Chapter 09 - 객체 지향 설계 원칙

생성일	@2025년 11월 22일 오후 4:17
태그	

01) 단일 책임 원칙 (SRP)

SOLID 원칙이란?

- 로버트 C.마틴이 2000년대 초반에 발표힌 객체지향 설계의 다섯 가지 원칙
- 마이클 C.페더스가 부르기 쉽게 머리글자로 소개한 것
- **SOLID**
  - **S**ingle Responsibility Principle (SRP) : 단일 책임 원칙
  - **O**pen-Closed Principle (OCP) : 개방 폐쇄 원칙
  - **L**iskov Substitution Principle (LSP) : 리스코프 치환 원칙
  - **I**nterface Segregation Principle (ISP) : 인터페이스 분리 원칙
  - **D**ependency Inversion Principle (DIP) : 의존성 역전 원칙

단일 책임 원칙 (SRP)

- '클래스는 한 가지 기능만 수행해야 하고, 한 가지 이유로만 변경해야 한다'는 원칙
- **산탄총 수술**
  - 한 가지 기능을 수행할 때 클래스를 여러 개 수행해야 한다면 유지, 보수성은 떨어지기 마련
  - 이러한 현상을 '**산탄총 수술**'이라고 함 (탄환을 사방에 남긴다는 의미)
- **클래스 추출**
  - 단일 책임 원칙을 적용하는 구체적인 방법 : 리팩터링
  - 상속 관계보다는 컴포지션, 어그리게이션 적극 활용

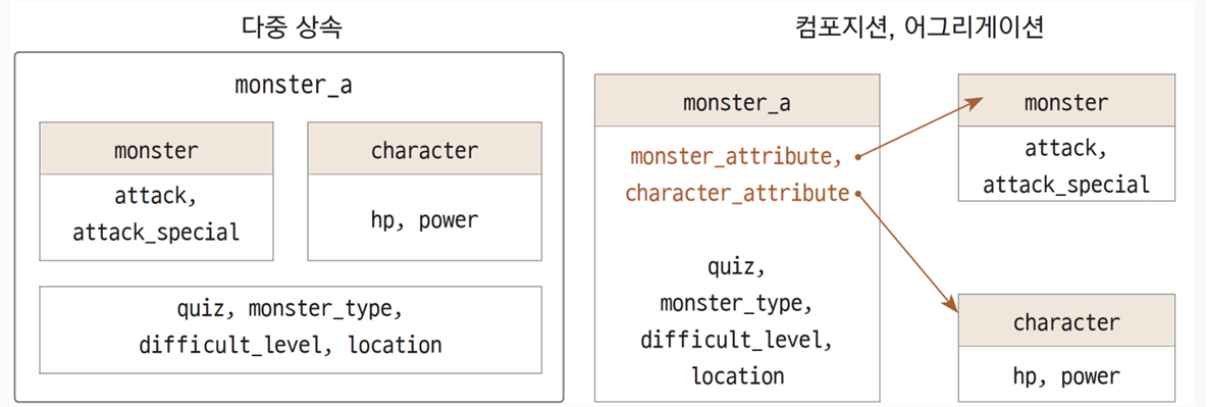


그림 다중 상속 vs 컴포지션, 어그리게이션

- 거대 클래스를 작은 단위로 분할, 분할된 클래스를 포함하여 이전 논리적인 관계 유지
- 기존 거대 클래스는 여러 가지 기능을 묶는 역할, 추출한 하위 클래스를 **has-a 관계**로 설정

02) 개방, 폐쇄 원칙 (OCP)

확장에 열려 있고, 수정에 닫혀 있다

- **'확장에 열려 있고, 수정에 닫혀 있어야 한다.'**
  - 동적 바인딩이 개방, 폐쇄 원칙을 잘 설명
    - 새로운 기능을 언제든 추가할 수 있음 (확장에 개방)
    - 다른 코드에 파급 효과가 없어 추가되는 기능 외에는 수정이 필요 없음 (수정에 폐쇄)
  - 커피 전문점 모델링
    - 커피 전문점에서 음료 주문 방법은 음료의 종류와 무관하게 동일한
      - 주문하는 메뉴에 따라 제조 방법만 다름
      - 연말 특별 상품이 추가되어도 고객과 점원 간 주문 방식이 변경되지 않음

추상 클래스 활용

- 개방, 폐쇄 원칙은 추상 클래스 (인터페이스)로 구현 가능

- 기능의 흐름은 추상 클래스를 활용해 직접 상속받아 구현하는 클래스에 세부 동작 구현
- 흐름의 뼈대는 템플릿으로 만들고 살을 붙이는 작업은 자식 클래스에 위임
- 이러한 방식으로 설계하는 패턴을 **템플릿 메서드 패턴**이라고 함

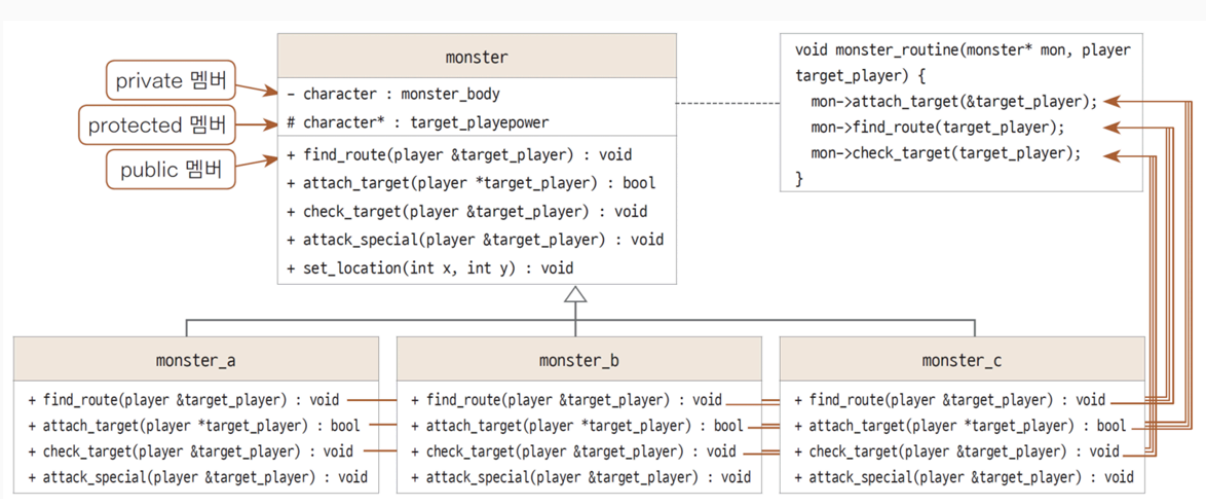


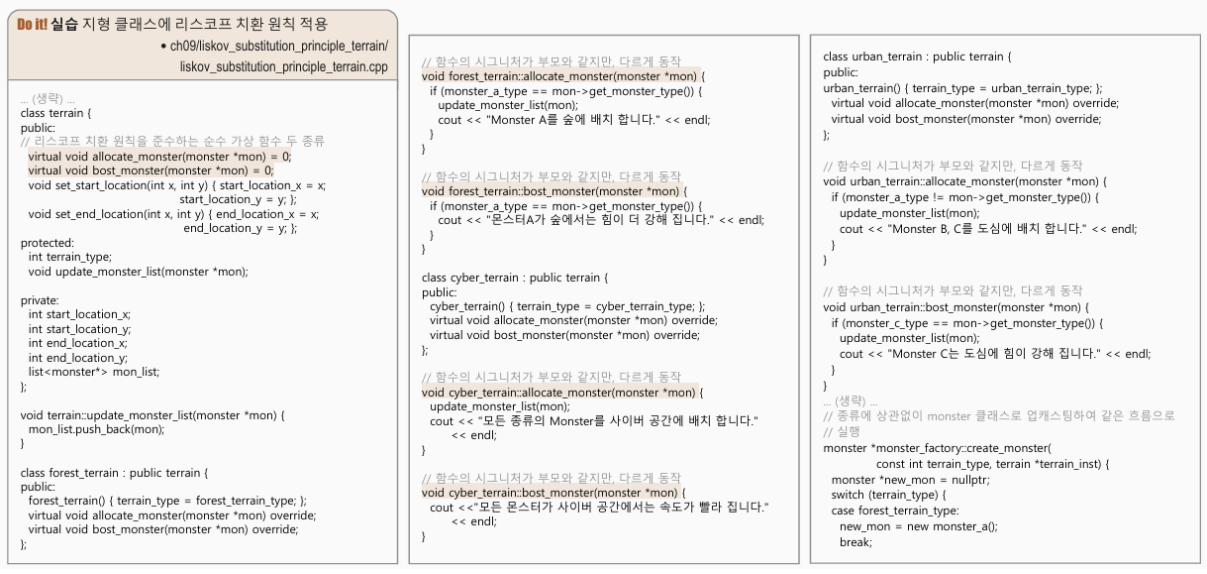
그림 템플릿 메서드 패턴으로 설계한 예

03) 리스코프 치환 원칙 (LSP)

리스코프 치환 원칙 (LSP)

- **'하위 클래스는 상위 클래스를 대체할 수 있어야 한다'**는 의미
  - 다형성의 동작 원리를 설명
- 자식 클래스가 부모 클래스를 치환
  - 부모 클래스의 역할을 자식 클래스가 수행할 수 있음
  - 자식 클래스가 부모 클래스를 완전히 대체할 수 있는 **'is-a'** 관계
- is-a 관계로 정의된 클래스는 리스코프 치환 원칙에 따르는 클래스
  - 다음의 두 가지를 의미

1. **부모 클래스를 상속받아 구현한 자식 클래스는 부모 클래스로 업캐스팅이 가능하다**
2. **자식 클래스에서 부모 클래스의 멤버 변수를 상속받아 오버라이딩하거나 유지해야 한다**



Do it! 실습 지형 클래스에 리스코프 치환 원칙 적용

• ch09/liskov_substitution_principle_terrain/liskov_substitution_principle_terrain.cpp

```
... (생략) ...
class terrain {
public:
// 리스코프 치환 원칙을 준수하는 순수 가상 함수 두 종류
  virtual void allocate_monster(monster *mon) = 0;
  virtual void bost_monster(monster *mon) = 0;
  void set_start_location(int x, int y) { start_location_x = x;
                                          start_location_y = y; };
  void set_end_location(int x, int y) { end_location_x = x;
                                        end_location_y = y; };
protected:
  int terrain_type;
  void update_monster_list(monster *mon);

private:
  int start_location_x;
  int start_location_y;
  int end_location_x;
  int end_location_y;
  list<monster*> mon_list;
};

void terrain::update_monster_list(monster *mon) {
  mon_list.push_back(mon);
}

class forest_terrain : public terrain {
public:
  forest_terrain() { terrain_type = forest_terrain_type; };
  virtual void allocate_monster(monster *mon) override;
  virtual void bost_monster(monster *mon) override;
};
```

```
// 함수의 시그니처가 부모와 같지만, 다르게 동작
void forest_terrain::allocate_monster(monster *mon) {
  if (monster_a_type == mon->get_monster_type()) {
    update_monster_list(mon);
    cout << "Monster A를 숲에 배치 합니다." << endl;
  }
}

// 함수의 시그니처가 부모와 같지만, 다르게 동작
void forest_terrain::bost_monster(monster *mon) {
  if (monster_a_type == mon->get_monster_type()) {
    cout << "몬스터A가 숲에서는 힘이 더 강해 집니다." << endl;
  }
}

class cyber_terrain : public terrain {
public:
  cyber_terrain() { terrain_type = cyber_terrain_type; };
  virtual void allocate_monster(monster *mon) override;
  virtual void bost_monster(monster *mon) override;
};

// 함수의 시그니처가 부모와 같지만, 다르게 동작
void cyber_terrain::allocate_monster(monster *mon) {
  update_monster_list(mon);
  cout << "모든 종류의 Monster를 사이버 공간에 배치 합니다."
       << endl;
}

// 함수의 시그니처가 부모와 같지만, 다르게 동작
void cyber_terrain::bost_monster(monster *mon) {
  cout <<"모든 몬스터가 사이버 공간에서는 속도가 빨라 집니다."
       << endl;
}
```

```
class urban_terrain : public terrain {
public:
urban_terrain() { terrain_type = urban_terrain_type; };
  virtual void allocate_monster(monster *mon) override;
  virtual void bost_monster(monster *mon) override;
};

// 함수의 시그니처가 부모와 같지만, 다르게 동작
void urban_terrain::allocate_monster(monster *mon) {
  if (monster_a_type != mon->get_monster_type()) {
    update_monster_list(mon);
    cout << "Monster B, C를 도심에 배치 합니다." << endl;
  }
}

// 함수의 시그니처가 부모와 같지만, 다르게 동작
void urban_terrain::bost_monster(monster *mon) {
  if (monster_c_type == mon->get_monster_type()) {
    update_monster_list(mon);
    cout << "Monster C는 도심에 힘이 강해 집니다." << endl;
  }
}
... (생략) ...
// 종류에 상관없이 monster 클래스로 업캐스팅하여 같은 흐름으로
// 실행
monster *monster_factory::create_monster(
            const int terrain_type, terrain *terrain_inst) {
  monster *new_mon = nullptr;
  switch (terrain_type) {
  case forest_terrain_type:
    new_mon = new monster_a();
    break;
```

```
  case cyber_terrain_type:
    new_mon = new monster_b();
    break;
  case urban_terrain_type:
    new_mon = new monster_c();
    break;
  }

  terrain_inst->allocate_monster(new_mon);
  terrain_inst->bost_monster(new_mon);
  mon_list.push_back(new_mon);
  mon_count++;
  return new_mon;
  }
... (생략) ...

int main() {
  int mon_count, i;
  player target_player_dummy;

  target_player_dummy.set_location(dis(gen), dis(gen));
  monster_factory::create_terrain();
  monster_factory::initialize_monster();
```

```
  mon_count = monster_factory::get_monster_count();
  for (i = 0; i < mon_count; ++i) {
    cout << endl;
    monster_routine(monster_factory::get_monster(i),
                    target_player_dummy);
  }

  monster_factory::destroy_monster();
  monster_factory::destroy_terrain();
  return 0;
}
```

실행 결과(출력 결과는 랜덤)

```
Monster A를 숲에 배치 합니다.
몬스터A가 숲에서는 힘이 더 강해 집니다.
Monster B, C를 도심에 배치 합니다.
... (생략) ...
```

04) 인터페이스 분리 원칙 (ISP)

인터페이스 분리 원칙 (ISP)

- ‘인터페이스는 작고 섬세해야 하고, 클래스는 필요한 인터페이스만 구현해야 한다’
- 단일 책임 원칙 (SRP)과 연관해서
  - SRP - ‘클래스는 한 가지 기능만 수행해야 하고, 한 가지 이유로만 변경해야 한다’
  - 클래스가 여러 인터페이스를 상속 받으면 SRP를 위반 하게 됨
  - 인터페이스가 여러 목적을 정의 하여도 역시 SRP를 위반하게 됨
- 다시 인터페이스 분리 원칙(ISP)으로
  - 인터페이스는 반드시 최소의 기능만 정의(작고 섬세)해야 함
  - 클래스는 최소한의 인터페이스만을 상속받아서 구현해야 함
  - 여러 기능이 필요한 클래스라면 ‘has-a’로 직접 구현이 아닌 멤버로 포함해야 함

Do it! 실습 인터페이스 분리 원칙 적용

• ch09/ISP_monster_example/ISP_monster_example.cpp

```
... (생략) ...
// 인터페이스로 사용할 추상 클래스
class IRoute {
public:
  virtual void find_route(int x, int y) = 0;
  virtual void set_location(int x, int y) = 0;
  virtual int get_location(bool x) = 0;
};

// 인터페이스로 사용할 추상 클래스
class IAttack {
public:
  virtual bool attach_target(character* target_player) = 0;
  virtual void check_target(character& target_player) = 0;
  virtual void attack_special(character& target_player) = 0;
};

// character 클래스와 추상 클래스 IRoute 상속
class player : public character, public IRoute {
public:
  player();
  virtual void find_route(int x, int y) override;
  virtual void set_location(int x, int y) override;
  virtual int get_location(bool x) override;
private:
  int location_x;
  int location_y;
};
... (생략)...

// IRoute, IAttack 추상 클래스를 상속받아 정의
class monster : public IRoute, public IAttack {
public:
  int get_monster_type() { return monster_type; };
  virtual void set_location(int x, int y) override
                              { location_x = x;
                                location_y = y; };
  virtual int get_location(bool x) override
                    { return x ? location_x : location_y; };
protected:
  int calculate_distance(int x, int y);
  character *target_player = nullptr;
  int monster_type;
  character monster_body;

private:
  int location_x;
  int location_y;
};
... (생략) ...
class npc_object : public IRoute {
public:
  virtual void find_route(int x, int y) override;
  virtual void set_location(int x, int y) override;
  virtual int get_location(bool x) override;

private:
  int location_x; int location_y;
};
... (생략)...

// 작은 범위로 정의된 여러 추상 클래스를 상속받아
// 정의한 함수 사용
void monster_routine(monster *mon, player target_player) {
  mon->attack_target(&target_player);
  mon->find_route(target_player.get_location(true),
                  target_player.get_location(false));
  mon->check_target(target_player);
}
... (생략) ...

int main() {
  int mon_count, i;
  player target_player_dummy;

  target_player_dummy.set_location(dis(gen), dis(gen));
  character_factory::create_terrain();
  character_factory::initialize_monster();

  mon_count = character_factory::get_monster_count();
  for (i = 0; i < mon_count; ++i) {
  cout << endl;
  monster_routine(character_factory::get_monster(i),
                  target_player_dummy);
  }

  character_factory::destroy_monster();
  character_factory::destroy_terrain();
  return 0;
}
```

실행 결과(출력 내용은 다를 수 있음)

```
Monster A를 숲에 배치 합니다.
몬스터A가 숲에서는 힘이 더 강해 집니다.
Monster A를 숲에 배치 합니다.
... (생략) ...
```

예제에서 OCP 확인하기

- 기본 몬스터 클래스에 있던 순수 가상 함수 분리
  - 이동 인터페이스인 IRoute, 공격 인터페이스 IAttack로 각각 분리
  - 몬스터, NPC 이동 인터페이스 활용, 공격이 필요하지 않은 NPC는 공격 인터페이스를 구현하지 않음

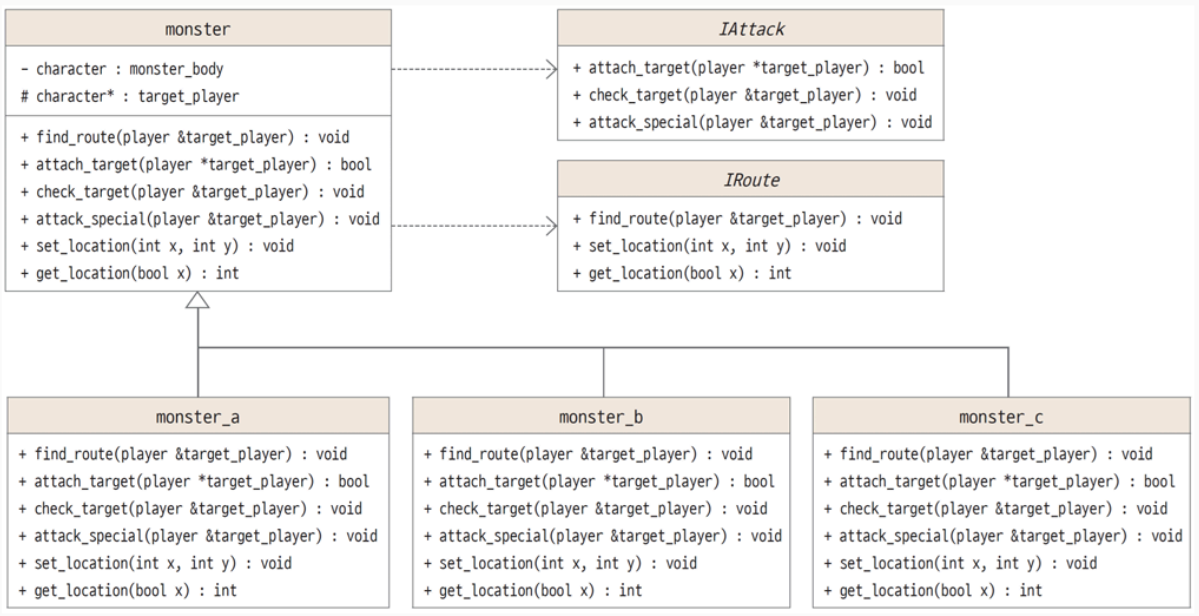


그림 인터페이스 분리 원칙이 적용된 몬스터 클래스 다이어그램

05) 의존성 역전 원칙 (DIP)

의존성 역전 원칙 (DIP)

- '상위 수준 모듈은 하위 수준의 모듈에 의존해서는 안 되며 상위, 하위 수준 모두 추상 레이어 (인터페이스)에 의존해야 한다'
- DIP 고려 없는 설계 예제
  - 무기와 탈 것 모두 인터페이스가 아닌 구상 클래스에 직접 의존(사용)하는 예제
  - 따라서 무기나 탈 것을 추가하거나 삭제할 때에 플레이어 클래스를 수정해야 함

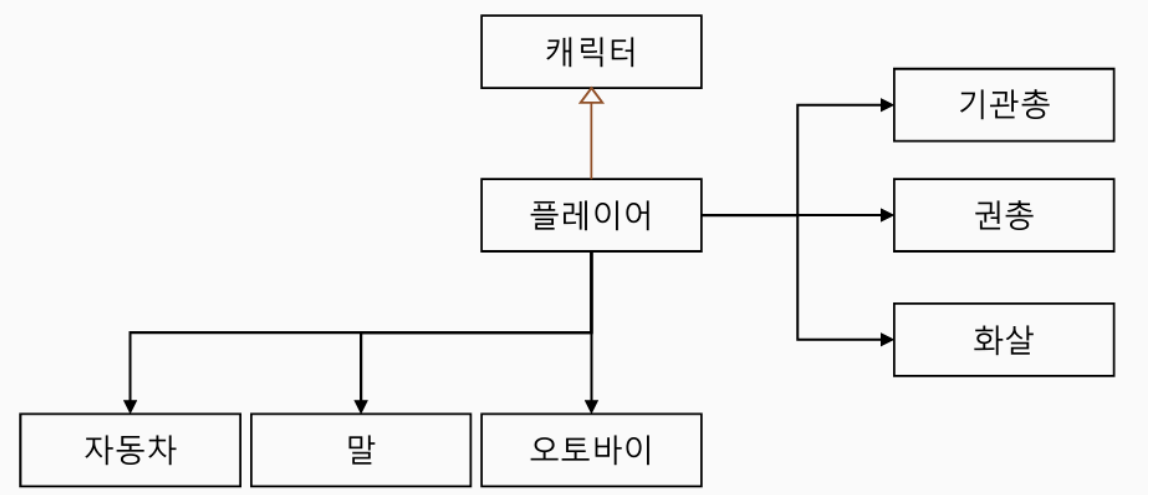


그림 클래스로 플레이어 설계(의존성 역전 원칙 미적용)

- DIP가 고려된 설계 예제
  - 무기와 탈 것이 추상 계층, 즉 인터페이스를 거쳐서 플레이어가 사용하도록 설계

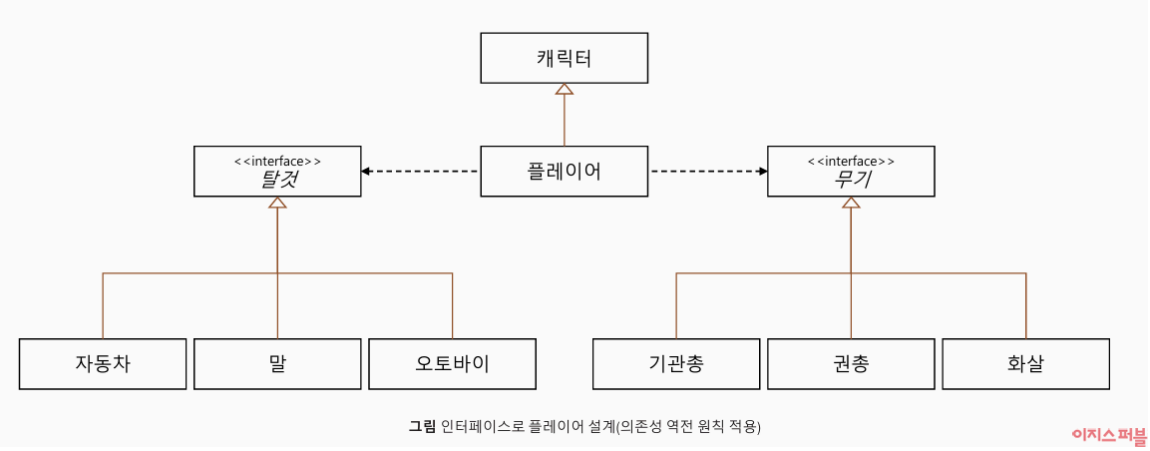


그림 인터페이스로 플레이어 설계(의존성 역전 원칙 적용)

이지스퍼블

Do it! 실습 의존성 역전 원칙 적용

• ch09/DIP_monster_example/DIP_monster_example.cpp

```
... (생략) ...
// 인터페이스로 사용할 클래스 IWeapon
class IWeapon {
public:
  virtual void reload_bullet() = 0;
  virtual bool is_bullet_empty() = 0;
  virtual void shoot_weapon(void* target_plaery) = 0;
};

// 인터페이스로 사용할 추상 클래스 IRiding_object
class IRiding_object {
public:
  virtual int check_energy() = 0;
  virtual void set_destination(int x, int y) = 0;
  virtual void run_to_destination() = 0;
};
... (생략) ...

class player : public character, public IRoute {
public:
  virtual void find_route(int x, int y) override;
  virtual void set_location(int x, int y) override;
  virtual int get_location(bool x) override;
  void set_weapon(IWeapon *new_weapon)
                          { weapon = new_weapon; };
  void set_riding_object(IRiding_object *new_riding_object) {
                    riding_object = new_riding_object; };
  void release_weapone() { weapon = nullptr; };
  void release_riding_object() { riding_object = nullptr; };
  void *get_weapon() { return weapon; };
  void *get_riding_object() { return riding_object; };
private:
  int location_x;
  int location_y;
```

```
  // 클래스가 아닌 인터페이스에 의존
  IWeapon *weapon = nullptr;
  IRiding_object *riding_object = nullptr;
};
... (생략) ...

class gun : public IWeapon { ... (생략) ... };
class machine_gun : public IWeapon { ... (생략) ... };
class arrow : public IWeapon { ... (생략) ... };
... (생략) ...

class car : public IRiding_object { ... (생략) ... };
class horse : public IRiding_object { ... (생략) ... };
class motor_cycle : public IRiding_object { ... (생략) ... };
... (생략) ...

void monster::dependence_strike_back(void *target_player) {
  player *target_player_inst = (player*)target_player;
  IWeapon *weapon = (IWeapon*)target_player_inst->get_weapon();
  IRiding_object *riding_object =
      (IRiding_object*)target_player_inst->get_riding_object();
  if (nullptr == weapon) {
    return;
  }

  if (weapon->is_bullet_empty()) {
    weapon->reload_bullet();
  }
  weapon->shoot_weapon(this);

  if (riding_object->check_energy() > 10) {
    riding_object->set_destination(get_location(true) + 30,
                                   get_location(false) + 30);
```

반격을 막아내는 함수. 무기와 탈것은 모두 인터페이스로 처리 인터페이스를 상속한 클래스가 무엇이든 이 함수는 변경되지 않음

```
    riding_object->run_to_destination();
  }
}
... (생략) ...

int main() {
  int mon_count, i;
  player target_player;
  car riding_car;
  machine_gun m_gun;

  random_device rd;
  mt19937 gen(rd());
  uniform_int_distribution<int> dis(0, 99);

  target_player.set_riding_object(&riding_car);
  target_player.set_weapon(&m_gun);
  target_player.set_location(dis(gen), dis(gen));

  character_factory::create_terrain();
  character_factory::initialize_monster();

  mon_count = character_factory::get_monster_count();
  for (i = 0; i < mon_count; ++i) {
    cout << endl;
    monster_routine(character_factory::get_monster(i),
                    target_player);
  }

  character_factory::destroy_monster();
  character_factory::destroy_terrain();
return 0;
}
```

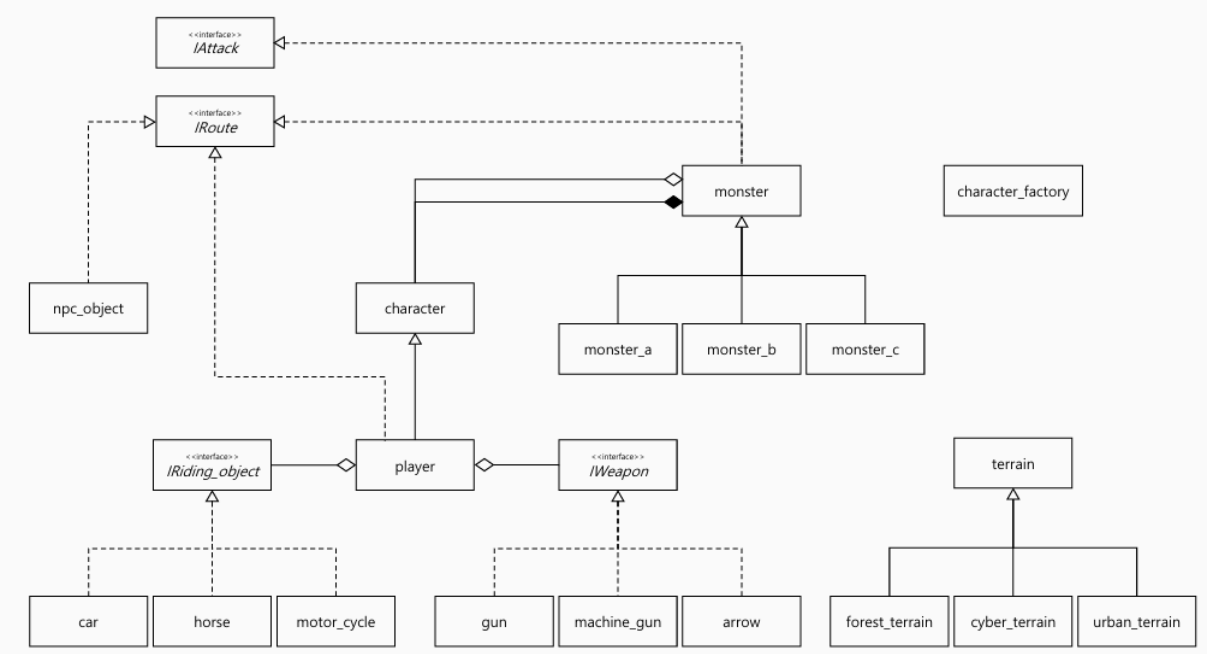


그림 의존성 역전 원칙이 적용된 플레이어의 클래스 다이어그램