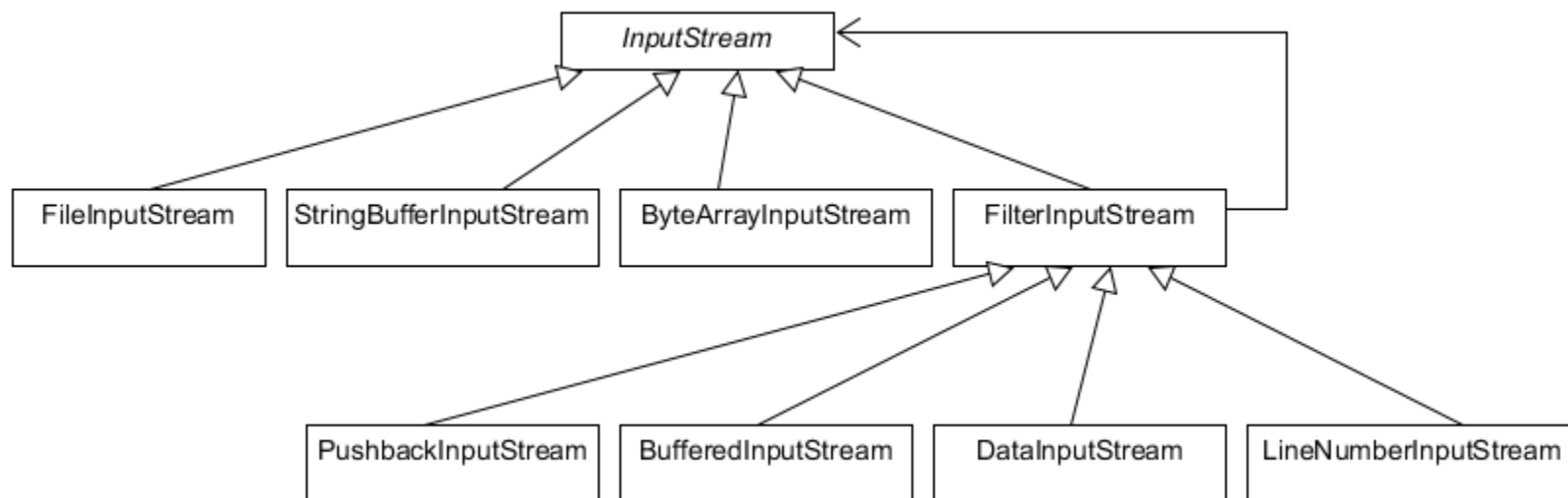
***ShiftInputStream(InputStream in, int offset)***

***ShiftInputStream(InputStream in)***

***//offset = 0 버전***

Ex) offset = 1에 대해, input: abc 이면 result: bcd

offset = -1에 대해, input: A12ZX a# 이면 result: Z12YW z#



# A Sample of Java I/O Decorator

```
Public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? C : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

# Run our Java I/O decorator

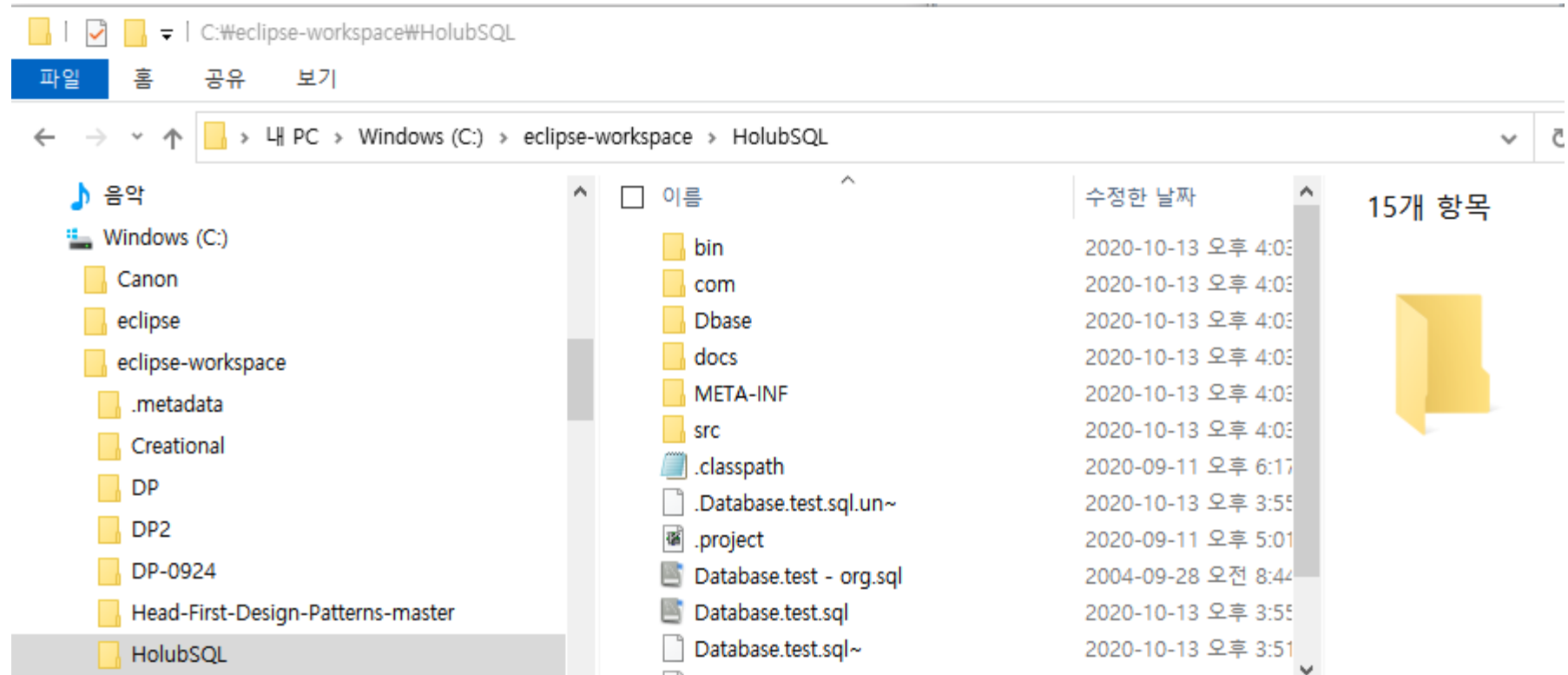
```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.println((char)c);
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## **6. Holub's Database Project 코드 분석 및 Pattern Practice**

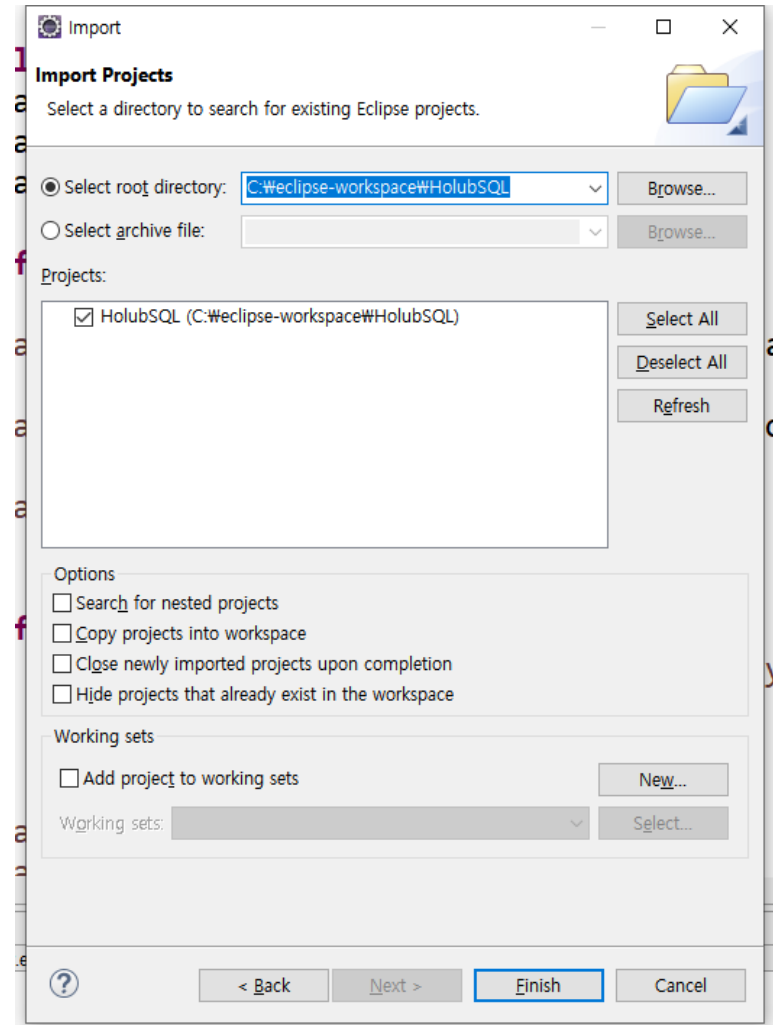
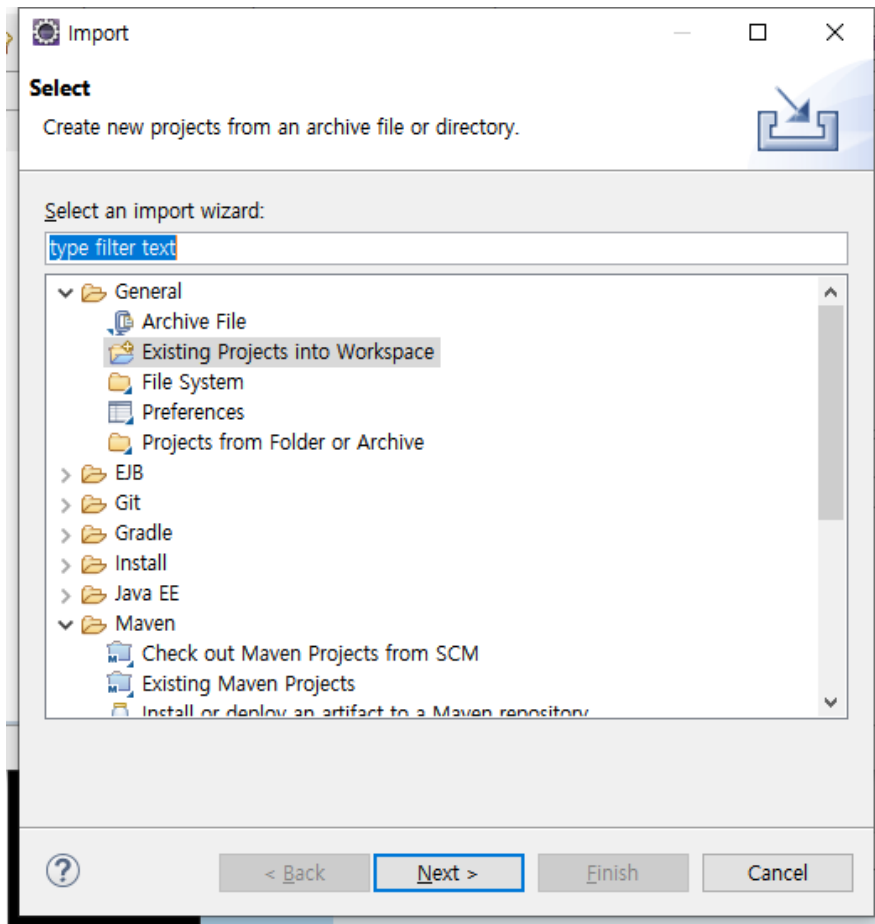
# \*실습환경 구축 Step 1: 실습용 SQL 예제 복사

- 배포된 "HolubSQLFixed.zip" 파일을 열고, HolubSQL 디렉토리를 포함하여 workspace 디렉토리 아래에 압축해제



# \*실습환경 구축 Step 2: 이클립스에서 임포트

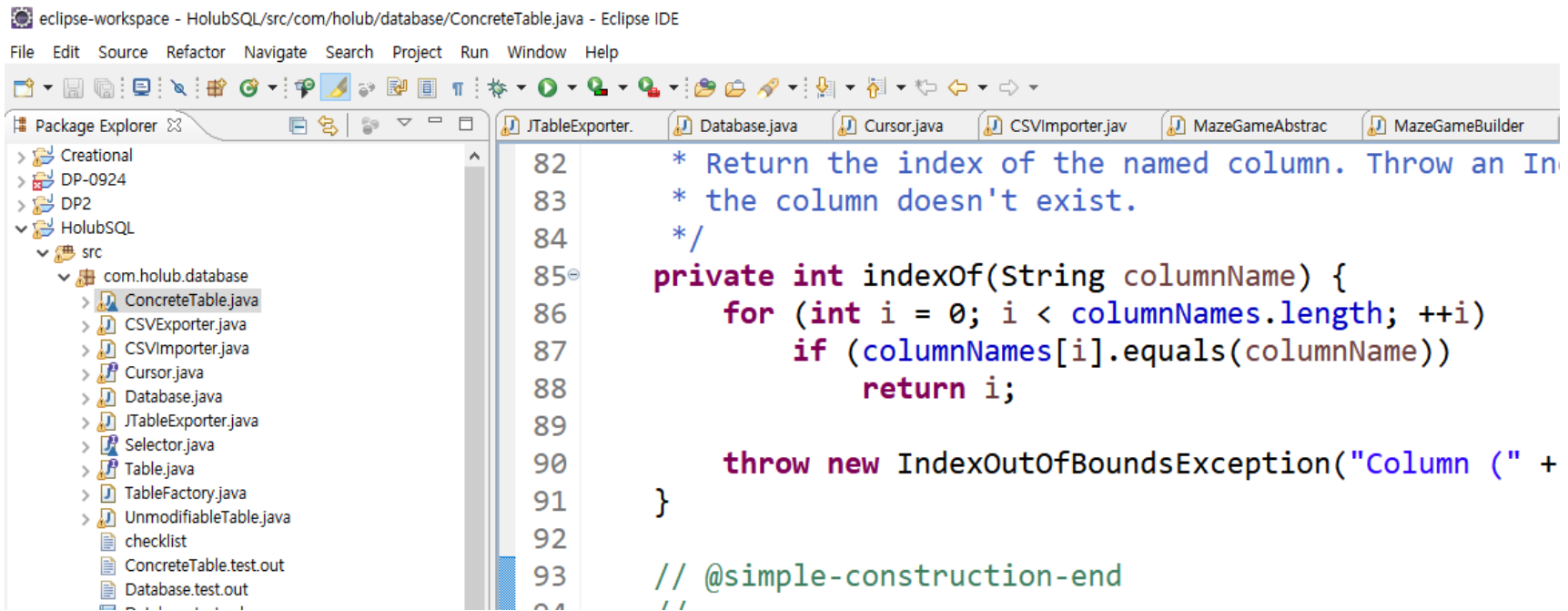
- Eclipse를 구동하여 File 메뉴의 Import를 선택하고 "Existing Projects into Workspace"를 선택하여 HolubSQL 프로젝트를 import





# \*실습환경 구축 Step 3: ConcreteTable 실행

- Package Explorer 탭에서 HolubSQL/src/com.holub.database/ConcreteTable.java를 더블클릭 후 메뉴에서 Run>Run 선택 혹은 단축키 ctrl+F11 로 Main.java 컴파일 및 실행



```
82      * Return the index of the named column. Throw an In
83      * the column doesn't exist.
84      */
85      private int indexOf(String columnName) {
86          for (int i = 0; i < columnNames.length; ++i)
87              if (columnNames[i].equals(columnName))
88                  return i;
89
90          throw new IndexOutOfBoundsException("Column (" +
91      )
92
93      // @simple-construction-end
94      //
```

Problems Console  
<terminated> ConcreteTable.Test (1) [Java Application] C:\Java\jdk1.8.0\_212\bin\javaw.exe (2020. 10. 13. 오후 4:09:34)

rollback(Table.THIS\_LEVEL) insert

people

last	first	addrId
------	-------	--------

-----

Holub	Allen	1
Flintstone	Wilma	2
Flintstone	Fred	2

실행화면 (마지막 부분)

# Practice 1 : Holub Database 코드 분석

- 연습 소스: HolubSQL/src/com.holub.database/ConcreteTable.java
- ConcreteTable을 비롯한 주요 소스코드 분석 및 테스트 실행
  - 각 패턴의 구현 코드 확인
  - ConcreteTable의 main 메소드 실행 및 결과 확인
  - 쿼리 문장 만들기

```
public final static class Test {  
    Table people = TableFactory.create("people", new String[] { "last", "first",  
"addrId" });  
    Table address = TableFactory.create("address", new String[] { "addrId", "street",  
"city", "state", "zip" });  
    public void test() {  
        testInsert(); testUpdate(); testDelete();  
        testSelect(); testStore(); testJoin();  
        testUndo();  
    }  
}
```

# Practice 2 : HTMLExporter 코딩 (빌더 패턴)

- 연습 소스: HolubSQL/src/com.holub.database/ConcreteTable.java, CSVExporter.java
- CSVExporter.java 를 참고하여 HTMLExporter.java를 작성하시오. 단, HTMLExporter는 CSVExporter와 유사한 동작을 하는데 그 포맷이 HTML인 점만 다름
- ConcreteTable.java 파일의 main 부분을 수정하여, 위에서 작성한 HTMLExporter 클래스를 테스트하시오. (testStore method 참고)

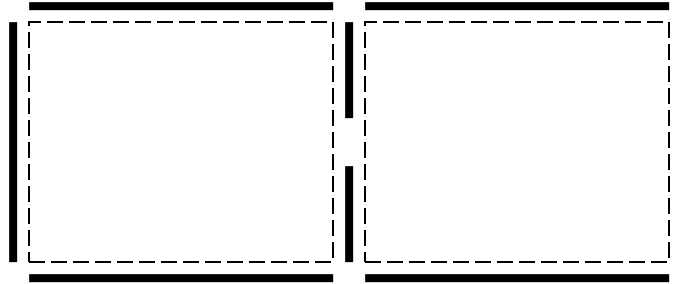
```
public void testStore() throws IOException, ClassNotFoundException {  
    Writer out = new FileWriter("people");  
    people.export(new CSVExporter(out));  
    out.close();  
    Reader in = new FileReader("people");  
    people = new ConcreteTable(new CSVImporter(in));  
    in.close();  
}
```

# **7. Factory Method vs Abstract Factory Pattern**

# Running Example

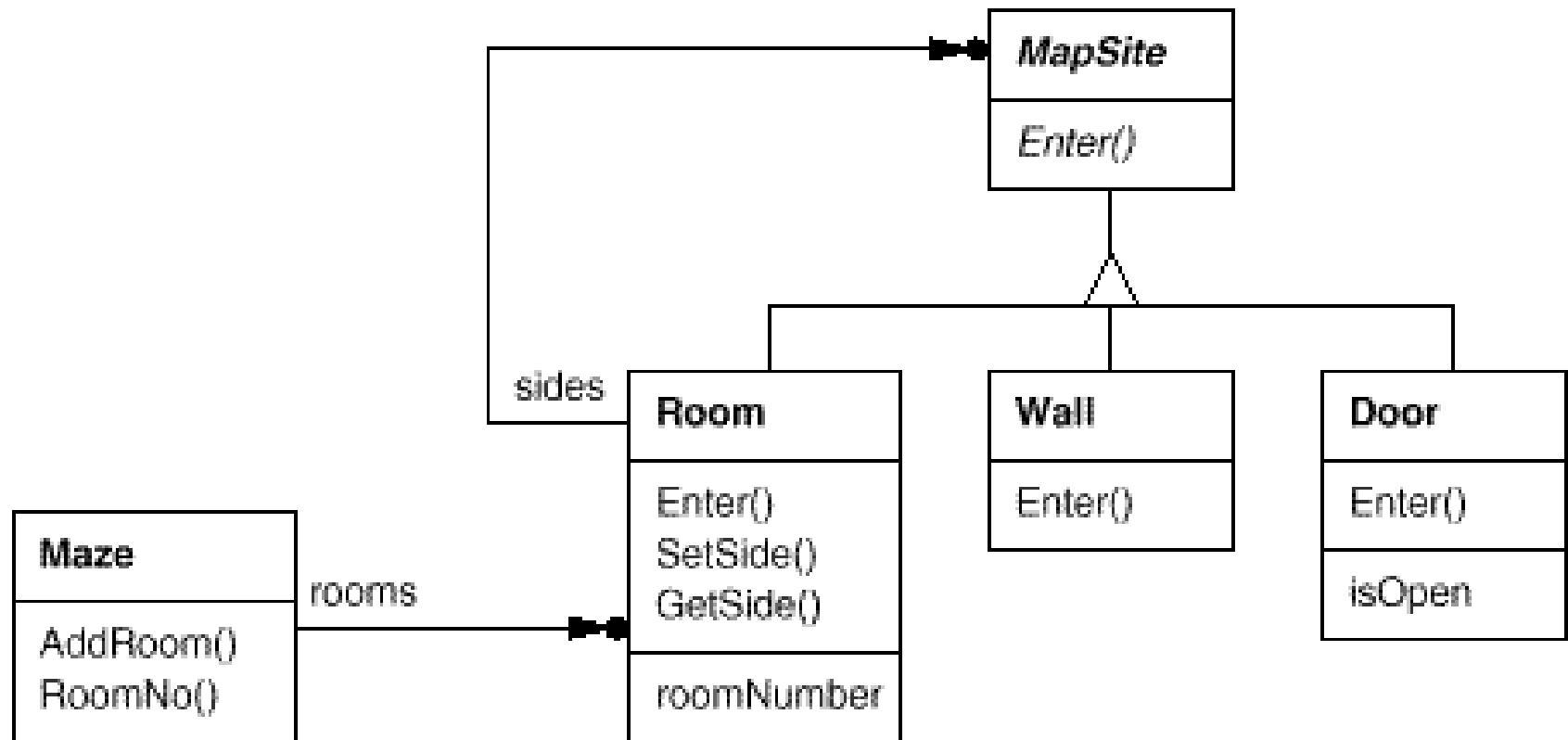
---

- Building a maze for a computer game.



- A Maze is a set of Rooms.
- A Room knows its neighbours.
  - another room
  - a wall
  - a door

# Maze Class Structure



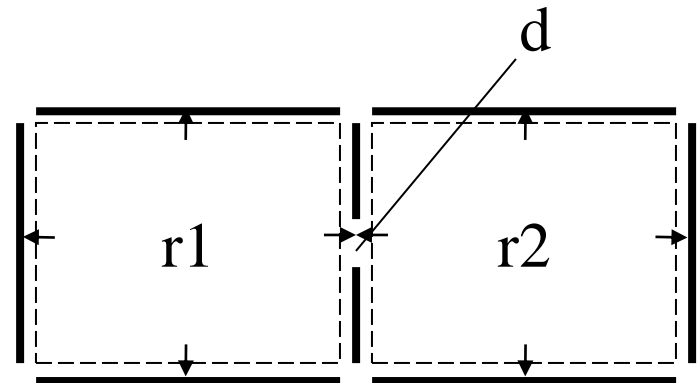
# Creating Mazes

```
public class MazeGame
{
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze();
    }
    public Maze createMaze() {
        Room r1 = new Room(1);
        Room r2 = new Room(2);
        Door d  = new Door(r1,r2);

        r1.setSide(Direction.North, new Wall());
        r1.setSide(Direction.East, d);
        r1.setSide(Direction.West, new Wall());
        r1.setSide(Direction.South, new Wall());

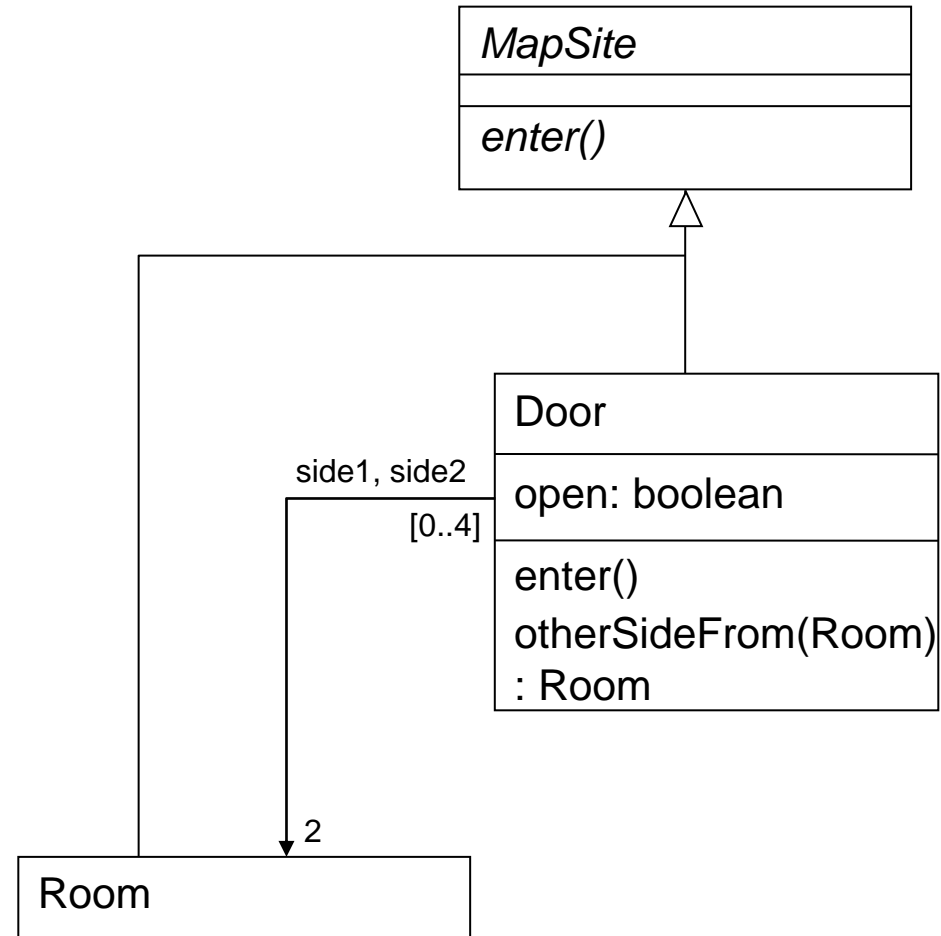
        r2.setSide(Direction.North, new Wall());
        r2.setSide(Direction.West, d);
        r2.setSide(Direction.East, new Wall());
        r2.setSide(Direction.South, new Wall());

        Maze m  = new Maze();
        m.addRoom(r1);
        m.addRoom(r2);
        return m;
    }
}
```



# Maze Classes

```
public class Door extends MapSite
{
    Door(Room s1, Room s2) {
        side1 = s1;
        side2 = s2;
    }
    public void enter() { }
    public Room otherSideFrom(Room r) {
        if( r == side1 ) return side2;
        else if( r == side2 ) return side1;
        else return null;
    }
    public void setOpen(boolean b) {
        open = b;
    }
    public boolean getOpen() {
        return open;
    }
    private Room side1;
    private Room side2;
    boolean open;
}
```





# Maze Classes

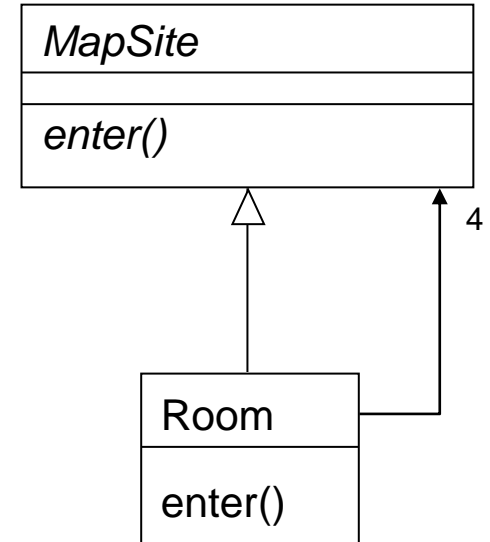
---

```
public class Direction
{
    public final static int First = 0;
    public final static int North = First;
    public final static int South = North+1;
    public final static int East = South+1;
    public final static int West = East+1;
    public final static int Last = West;
    public final static int Num = Last-First+1;
}
```

# Maze Classes

```
public class Room extends MapSite
{
    public Room(int r) { room_no = r; }
    public void enter() { }

    public void setSide(int direction, MapSite ms) {
        side[direction] = ms;
    }
    public MapSite getSide(int direction) {
        return side[direction];
    }
    public void setRoom_no(int r) {
        room_no = r;
    }
    public int getRoom_no() {
        return room_no;
    }
    private int room_no;
    private MapSite[] side = new MapSite[Direction.Num];
}
```



# Maze Classes

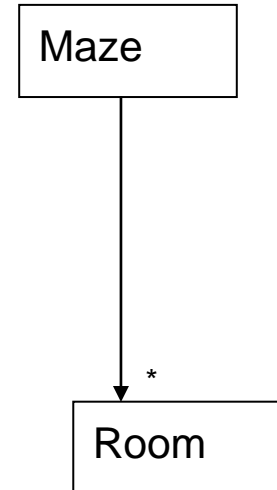
```
import java.util.Vector;

public class Maze
{
    public void addRoom(Room r) {
        rooms.addElement(r);
    }

    public Room getRoom(int r) {
        return (Room)rooms.elementAt(r);
    }

    public int numRooms() {
        return rooms.size();
    }

    private Vector rooms = new Vector();
}
```



# Maze Creation

---

```
public Maze createMaze() {
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door d   = new Door(r1,r2);

    r1.setSide(Direction.North, new Wall());
    r1.setSide(Direction.East,  d);
    r1.setSide(Direction.West,  new Wall());
    r1.setSide(Direction.South, new Wall());

    r2.setSide(Direction.North, new Wall());
    r2.setSide(Direction.East,  d);
    r2.setSide(Direction.West,  new Wall());
    r2.setSide(Direction.South, new Wall());

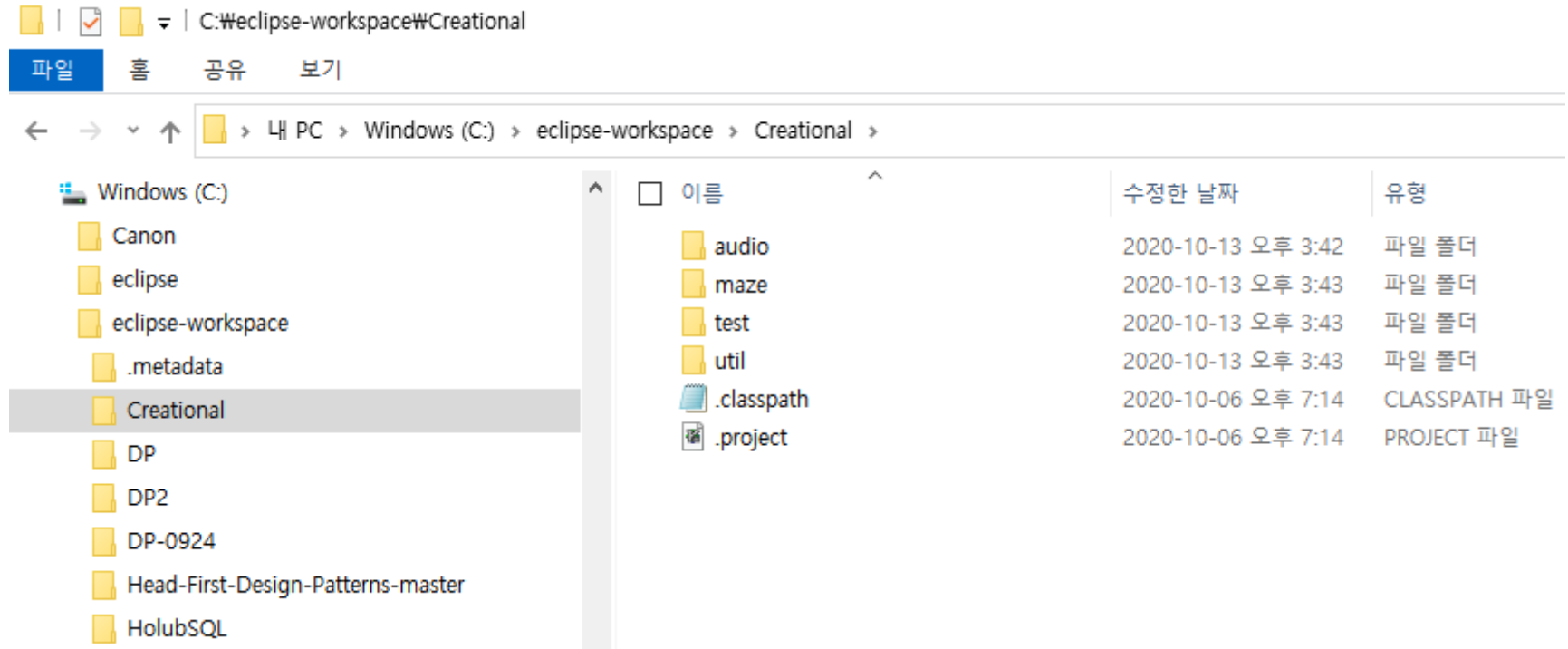
    Maze m  = new Maze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```

# Practice 1 : Factory Method와 Abstract Factory 패턴 이용 설계 확장

- 마법 미로(EnchantedMaze)를 가지는 새로운 게임을 만들기 위해 기존의 Maze 클래스를 재사용하고자 한다. 마법 미로는 다음과 같은 새로운 종류의 부품(component) 조합으로 구성된다.
  - DecoratedWall: 문양이 화려한 벽(Wall)
  - SpellDoor: 달힌 뒤로는 마법 주문에 의해서만 열리는 문(Door)
  - EnchantedRoom: 마법열쇠나 마법주문 등과 같은 여러 기이한 물품이 있는 방(Room)
- 추후 폭탄 미로(BombedMaze)를 가지는 게임도 추가될 예정이다. 폭탄 미로는 다음과 같은 새로운 종류의 부품 조합으로 구성된다.
  - BombedWall: 폭탄이 설치될 수 있는 벽(Wall)
  - BombProofDoor: 폭탄이 터져도 안전한 문(Door)
  - BombedRoom: 폭탄이 설치될 수 있는 방(Room)
- Task 1: 팩토리 메소드 (Factory Method) 설계패턴을 적용할 때 위 요구사항들을 손쉽게 지원할 수 있도록 createMaze() 메소드를 변경하시오.
- Task 2: 추상 팩토리 (Abstract Factory) 설계패턴을 적용할 때 위 요구사항들을 손쉽게 지원할 수 있도록 createMaze() 메소드를 변경하시오.

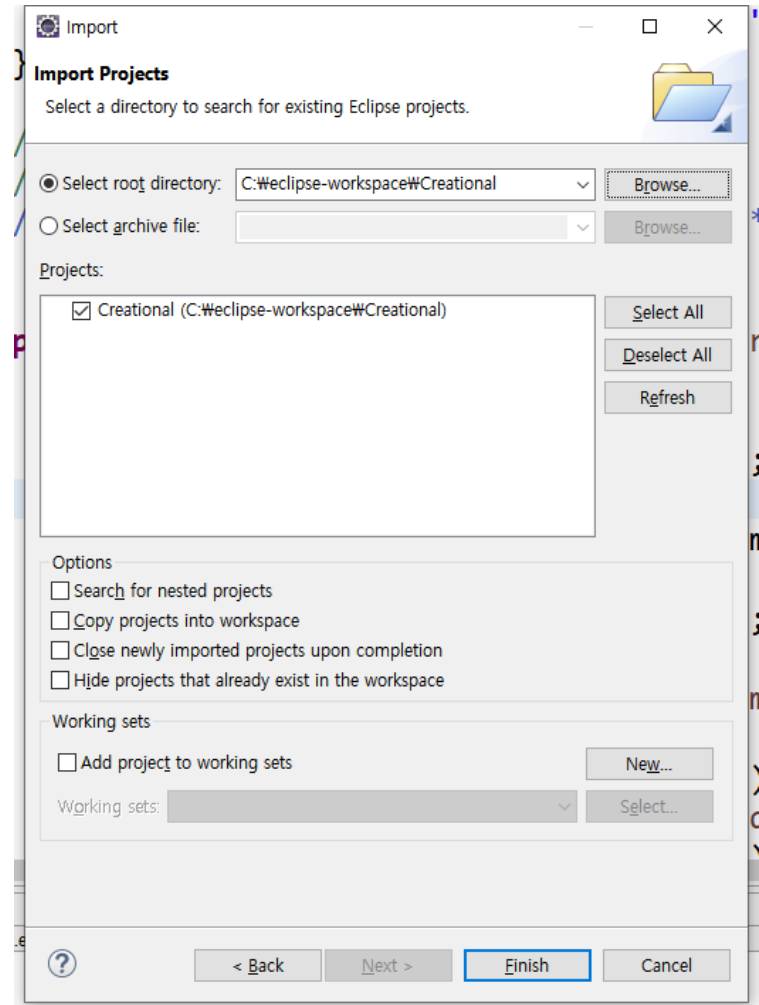
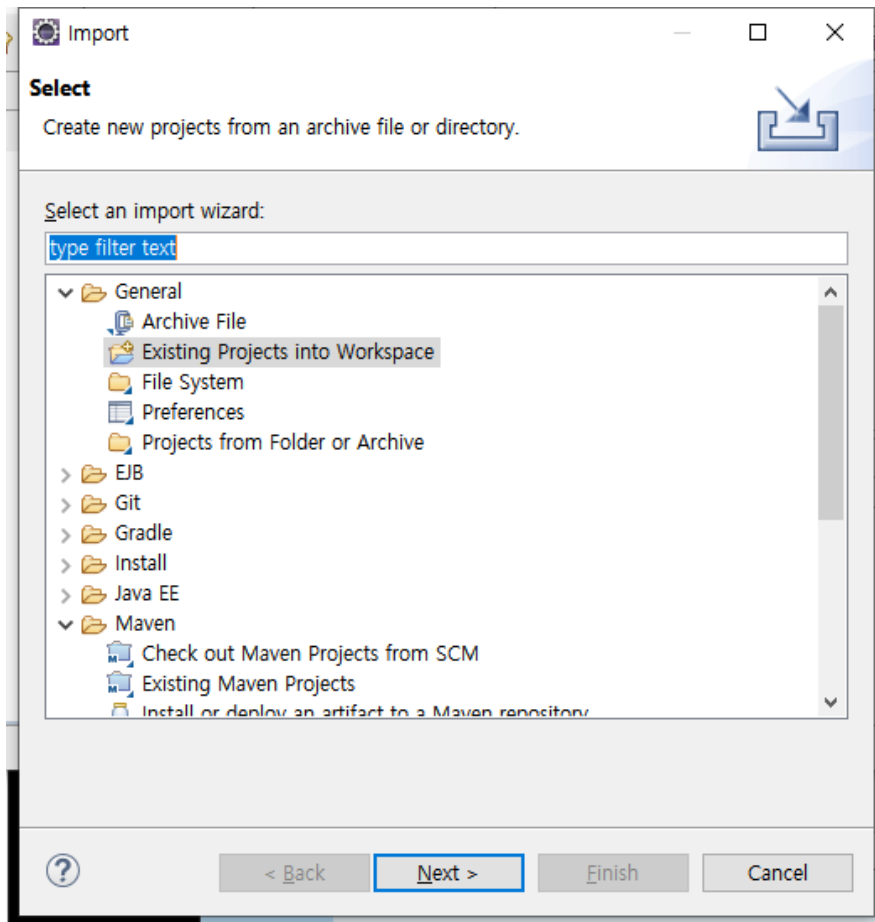
# \*실습환경 구축 Step 1: 실습용 Creational 예제 복사

- 배포된 "Creational.zip" 파일을 열고, Creational 디렉토리를 포함하여 workspace 디렉토리 아래에 압축해제



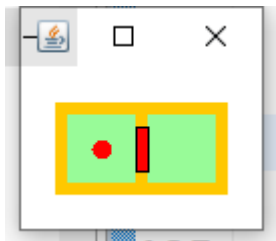
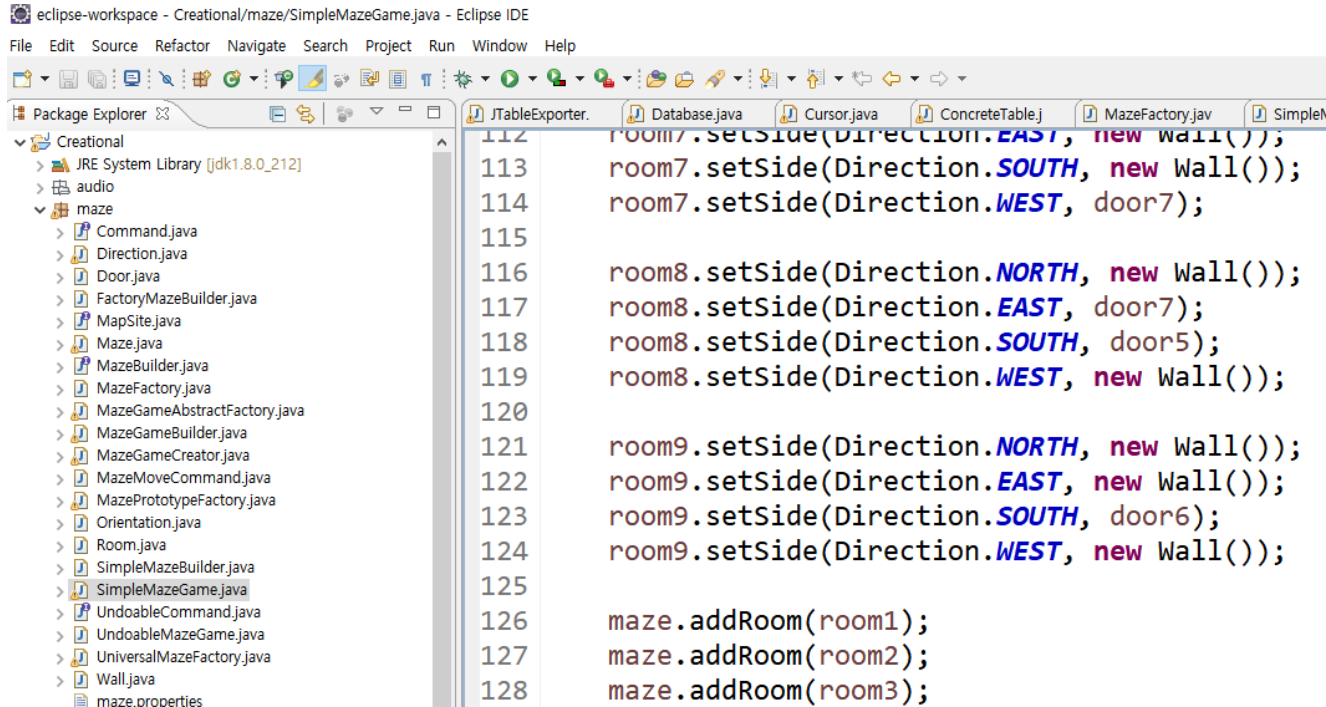
# \*실습환경 구축 Step 2: 이클립스에서 임포트

- Eclipse를 구동하여 File 메뉴의 Import를 선택하고 "Existing Projects into Workspace"를 선택하여 Creational 프로젝트를 import



# \*실습환경 구축 Step 3: SimpleMazeGame 실행

- Package Explorer 탭에서 Creational/maze/SimpleMazeGame.java를 더블클릭 후 메뉴에서 Run>Run 선택 혹은 단축키 ctrl+F11 로 Main.java 컴파일 및 실행

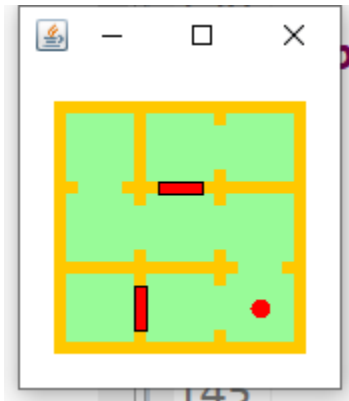
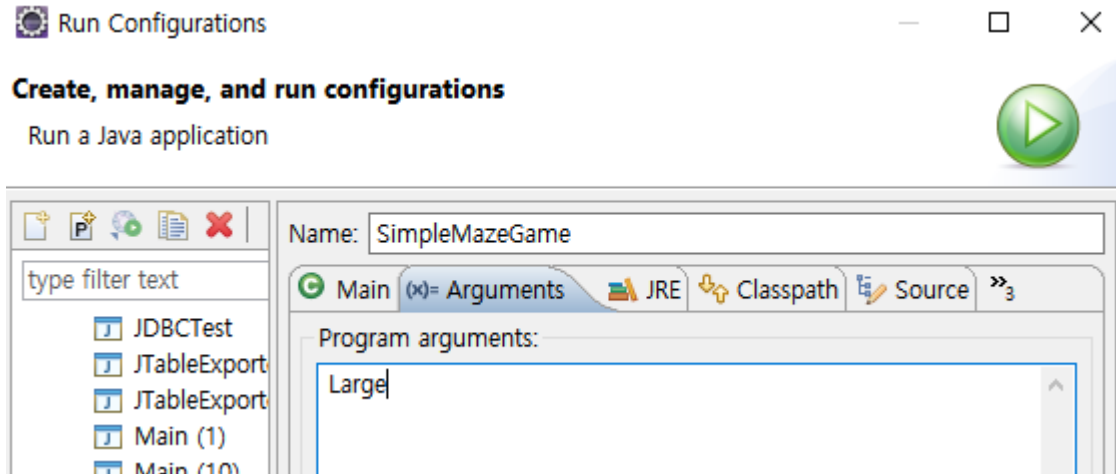


실행화면 (커서키로 이동 가능)



# \*실습환경 구축 Step 4: SimpleMazeGame 실행

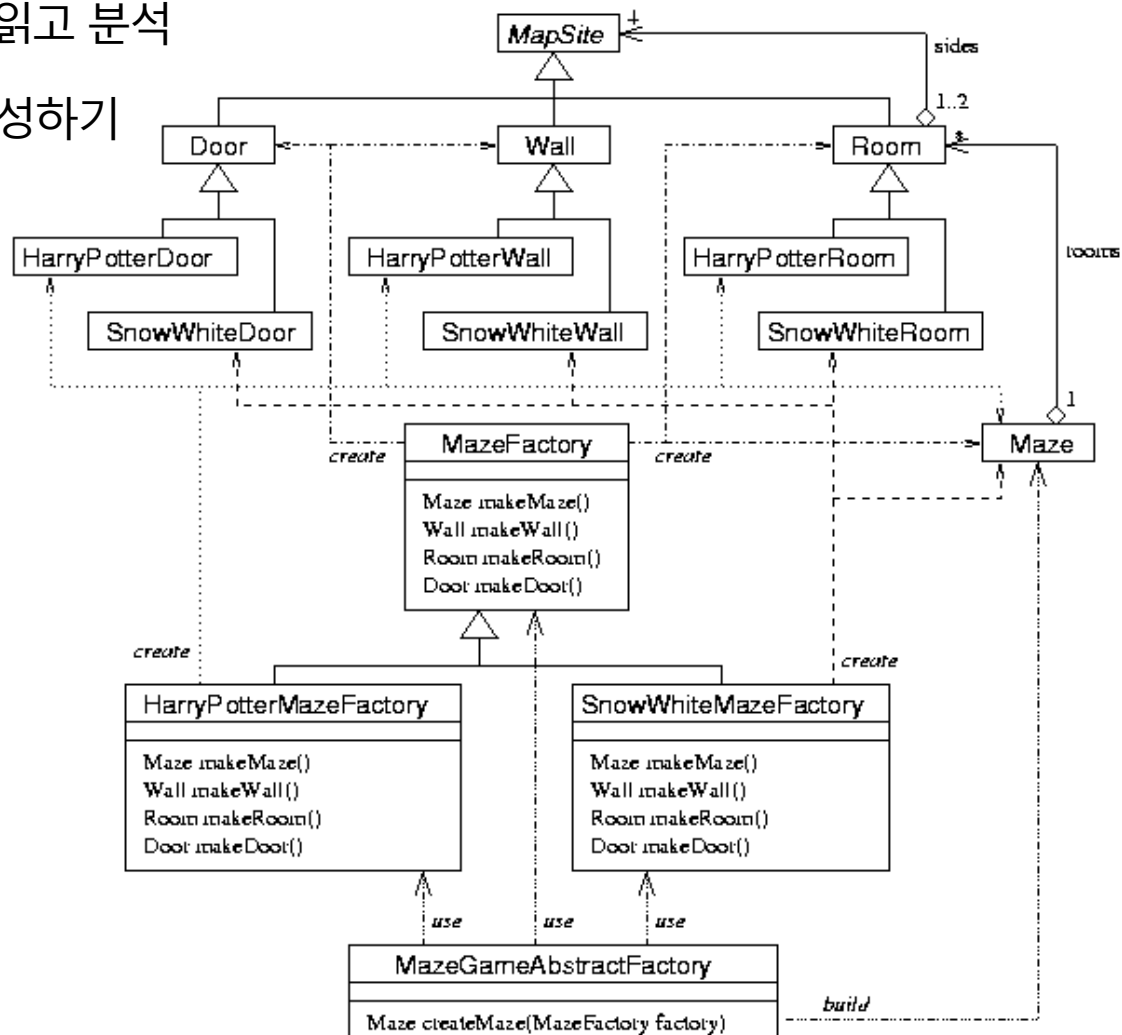
- 메뉴의 Run/RunConfiguration 을 선택하고 다음 화면에서 Arguments 탭으로 이동하여 Large라 고 입력하고 Run 버튼 클릭



실행화면 (커서키로 이동 가능)

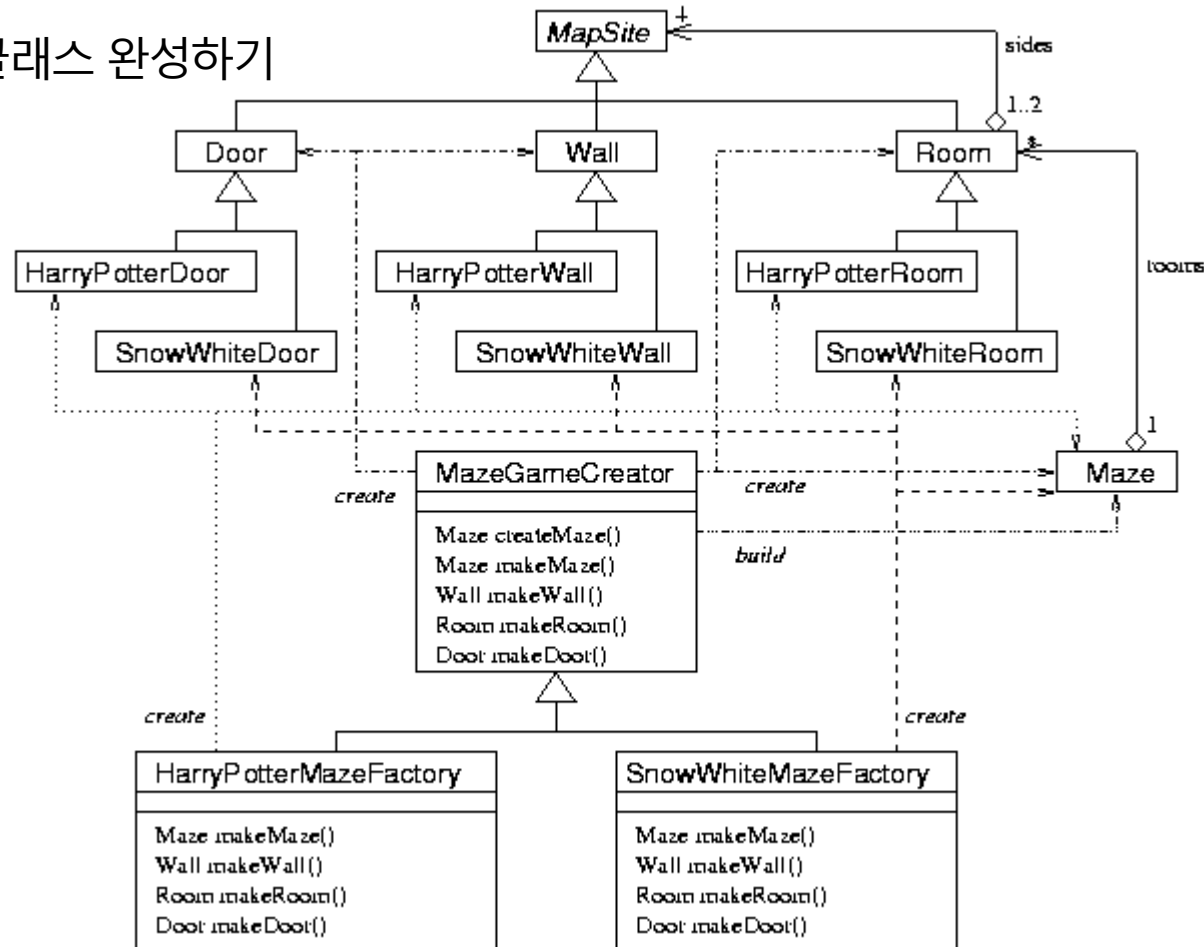
# Practice 2 : Abstract Factory 기반 Maze Game 코딩

- 연습 소스: Creational/maze/maze.snow/SnowWhiteMazeFactory.java
- HarryPotterMazeFactory 클래스를 읽고 분석
- SnowWhiteMazeFactory 클래스 완성하기



# Practice 3 : Factory Method 기반 Maze Game 코딩

- 연습 소스: Creational/maze/maze.snow/SnowWhiteMazeGameCreator.java
- HarryPotterMazeGameCreator 클래스를 읽고 분석
- SnowWhiteMazeGameCreator 클래스 완성하기



# Practice 4 : Prototype 기반 Maze Game 코딩

---

- 연습 소스: Creational/maze/MazePrototypeFactory.java
- MazePrototypeFactory 클래스를 읽고 분석
- MazePrototypeFactory 클래스의 main 메소드를 실행할 때 Argument에 따라 다음과 같이 동작하도록 코딩할 것
  - Argument 없이 실행되면 기본 (default) 부품으로 Maze가 만들어짐
  - Argument가 "Harry"인 경우 HarryPotter 테마의 Door, Room, Wall 부품으로 Maze가 만들어짐
  - Argument가 "Snow"인 경우 SnowWhite 테마의 Door, Room, Wall 부품으로 Maze가 만들어짐

# Practice 4 : Prototype 기반 Maze Game 코딩

