Chapter 04

디자인 패턴

(Design Pattern)

Contents

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴

◆ 게임에서 플레이어 클래스의 인스턴스가 두 개라면??

- ✓ 게임 로직 상 문제 발생 소지
- ✓ 자원의 불필요한 소비

◆ 특징

- ✓ 전역변수처럼 어디서나 인스턴스에 접근 가능
- ✓ 사용할 때 자원을 할당하고, 원할 때 자원 해제 가능

◆ 예제

- ✓ 전쟁터에 있는 한 부대의 지휘관이 두 명이 되는 것을 방지

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

```
public static soldier soldiers;

public class Commander  {
    public void orderAttack( )  {
            soldiers.gotoAttack( );
    }
    public void orderDefense( )  {
            soldiers.backtoDefense( );
    }
}
```

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

```
public void processWar  {
    Commander commander = new Commander( );

    while (bWar)  {
            if (공격해야 할 것 같으면)          commander.orderAttack( );
            else if (방어해야 할 것 같으면)     commander.orderDefense( );
    }
}
```

✓ 별 문제 없음

✓ 사용자가 지휘관 클래스의 인스턴스를 하나 더 생성한다면?

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

```
public void processWar  {
    Commander commander = new Commander( );
    Commander commander2 = new Commander( );

    while (bWar)  {
            if (공격해야 할 것 같으면)          commander.orderAttack( );
            else if (방어해야 할 것 같으면)     commander.orderDefense( );

            if (공격해야 할 것 같으면)          commander2.orderAttack( );
            else if (방어해야 할 것 같으면)     commander2.orderDefense( );
    }
}
```

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

- ✓ 지휘관이 두 명이 되면
  - 각 지휘관이 서로 다른 명령을 내려, 한 명은 공격을 한 명은 방어를 명령한다면 문제 발생
- ✓ 사용자가 지휘관 인스턴스의 임의 생성을 막기 위해 Commander 클래스의 생성자를 private으로 선언

```
public class Commander  {
    private Commander( )  {
    }

    public void orderAttack( )  { … …
    }
    public void orderDefense( )  { … …
    }
}
```

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

- ✓ 생성자가 private 으로 선언되면 new 연산자로 인스턴스 생성 불가능
- ✓ private 메서드는 클래스 내부에서 실행 가능

```
public class Commander  {
    private Commander( )  {
    }
    … …
    public static Commander CreateInstance( )  {
            return new Commander( );
    }
}
```

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

✓ processWar

```
public void processWar  {
    Commander commander = Commander.CreateInstance( ); //new Commander( );
    Commander commander2 = Commander.CreateInstance( ); //new Commander( );
    … …
}
```

딩

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

```
public class Commander  {
    private static Commander s_instance;
    private Commander( )  {
    }
    … …
    public static Commander CreateInstance( )  {
            if (s_instance == null)  s_instance = new Commander( );
            return s_instance;
    }
}
```

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 지휘관 클래스, Commander

```
public void processWar  {
    Commander commander = Commander.CreateInstance( );
    Commander commander2 = Commander.CreateInstance( ); // 같은 인스턴스가 대입

    while (bWar)  {
            if (공격해야 할 것 같으면)          commander.orderAttack( );
            else if (방어해야 할 것 같으면)     commander.orderDefense( );
    }
}
```

4. 1 디자인 패턴

❖ 싱글턴 (Singleton) 패턴 (cont.)

◆ 기본 싱글턴 패턴

```
public class Singleton  {
    private static Singleton s_instance;

    private Singleton( )  {
            … …
    }
    // 필요에 따라 추가적인 멤버 변수와 메서드
    public static Singleton CreateInstance( )  {
            if (s_instance == null)  s_instance = new Singleton( );
            return s_instance;
    }
}
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴

◆ 개요

- ✓ 게임을 개발하면서 동작은 다르지만, 서로 밀접한 관계를 가지는 여러 클래스에 대해 필요한 시점에 수행하는 클래스를 골라 사용하고자 할 때 사용하는 패턴

◆ RPG(Role Playing Game) 예

- ✓ 몬스터를 타겟팅하고 공격을 실행하면 Character의 Attack 메서드가 호출
- ✓ 현재의 무기에 따라 무기에 대한 동작(알고리즘)이 실행

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

```
class Character {
    private String weapon;
    … …
    void Attack( ) {
            if (weapon == “Fist”) {
                // 주먹으로 공격
                if (hasTargetMonster( ) == true) {
                        if (getTargetMonster( ).getDistance( ) <= 5) {
                            setAnimation(“punch”);
                            getTargetMonster( ).damage(2);
                            delayToReuse(0.1);
                        }
                }
            }
```

```
else if (weapon == “Sword”)  {
    // 검으로 공격
    displayWeapon(“Sword”);
    if (hasTargetMonster( ) == true)  {
            if (getTargetMonster( ).getDistance( ) <= 10)  {
                setAnimation(“swing”);
                getTargetMonster( ).damage(12);
                delayToReuse(0.3);
            }
    }
}
else if (weapon == “Bow”)  {
    // 활로 공격
    displayWeapon(“Bow”);
    if (hasTargetMonster( ) == true)  {
            if (getTargetMonster( ).getDistance( ) <= 70 &&
                getTargetMonster( ).getDistance( ) >= 10 )  {
                setAnimation(“shoot”);
                getTargetMonster( ).damage(8);
                delayToReuse(0.7);
            }
    }
}
```

4. 1 디자인 패턴

```
            else if (weapon == "Staff")  {
                // 지팡이로 공격
                displayWeapon("Staff");
                if (hasTargetMonster( ) == true)  {
                        if (getTargetMonster( ).getDistance( ) <= 8)  {
                            setAnimation("swing");
                            getTargetMonster( ).damage(3);
                            getTargetMonster( ).addStatus("Burning");
                            delayToReuse(0.8);
                        }
                }
            }
    } // Attack( )
    … …
}
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

- ✓ Attack( ) 메서드는 weapon에 저장된 값을 if, else 문에서 무기 이름과 비교하여 무기에 따른 동작을 실행
- ✓ 몇가지 무기를 추가한다고 가정하면

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

```
void Attack( ) {
        … …
        else if (weapon == “Magic Sword”) {
            // 마법검으로 공격
            displayWeapon(“Magic Sword”);
            if (hasTargetMonster( ) == true) {
                    if (getTargetMonster( ).getDistance( ) <= 10) {
                        setAnimation(“swing”);
                        getTargetMonster( ).damage(12);
                        getTargetMonster( ).addStatus(“Confuse”);
                        delayToReuse(0.3);
                    }
            }
        }
        … …
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

```
void Attack( ) {
        … …
        else if (weapon == “CrossBow”) {
            // 석궁으로 공격
            displayWeapon(“CrossBow”);
            if (hasTargetMonster( ) == true) {
                    if (getTargetMonster( ).getDistance( ) <= 50 &&
                        getTargetMonster( ).getDistance( ) >= 10) {
                        setAnimation(“shoot”);
                        getTargetMonster( ).damage(10);
                        delayToReuse(0.5);
                    }
            }
        }
        … …
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

- ✓ 계속 무기를 추가한다면? / 무기를 삭제한다면?
- ✓ 코드에서 변화하는 부분을 끝없는 if문으로 관리하는 것은 비효율적
- ✓ 변화하는 부분을 분리해서 관리 → 스트래티지 패턴 적용
- ✓ 스트래티지 패턴을 사용해서 무기를 하나씩 클래스화하고 관리

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

✓ 무기에 대한 인터페이스 IWeapon 생성

```
interface IWeapon  {
    void Attack(Character _char);
}
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

✓ Weapon_Sword 클래스 ; 검에 대한 클래스

```
class Weapon_Sword implements IWeapon  {
    @Override
    public void Attack (Character _char)  {
            // 검으로 공격
            _char.displayWeapon(this);
            if (_char.hasTargetMonster( ) == true)  {
                if (_char.getTargetMonster( ).getDistance( ) <= 10)  {
                        _char.setAnimation("swing");
                        _char.getTargetMonster( ).damage(12);
                        _char.delayToReuse(0.3);
                }
            }
    }
} // Weapon_Sword
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

- ✓ Weapon_Fist 클래스 ; 주먹 공격에 대한 클래스
- ✓ Weapon_Sword 클래스 ; 검 공격에 대한 클래스
- ✓ Weapon_Bow 클래스 ; 활 공격에 대한 클래스
- ✓ Weapon_Staff 클래스 ; 지팡이 공격에 대한 클래스
- ✓ Weapon_CrossBow 클래스 ; 석궁 공격에 대한 클래스

```
class Character  {
    private IWeapon weapon;      // private String weapon;
    … …
    Character( )  {
            weapon = new Weapon_Fist( );
    }
    … …
}
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

✓ Attack( ) 메서드의 수정

```
class Character {
    … …
    public void Attack( ) {
            weapon.Attack(this);
    }
    // 무기 바꾸기
    public void setWeapon (IWeapon _weapon) {
            weapon = _weapon;
    }
}
```

4. 1 디자인 패턴

❖ 스트래티지(Strategy) 패턴 (cont.)

◆ RPG(Role Playing Game) 예

✓ 사용

```
Character _character = new Character( );
_character.setWeapon(new Weapon_Bow( ));
_character.Attack( );
```

✓ 스트래티지 패턴

- 서로 행위만 다를뿐 밀접한 연관 관계를 가지는 여러 클래스를 사용할 때 유용
- 조건문의 나열된 형태일 때 유용

4. 1 디자인 패턴

❖ 스테이트(State) 패턴

◆ 개요

✓ 게임 상태와 관련되어 객체의 상태에 따른 행동을 클래스화해서 관리

◆ 게임의 상인(NPC; Non Player Character) 구현 예

✓ 게임 속 상인 설정

- 이 상인은 한 번에 한 명의 플레이어와만 거래할 수 있다. (제약)
- 상인은 한 가지의 물건만 팔고 하루에 들어오는 수량이 한정되어 있다. (제약)
- 상인은 플레이어가 물건을 구입하러 오기 전까지 대기하고 있다. (파는상태)
- 플레이어가 물건을 구입하면 그 물건에 따라 돈을 받는다. (거래상태)
- 물건이 없다면 받았던 돈을 플레이어에게 돌려주고 가게 문을 닫는다. (종료상태)

→ 세가지 상태

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 ; 상태

```
public class Merchant  {
    static final int STATE_IDLE = 0;
    static final int STATE_DEALING = 1;
    static final int STATE_CLOSESTORE = 2;

    int quantity;
    int state;

    Merchant( )  {
            state = STATE_IDLE;
            quantity = 10;
    }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 ; 플레이어와 상인

✓ 플레이어가 상인에게 접근하는 몇 가지 경우

- 플레이어는 상인에게 대화를 걸거나
- 플레이어는 상인에게 물건을 구입하거나
- 플레이어는 상인에게 물건을 팔거나

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 ; 플레이어와 상인

```
public class Merchant  {
    … …

    public void bytItem( )  {
            // 플레이어가 물건을 구입하려 합니다.
            if (state == STATE_IDLE)  {
                // 상인 : 물건을 구입하겠는가?
                state = STATE_DEALING;
            }
            else if (state == STATE_DEALING)  {
                // 상인 : 난 지금 거래 중이라네 잠시만 기다리게.
            }
            else if (state == STATE_CLOSESTORE)  {
                // … (가게 문이 닫힌 듯 하다.)
            }
    }
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 ; 플레이어와 상인

```
public void accostHim( ) {
        // 플레이어가 대화를 요청하려 합니다.
        if (state == STATE_IDLE) {
            // 상인 : 반갑네~ 손님이 오기 저까지 얘기 좀 나누세.
        }
        else if (state == STATE_DEALING) {
            // 상인 : 할 말이 있으면 거래가 끝난 뒤 하자구.
        }
        else if (state == STATE_CLOSESTORE) {
            // ... (가게 문이 닫힌 듯 하다.)
        }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 ; 플레이어와 상인

```
public void saleItem( ) {
        // 플레이어의 아이템을 매입하려 합니다.
        if (state == STATE_IDLE) {
            // 상인 : 물건을 판매하겠는가?
            state = STATE_DEALING;
        }
        else if (state == STATE_DEALING) {
            // 상인 : 난 지금 거래 중이라네 잠시만 기다리게.
        }
        else if (state == STATE_CLOSESTORE) {
            // ... (가게 문이 닫힌 듯 하다.)
        }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 ; 플레이어와 상인

- ✓ 상태별로 삭제, 추가, 관리는 물론 기능을 수정할 때도 어려움
- ✓ 무수히 반복되는 if, else 문을 해결하기 위해 스테이트 패턴 사용
- ✓ 상태를 개별적으로 관리

- ✓ 상인의 세가지 상태를 각각 클래스로 구성
- ✓ 슈퍼 클래스를 작성하고 상태 클래스가 상속
- ✓ 상태별로 추가해줄 동작을 메서드로 분류

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ IState 인터페이스 ; 슈퍼 클래스

✓ 세가지 상태를 나타내는 클래스

```
public interface IStae  {
    // 플레이어가 대화를 요청하려 합니다.
    public void accostHim( );

    // 플레이어가 물건을 구입하려 합니다.
    public void buyItem( );

    // 플레이어의 아이템을 매입하려 합니다.
    public void saleItem( );
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ 상태 클래스 ; 물건 판매 대기 중 상태

```
public class IdleState implements IState  {
    @Override
    public void accosHim( )  {
            // 상인 : 반갑네~ 손님이 오기 저까지 얘기 좀 나누세.
    }
    @Override
    public void buyItem( )  {
            // 상인 : 물건을 구입하겠는가?
            state = STATE_DEALING;  // error
    }
    @Override
    public void saleItem( )  {
            // 상인 : 물건을 판매하겠는가?
            state = STATE_DEALING;  // error
    }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ IdleState 클래스 수정

- ✓ 상인 클래스의 인스턴스를 멤버 변수로 선언
- ✓ 상인의 상태를 현재 상태에서 다른 상태로 변경 가능하도록 구현

```
public class IdleState implements IState  {
    private Merchant m_merchant;

    public IdleState(Merchant _merchant)  {
            m_merchant = _merchant;
    }
    … …
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 수정

✓ 상인 클래스에 상태를 바꿔주는 메서드를 추가

```
public class Merchant  {
    int quantity;

    Merchant( )  {
            quantity = 10;
    }
    public void buyItem( )  {  // 플레이어가 물건을 구입하려 합니다.  }
    public void accostHim( )  {  // 플레이어가 대화를 요청하려 합니다.  }
    public void saleItem( )  {  // 플레이어의 아이템을 매입하려 합니다.  }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 수정

- ✓ 상인 클래스에 상태를 나타내는 인터페이스 IState를 멤버 변수로 추가
- ✓ 상인 자신의 인스턴스를 넘겨주는 코드 작성
- ✓ 각 상태에 대한 getter 메서드 작성

4. 1

```
public class Merchant  {
    // 현재의 상태
    IState m_state;

    // 상태별 인스턴스
    IState m_idleState;
    IState m_dealState;
    IState m_closeState;
    int quantity;

    Merchant( )  {
            m_idleState = new IdleState(this);
            m_dealState = new DealState(this);
            m_closeState = new CloseState(this);
            m_state = m_idleSate;
            quantity = 10;
    }
    public IState getIdleState ( )  {  return m_idleSate;  }
    public IState getDealState ( )  {  return m_dealSate;  }
    public IState getCloseState ( )  {  return m_closeSate;  }
    … …
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ Merchant 클래스 수정

✓ 상태를 바꿔주는 changeState 메서드 작성 (setter)

```
public class Merchant  {
    … …

    public void changeState(IState _state)  {
            m_state = _state;
    }
    … …
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ IdleSate 클래스 수정

- ✓ 상태 전환 코드 작성

4. 1 디자인 패턴

```
public class IdleState implements IState  {
    … …

    @Override
    public void accosHim( )  {
            // 상인 : 반갑네~ 손님이 오기 저까지 얘기 좀 나누세.
    }
    @Override
    public void buyItem( )  {
            // 상인 : 물건을 구입하겠는가?
            // state = STATE_DEALING;  // error
            m_merchant .changeState( m_merchant .getDealSate( ));
    }
    @Override
    public void saleItem( )  {
            // 상인 : 물건을 판매하겠는가?
            //state = STATE_DEALING;  // error
            m_merchant .changeState( m_merchant .getDealSate( ));
    }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ 스테이트 패턴의 사용

✓ 상인 클래스로 이동해서 동작별 메서드 코드 작성

```
public class Merchant  {
    … …
    public void buyItem( )  {
            // 플레이어가 물건을 구입하려 합니다.
            m_state .buyItem( );
}
    public void accostHim( )  {
            // 플레이어가 대화를 요청하려 합니다.
            m_state .accosHim( );
    }
    public void saleItem( )  {
            // 플레이어의 아이템을 매입하려 합니다.
            m_state .saleItem( );
    }
}
```

4. 1 디자인 패턴

❖ 스테이트(State) 패턴 (cont.)

◆ 스테이트 패턴

- ✓ 상태에 따른 동작 처리를 간편하게, 상태 값에 대한 관리도 용이

◆ 스트래티지 패턴과의 차이점

- ✓ 클라이언트(client)와 콘텍스트(context) 사이의 연관 관계
- ✓ 게임에서의 예
  - ▪ 스트래티지 패턴은 캐릭터 자신이 자산의 판단에 의해 필요한 무기로 변경
  - ▪ 스테이트 패턴은 캐릭터 자신의 판단이 아닌 무기(State)의 판단에 따라 무기 자체가 변경

◆ 적용 예

- ✓ 이전 짝맞추기 게임에서 그림 처리나 입력 처리를 게임의 상태에 따라 if, else 문을 사용해서 처리 → 이 게임의 상태를 스테이트 패턴으로 적용 가능

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴

◆ 개요

- ✓ 게임에서 인스턴스를 동적으로 생성할 때 유용한 패턴

◆ 전략 시뮬레이션 게임 예

- ✓ 수많은 객체를 동적으로 생성

```
public class Unit  {
    public int m_x;  // 유닛의 x 좌표
    public int m_y;  // 유닛의 y 좌표
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 전략 시뮬레이션 게임 예

✓ 유닛 10개가 각기 다른 정보를 갖고 생성되는 코드 예

```
public void SetUnit( ) {
    Unit _unit1 = new Unit( );
    _unit1. m_x = 0;
    _unit1. m_y = 0;
    Unit _unit1 = new Unit( );
    _unit1. m_x = 0;
    _unit1. m_y = 0;
    Unit _unit1 = new Unit( );
    _unit1. m_x = 0;
    _unit1. m_y = 0;
    Unit _unit1 = new Unit( );
    _unit1. m_x = 0;
    _unit1. m_y = 0;
    … … // 같은 코드
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 전략 시뮬레이션 게임 예

✓ 유닛의 종류가 세 가지라면

```
public class Unit  {
    public int m_x;              // 유닛의 x 좌표
    public int m_y;              // 유닛의 y 좌표
    public int m_sharp;          // 유닛의 모양
    public int hp;               // 유닛의 체력
}
```

```
public void SetUnit( )  {
    Unit _unit1 = new Unit( );
    _unit1. m_x = 0;
    _unit1. m_y = 0;
    _unit1. m_sharp = 0;
    _unit1. hp = 100;
    … … // 같은 코드
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 전략 시뮬레이션 게임 예

✓ 유닛 각각에 고유한 특징을 추가하면

- Unit_1 : 모든 공격을 방어합니다.
- Unit_2 : 모든 적을 공격합니다.
- Unit_3 : 아군을 치료합니다.

→ Unit 슈퍼 클래스를 만들고 각 Unit 별로 상속받아 사용하도록

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ Unit 클래스

```
public class Unit  {
    public int m_x;                 // 유닛의 x 좌표
    public int m_y;                 // 유닛의 y 좌표
    public int m_sharp;             // 유닛의 모양
    public int hp;                  // 유닛의 체력
    public int shield = 0;          // 추가 확장성 고려
    public int laserenergy = 0;     // 추가 확장성 고려

    void specialAttack( )  {  }
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 각 Unit 클래스

```
public class Unit_1 extends Unit  {
    @Override
    void SpecialAttack( )  {
            // Unit_1의 특수 기술
            // 없음
    }
}
```

```
public class Unit_2 extends Unit  {
    @Override
    void SpecialAttack( )  {
            // Unit_2의 특수 기술
            // 방어막을 펼쳐 hp 대신 shield 수치가 낮아짐. 0이 되면 hp가 닳음
    }
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 각 Unit 클래스

```
public class Unit_3 extends Unit  {
    @Override
    void SpecialAttack( )  {
            // Unit_3의 특수 기술
            // laserenergy를 소모하여 특수 공격을 함
    }
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 적군 생성 코드

✓ CreateUnit 메서드 ; 적군 생성

```
public void CreateUnit( ) {
    Unit _unit1 = new Unit_1( );
    _unit1. m_x = 0;
    _unit1. m_y = 0;
    _unit1. m_sharp = 0;
    _unit1. hp = 100;
    … … // 같은 코드
}
```

→ 팩토리 패턴 적용

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ UnitFactory 클래스

```
public class UnitFactory  {

    public Unit CreateUnit( int type, int x, int y)  {
            Unit _unit = null;

            // 유닛의 종류에 따른 생성 방법
            if (type == 1)  {
                _unit = new Unit_1( );
                _unit. m_sharp = 0;
                _unit. hp = 100;
            }
            if (type == 2)  {
                _unit = new Unit_2( );
                _unit. m_sharp = 1;
                _unit. hp = 50;
            }
```

4. 1 디자인 패턴

```
            if (type == 3)  {
                _unit = new Unit_3( );
                _unit. m_sharp = 2;
                _unit. hp = 250;
                _unit. laserenergy = 50;
            }

            // 공통 부분에 대한 코드
            _unit. m_x = x;
            _unit. m_y = y;

            return _unit;
        }
    }
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ Unit 생성 부분

```
public void setUnit( ) {
    UnitFactory unitFactory = new UnitFactory( );

    Unit _unit1  = unitFactory.CreateUnit(1, 0   , 0   );
    Unit _unit2  = unitFactory.CreateUnit(2, 30  , 20  );
    Unit _unit3  = unitFactory.CreateUnit(1, 60  , 40  );
    Unit _unit4  = unitFactory.CreateUnit(3, 80  , 120);
    Unit _unit5  = unitFactory.CreateUnit(2, 45  , 20  );
    Unit _unit6  = unitFactory.CreateUnit(3, 32  , 70  );
    Unit _unit7  = unitFactory.CreateUnit(1, 90  , 40  );
    Unit _unit8  = unitFactory.CreateUnit(1, 140, 10  );
    Unit _unit9  = unitFactory.CreateUnit(2, 120, 40  );
    Unit _unit10 = unitFactory.CreateUnit(3, 100, 20  );
}
```

4. 1 디자인 패턴

❖ 팩토리(Factory) 패턴 (cont.)

◆ 팩토리 패턴

- ✓ 생성에 대한 부하를 줄일 수 있음