게임 프로그래밍 패턴 2

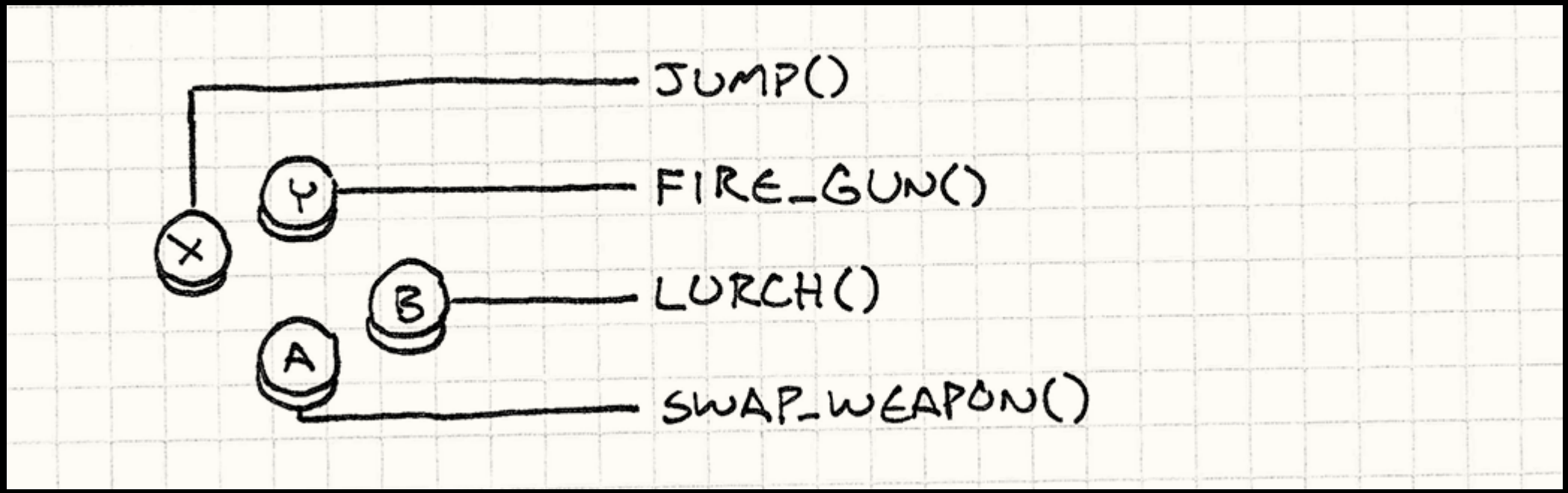
쿠재아이
김재경

목차

명령
Command

입력키 변경

- 그림을 코드로 구현해 보자



입력키 변경

```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) jump();
    else if (isPressed(BUTTON_Y)) fireGun();
    else if (isPressed(BUTTON_A)) swapWeapon();
    else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

입력키 변경

- 만약 게임 도중 키를 변경해야 한다면?
- 점프를 X -〉 Y로 바꾼다던지…

예제 코드

- 공통 상위 클래스를 정의한다

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
};
```

예제 코드

- 각 행동별로 하위 클래스를 만든다

```
class JumpCommand : public Command
{
public:
    virtual void execute() { jump(); }
};

class FireCommand : public Command
{
public:
    virtual void execute() { fireGun(); }
};
```

예제 코드

- 입력 핸들러에는 각 버튼별로 Command 클래스 포인터를 저장한다

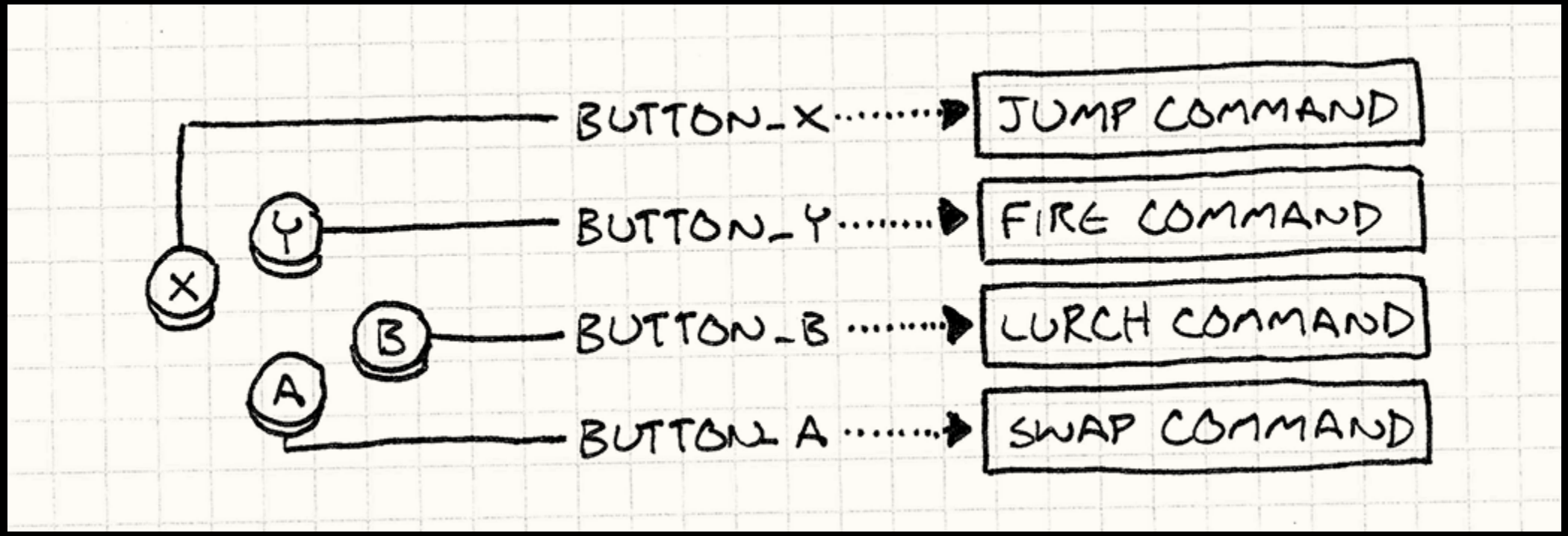
```
class InputHandler
{
public:
    void handleInput();
    // 명령을 바인드할 함수들

private:
    Command* buttonX_;
    Command* buttonY_;
    Command* buttonA_;
    Command* buttonB_;
};
```

예제 코드

```
void InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) buttonX_->execute();
    else if (isPressed(BUTTON_Y)) buttonY_->execute();
    else if (isPressed(BUTTON_A)) buttonA_->execute();
    else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

결과

- 한 겹 우회하는 계층이 생겼다



좀 더 유연하게

- 현재 JumpCommand 클래스는 오직 플레이어 캐릭터만 점프하게 만들 수 있다

- 이런 제약을 유연하게 만들기 위해 제어하려는 객체를 함수에서 직접 찾게 하지 말고 밖에서 전달해주자

좀 더 유연하게

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute(GameActor& actor) = 0;
};

class JumpCommand : public Command
{
public:
    virtual void execute(GameActor& actor)
    {
        actor.jump();
    }
};
```

좀 더 유연하게

```
Command* InputHandler::handleInput()
{
    if (isPressed(BUTTON_X)) return buttonX_;
    else if (isPressed(BUTTON_Y)) return buttonY_;
    else if (isPressed(BUTTON_A)) return buttonA_;
    else if (isPressed(BUTTON_B)) return buttonB_;

    return nullptr;
}
```

```
Command* command = inputHandler.handleInput();
if (command)
{
    command->execute(actor);
}
```

결과

- 명령을 실행할 때 액터만 바꾸면 플레이어가 게임에 있는 어떤 액터라도 제어할 수 있게 되었다

- 함수를 직접 호출하는 형태의 강한 커플링을 제거하여 명령을 큐나 스트림으로 만드는 것도 가능하다

실행 취소와 재실행

- 게임에서 이동 취소 기능을 추가한다고 해보자

```
class MoveUnitCommand : public Command
{
public:
    MoveUnitCommand(Unit* unit, int x, int y)
    : unit_(unit), x_(x), y_(y)
    {

    }

    virtual void execute()
    {
        unit_->moveTo(x_, y_);
    }

private:
    Unit* unit_;
    int x_;
    int y_;
};
```

실행 취소와 재실행

```
Command* handleInput()
{
    Unit* unit = getSelectedUnit();
    if (isPressed(BUTTON_UP))
    {
        // 유닛을 한 칸 위로 이동한다
        int destY = unit->y() - 1;
        return new MoveUnitCommand(unit, unit->x(), destY);
    }
    if (isPressed(BUTTON_DOWN))
    {
        // 유닛을 한 칸 아래로 이동한다
        int destY = unit->y() + 1;
        return new MoveUnitCommand(unit, unit->x(), destY);
    }
    // 다른 이동들...
    return nullptr;
}
```

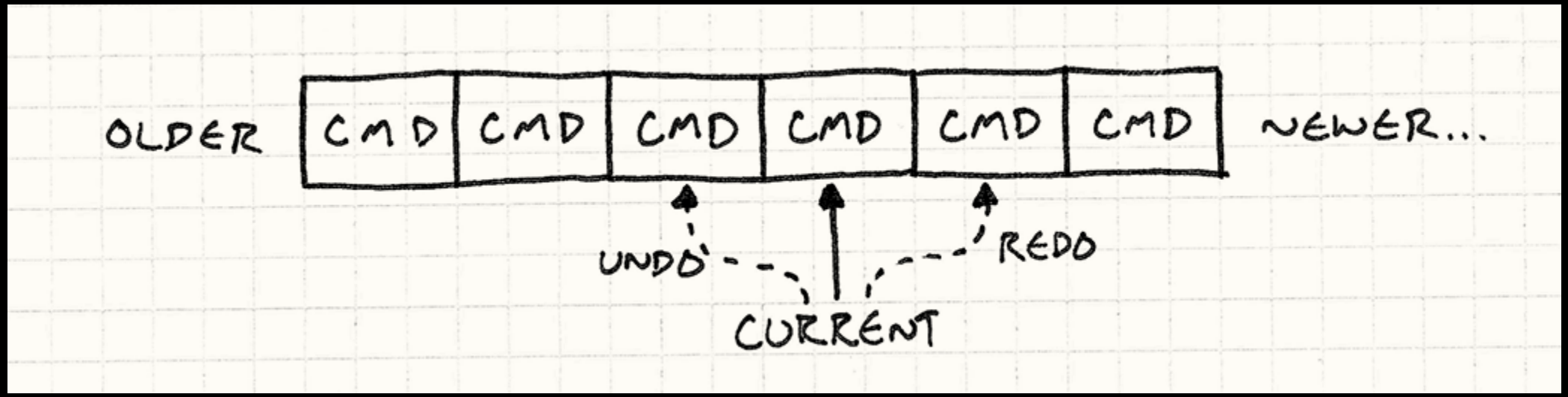
실행 취소와 재실행

```
class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
    virtual void undo() = 0;
};
```

실행 취소와 재실행

```
class MoveUnitCommand : public Command
{
public:
    MoveUnitCommand(Unit* unit, int x, int y)
    : unit_(unit), x_(x), y_(y),
      xBefore_(0), yBefore_(0)
    {
    }

    virtual void execute()
    {
        // 나중에 이동을 취소할 수 있도록 원래 유닛 위치를 저장한다
        xBefore_ = unit_->x();
        yBefore_ = unit_->y();
        unit_->moveTo(x_, y_);
    }

    virtual void undo()
    {
        unit_->moveTo(xBefore_, yBefore_);
    }

private:
    Unit* unit_;
    int x_;
    int y_;
    int xBefore_, yBefore_;
};
```

실행 취소와 재실행



관찰자
Observer

업적 달성을 구현해 보자

- ‘몬스터 100마리 죽이기’ 같은 업적 시스템을 추가한다고 해보자
  그리고 이런 업적이 수백 개가 넘는다고 하자

- 업적 종류가 광범위하고 달성할 수 있는 방법도 다양하다 보니
  깔끔하게 구현하기가 어렵다

- 충돌 검사 계산 코드 한가운데에서 ‘다리에서 떨어지기 업적 해제’ 함수를
  호출하고 싶진 않을 것이다

업적 달성을 구현해 보자

- 특정 기능을 담당하는 코드는 항상 한데 모아두는 게 좋다
- 문제는 업적을 여러 게임 플레이 요소에서 발생시킬 수 있다는 점이다
- 이런 코드 전부와 커플링되지 않고도 업적 코드가 동작하게 하려면?

업적 달성을 구현해 보자

- 물체가 표면에 놓여 있는지, 바닥으로 추락하는지를 추적하는 물리 코드가 있다고 해보자
- ‘다리에서 떨어지기’ 업적을 구현하기 위해 업적 코드를 물리 코드에 넣을 수 있지만 이러면 코드가 지저분해진다
- 다음과 같이 해보자

업적 달성을 구현해 보자

```
void Physics::updateEntity(Entity& entity)
{
    bool wasOnSurface = entity.isOnSurface();
    entity.accelerate(GRAVITY);
    entity.update();
    if (wasOnSurface && !entity.isOnSurface())
    {
        notify(entity, EVENT_START_FALL);
    }
}
```

이 코드는 '이게 방금 떨어지기 시작했으니 누군지는 몰라도 알아서 하시오'라고 알려주는 게 전부다

업적 달성을 구현해 보자

- 업적 시스템은 물리 엔진이 알림을 보낼 때마다 받을 수 있도록 스스로를 등록한다
- 업적 시스템은 떨어지는 물체가 캐릭터가 맞는지, 떨어지기 전에 다리 위에 있었는지를 확인한 뒤에 업적 잠금을 해제한다
- 이런 과정을 물리 코드는 전혀 몰라도 된다

업적 달성을 구현해 보자

- 물리 엔진 코드는 전혀 건드리지 않은 채로 업적 목록을 바꾸거나 아예 업적 시스템을 떼어낼 수도 있다

- 물리 엔진 코드는 누가 받든 말든 계속 알림을 보낸다

관찰자

```
class Observer
{
public:
    virtual ~Observer() {}
    virtual void onNotify(const Entity& entity, Event event) = 0;
};
```

관찰자

- 어떤 클래스든 Observer 인터페이스를 구현하기만 하면 관찰자가 될 수 있다

```
class Achievements : Observer
{
public:
    virtual void onNotify(const Entity& entity, Event event)
    {
        switch (event)
        {
        case EVENT_START_FALL:
            if (entity.isHero() && heroIsOnBridge_)
            {
                unlock(ACHIEVEMENT_FELL_OFF_BRIDGE);
            }
            break;
            // 그 외 다른 이벤트를 처리하고
            // heroIsOnBridge_ 값을 업데이트한다
        }
    }

private:
    void unlock(Achievement achievement)
    {
        // 아직 업적이 잠겨 있다면 잠금해제한다
    }
    bool heroIsOnBridge_;
};
```

대상

- Notify 함수는 관찰당하는 객체가 호출한다.
- 이런 객체를 대상(Subject)이라 부른다

```
class Subject
{
public:
    void addObserver(Observer* observer)
    {
        // 배열에 추가한다
    }

    void removeObserver(Observer* observer)
    {
        // 배열에서 제거한다
    }
    // 그 외...

protected:
    void notify(const Entity& entity, Event event)
    {
        for (int i = 0; i < numObservers_; ++i)
        {
            observers_[i]->onNotify(entity, event);
        }
    }
    // 그 외...

private:
    Observer* observers_[MAX_OBSERVERS];
    int numObservers_;
};
```

결과

- Subject는 Observer와 상호작용하지만 서로 커플링되어 있지 않다
- 물리 코드 어디에도 업적과 관련된 부분은 없지만 업적 시스템으로 알림을 보낼 수 있다
- Observer를 여러 개 등록할 수 있게 하면 Observer들이 각각 독립적으로 다뤄지는 걸 보장할 수 있다

남은 작업

- 물리 엔진이 알림을 보낼 수 있게 하자

```
class Physics : public Subject
{
public:
    void updateEntity(Entity& entity);
};
```

- 실제 코드였다면 상속 대신 Subject 인스턴스를 포함하게 만들었을 것이다

주의할 점 1

- 패턴이 동기적이다. 모든 Observer가 Notify 함수를 끝내기 전에는 다음 작업을 진행할 수 없다
- Observer 중 하나라도 느리면 대상이 블록될 수 있다
- 멀티스레드와 함께 사용할 때는 조심해야 한다. 어떤 Observer가 Subject의 락을 물고 있다면 게임 전체가 교착상태에 빠질 수 있다

주의할 점 2

- Observer를 부주의하게 삭제하다 보면 Dangling Pointer가 될 수 있다
- Subject를 삭제한 후 더 이상 알림을 받을 수 없는데도 Observer가 알림을 기다릴 수도 있다
- Observer를 제거하면 Subject에 있는 Observer들도 제거돼야 한다

GC가 있는데요?

- GC가 있는 언어는 명시적으로 삭제하지 않아도 될까?

- 캐릭터 체력 같은 상태를 보여주는 UI 화면을 생각해보자

1. 유저가 상태창을 열면 상태창 UI 객체를 생성한다
2. 상태창을 닫으면 UI 객체를 따로 삭제하지 않고 GC가 알아서 정리하게 한다

GC가 있는데요?

- 캐릭터가 어떤 행위로 UI 창에 알림을 보내 UI를 갱신한다

- 유저가 상태창을 닫으면?

1. UI는 보이지 않지만 캐릭터는 여전히 상태창 UI를 참조하고 있다
   따라서 GC가 수거해 가지 않고 알림이 계속 보내진다
2. 상태창을 열 때마다 상태창 인스턴스를 새로 만들어 Observer 목록이 점점 커진다

- 이를 사라진 리스너 문제(Lapsed listener Problem)이라 한다

주의할 점 3

- 버그가 여러 Observer에 퍼져 있다면 상호 작용 흐름을 추론하기가 어렵다
- 실제로 어떤 Observer가 알림을 받는지는 런타임에서 확인할 수 밖에 없다
- 즉, 프로그램에서 코드가 어떻게 상호작용하는지를 정적으로는 알 수 없고 동적으로 추론해야 한다

미래의 관찰자

- Dataflow Programming
  ex) 언리얼의 블루프린트, 유니티의 메카님
- Functional Reactive Programming
- Data Binding

객체 풀
Object Pool

의도

- 객체를 매번 할당, 해제하지 않고 고정 크기 풀에 들어 있는 객체를 재사용함으로써 메모리 사용 성능을 개선한다

동기

- 파티클 시스템을 예로 들자
- 한 번에 수백 개의 파티클이 생성될 수 있기 때문에 파티클을 굉장히 빠르게 만들 수 있어야 한다
- 더 중요한 것은 파티클을 생성, 제거하는 과정에서 메모리 단편화가 생겨서는 안 된다

패턴

- 재사용 가능한 객체들을 모아놓은 Object Pool 클래스를 정의한다
- 여기에 들어가는 객체는 현재 자신이 '사용 중'인지 여부를 알 수 있는 방법을 제공해야 한다
- 풀은 초기화될 때 사용할 객체들을 미리 생성하고 이들 객체를 '사용 안 함' 상태로 초기화한다

패턴

- 새로운 객체가 필요하면 풀에 요청한다
- 풀은 사용 가능한 객체를 찾아 '사용 중'으로 초기화한 뒤 반환한다
- 객체를 더 이상 사용하지 않는다면 '사용 안 함' 상태로 되돌린다

언제 쓸 것인가?

- 객체를 빈번하게 생성/삭제해야 한다
- 객체들의 크기가 비슷하다
- 객체를 힙에 생성하기가 느리거나 메모리 단편화가 우려된다
- 데이터베이스 연결이나 네트워크 연결같이 접근 비용이 비싸면서 재사용 가능한 자원을 객체가 캡슐화하고 있다

주의 사항

- Object Pool을 쓰겠다는 것은

  ‘메모리를 어떻게 처리할지를 내가 더 잘 안다’라고 선언하는 셈이다

- 즉, Object Pool의 한계도 직접 해결해야 한다

주의 사항

1. 사용되지 않는 객체는 메모리 낭비와 다를 바 없다

- 필요에 따라 크기를 조절해야 한다
- 크기가 너무 작으면 (크래시가 날 테니) 바로 알 수 있다
- 하지만 너무 커지지도 않도록 주의해야 한다

주의 사항

2. 한 번에 사용 가능한 객체 개수가 정해져 있다

- Object Pool의 모든 객체가 사용 중이어서 재사용할 객체를 반환받지 못할 때를 대비해야 한다
- 몇 가지 대비책이 있다

주의 사항

2. 한 번에 사용 가능한 객체 개수가 정해져 있다

1) 이런 일이 아예 생기지 않게 한다

- 가장 흔한 해결 방법이다. 사용자가 어떻게 하더라도 풀이 절대 부족하지 않도록 풀의 크기를 조절한다
- 적이나 아이템같이 중요한 Object Pool에서는 이게 답인 경우가 많다
- 몇 번 없을 최악의 상황에 맞춰서 풀의 크기를 유지해야 한다는 게 단점이다

주의 사항

2. 한 번에 사용 가능한 객체 개수가 정해져 있다

2) 그냥 객체를 생성하지 않는다

- 파티클 시스템 같은 곳에서 사용할 수 있다
- 모든 파티클이 사용 중이라면 이미 번쩍거리는 그래픽이 화면을 뒤덮고 있다는 얘기다. 이럴 때 새로운 폭발 이펙트는 이전 것보다 화려하게 터지지 않는 이상 그다지 눈에 띄지 않는다

주의 사항

2. 한 번에 사용 가능한 객체 개수가 정해져 있다

3) 기존 객체를 강제로 제거한다

- 풀이 꽉찬 상태에서 새로운 사운드를 틀어야 한다고 해보자
- 만약 새로운 사운드 생성을 무시하고 싶지 않다면 재생 중인 사운드 중에서 가장 소리가 작은 것을 새로운 사운드로 교체하는 게 낫다

주의 사항

2. 한 번에 사용 가능한 객체 개수가 정해져 있다

4) 풀의 크기를 늘린다

- 추가로 늘린 메모리를 더 이상 쓰지 않을 때는 풀의 크기를 원래대로 줄일 것인지 그대로 둘지를 정해야 한다

주의 사항

3. 객체를 위한 메모리 크기는 고정되어 있다

- 풀에 들어가는 객체가 전부 같은 자료형이라면 상관없다
- 다른 자료형인 객체나 필드가 추기된 하위 클래스의 인스턴스를 같은 풀에 넣고 싶다면 풀의 배열 한 칸 크기를 크기가 가장 큰 자료형에 맞춰야 한다
- 상황에 따라서는 객체 크기별로 풀을 나누는 게 좋다

주의 사항

4. 사용되는 객체는 저절로 초기화되지 않는다

- 새로운 객체로 할당받은 메모리에는 이전 객체의 상태가 거의 그대로 들어 있어서 초기화를 했는지 여부를 구별하기가 거의 불가능하다
- 때문에 풀에서 새로운 객체를 초기화할 때에는 주의해서 객체를 완전히 초기화해야 한다
- 객체를 회수할 때 객체가 들어 있는 배열의 메모리를 싹 초기화하는 디버깅 기능을 추가하는 것도 고려해볼만 하다

주의 사항

5. 사용 중이지 않은 객체도 메모리에 남아 있다

- GC와 Object Pool을 같이 사용한다면 충돌에 주의해야 한다
- 풀에서는 객체가 사용 중이 아니어도 메모리를 해제하지 않기 때문에 객체가 계속 메모리에 남는다
- 이 때 이들 객체가 다른 객체를 참조하고 있다면 GC에서 그 객체를 회수할 수 없다

예제 코드

```
class Particle
{
public:
    Particle() : framesLeft_(0) {}
    void init(double x, double y, double xVel, double yVel, int lifetime);
    void animate();
    bool inUse() const { return framesLeft_ > 0; }

private:
    int framesLeft_;
    double x_, y_;
    double xVel_, yVel_;
};
```

예제 코드

```
void Particle::init(double x, double y, double xVel, double yVel, int lifetime)
{
    x_ = x;
    y_ = y;
    xVel_ = xVel;
    yVel_ = yVel;
    framesLeft_ = lifetime;
}

void Particle::animate()
{
    if (!inUse()) return;

    framesLeft_--;
    x_ += xVel_;
    y_ += yVel_;
}
```

예제 코드

```
class ParticlePool
{
public:
    void create(double x, double y, double xVel, double yVel, int lifetime);
    void animate();

private:
    static const int POOL_SIZE = 100;
    Particle particles_[POOL_SIZE];
};
```

예제 코드

```
void ParticlePool::animate()
{
    for (int i = 0; i < POOL_SIZE; ++i)
    {
        particles_[i].animate();
    }
}

void ParticlePool::create(double x, double y, double xVel, double yVel, int lifetime)
{
    // 사용 가능한 파티클을 찾는다
    for (int i = 0; i < POOL_SIZE; ++i)
    {
        if (!particles_[i].inUse())
        {
            particles_[i].init(x, y, xVel, yVel, lifetime);
            return;
        }
    }
}
```

빈칸 리스트

- 사용 가능한 파티클을 찾기 위해 전체 순회를 하고 있다
- 배열이 매우 커진다면 이 작업이 느릴 수 있다
- 개선해보자

빈칸 리스트

- 파티클이 사용 중이면 live를,
  미사용 중이면 next를 사용한다

```
class Particle
{
public:
    // 원래 있던 코드들...
    Particle* getNext() const { return state_.next; }
    void setNext(Particle* next)
    {
        state_.next = next;
    }

private:
    int framesLeft_;

    union
    {
        // 사용 중일 때의 상태
        struct
        {
            double x, y;
            double xVel, yVel;
        } live;

        // 사용 중이 아닐 때의 상태
        Particle* next;
    } state_;
};
```

빈칸 리스트

```
bool Particle::animate()
{
    if (!inUse()) return;

    framesLeft_--;
    state_.live.x += state_.live.xVel;
    state_.live.y += state_.live.yVel;

    return framesLeft_ == 0;
}
```

빈칸 리스트

```
class ParticlePool
{
public:
    ParticlePool();
    void create(double x, double y, double xVel, double yVel, int lifetime);
    void animate();

private:
    static const int POOL_SIZE = 100;
    Particle particles_[POOL_SIZE];
    Particle* firstAvailabe_;
};
```

빈칸 리스트

```
ParticlePool::ParticlePool()
{
    // 처음 파티클부터 사용 가능하다
    firstAvailabe_ = &particles_[0];

    // 모든 파티클은 다음 파티클을 가리킨다
    for (int i = 0; i < POOL_SIZE - 1; ++i)
    {
        particles_[i].setNext(&particles_[i + 1]);
    }

    // 마지막 파티클에서 리스트를 종료한다
    particles_[POOL_SIZE - 1].setNext(nullptr);
}
```

빈칸 리스트

```
void ParticlePool::animate()
{
    for (int i = 0; i < POOL_SIZE; ++i)
    {
        if (particles_[i].animate())
        {
            // 방금 죽은 파티클을 빈칸 리스트 앞에 추가한다
            particles_[i].setNext(firstAvailabe_);
            firstAvailabe_ = &particles_[i];
        }
    }
}
```

빈칸 리스트

```
void ParticlePool::create(double x, double y, double xVel, double yVel, int lifetime)
{
    // 풀이 비어 있지 않은지 확인한다
    assert(firstAvailabe_ != nullptr);

    // 얻은 파티클을 빈칸 목록에서 제거한다
    Particle* newParticle = firstAvailabe_;
    firstAvailabe_ = newParticle->getNext();
    newParticle->init(x, y, xVel, yVel, lifetime);
}
```

디자인 결정

- 실제 제품 코드에서는 보통 이 정도로는 부족하다
- 몇 가지 방법으로 Object Pool을 더 일반적이면서 안전하거나 유지보수 하기 쉽게 만들 수 있다
- 게임에 Object Pool을 구현하려면 다음과 같은 사항을 결정해야 한다

주의 사항

1. 풀이 객체와 커플링되는가?

- 객체가 자신이 풀에 들어 있는지를 알게 할 것인지부터 결정해야 한다

- 아무 객체나 담을 수 있는 일반적인 풀 클래스를 구현해야 한다면
  이런 상태를 따로 구현해야 한다

주의 사항

1. 풀이 객체와 커플링되는가?

1) 객체가 풀과 커플링 된다면

- 더 간단하게 구현할 수 있다
- 풀에 들어가는 객체에 ‘사용 중’ 플래그나 이런 역할을 하는 함수를 추가하기만 하면 된다

주의 사항

1. 풀이 객체와 커플링되는가?

1) 객체가 풀과 커플링 된다면

- 객체가 풀을 통해서만 생성할 수 있도록 강제할 수 있다

```
class Particle
{
    friend class ParticlePool;

private:
    Particle() : inUse_(false) {}
    bool inUse_;
};

class ParticlePool
{
    Particle pool_[100];
};
```

주의 사항

1. 풀이 객체와 커플링되는가?

1) 객체가 풀과 커플링 된다면

- ‘사용 중’ 플래그가 꼭 필요한 건 아닐 수도 있다
- 객체에 자신이 사용 중인지를 알 수 있는 상태가 이미 있는 경우가 많다

주의 사항

1. 풀이 객체와 커플링되는가?

2) 객체가 풀과 커플링되지 않는다면

- 어떤 객체라도 풀에 넣을 수 있다
- 객체와 풀을 디커플링함으로써 일반적이면서도 재사용 가능한 풀 클래스를 구현할 수 있다

주의 사항

1. 풀이 객체와 커플링되는가?

2) 객체가 풀과 커플링되지 않는다면

- '사용 중' 상태를 객체 외부에서 관리해야 한다

- 가장 간단한 방법은 비트 필드를 따로 두는 것이다

```
template <class TObejct>
class GenericPool
{
private:
    static const int POOL_SIZE = 100;

    TObejct pool_[POOL_SIZE];
    bool inUse_[POOL_SIZE];
};
```

주의 사항

2. 재사용되는 객체를 초기화할 때

- 기존 객체를 재사용하기 위해서는 먼저 상태를 새로 초기화해야 한다
- 이 때 객체 초기화를 클래스 안에서 할지, 밖에서 할지를 정해야 한다

주의 사항

2. 재사용되는 객체를 초기화할 때

1) 객체를 풀 안에서 초기화 한다면

- 풀은 객체를 완전히 캡슐화할 수 있다
- 잘하면 객체를 풀 내부에 완전히 숨길 수 있다. 이러면 밖에서 객체를 아예 참조할 수 없기 때문에 예상치 못하게 재사용되는 걸 막을 수 있다

주의 사항

2. 재사용되는 객체를 초기화할 때

1) 객체를 풀 안에서 초기화 한다면

- 풀 클래스는 객체가 초기화되는 방법과 결합된다

- 풀에 들어가는 객체 중에는 초기화 함수를 여러 개 지원하는 게 있을 수 있다

```
class Particle
{
    // 다양한 초기화 방법
    void init(double x, double y);
    void init(double x, double y, double angle);
    void init(double x, double y, double xVel, double yVel);
};

class ParticlePool
{
public:
    void create(double x, double y)
    {
        // Particle 클래스로 포워딩한다
    }

    void create(double x, double y, double angle)
    {
        // Particle 클래스로 포워딩한다
    }

    void create(double x, double y, double xVel, double yVel)
    {
        // Particle 클래스로 포워딩한다
    }
};
```

주의 사항

2. 재사용되는 객체를 초기화할 때

2) 객체를 풀 밖에서 초기화 한다면

- 풀의 인터페이스는 단순해진다
- 풀은 객체 초기화 함수를 전부 제공할 거 없이 새로운 객체에 대한 레퍼런스만 반환하면 된다

```
class ParticlePool
{
public:
    Particle* create()
    {
        // 사용 가능한 파티클에 대한 레퍼런스를 반환한다
    }

private:
    Particle pool_[100];
};
```

```
ParticlePool pool;
pool.create()->init(1, 2);
pool.create()->init(1, 2, 0.3);
pool.create()->init(1, 2, 3.3, 4.4);
```

주의 사항

2. 재사용되는 객체를 초기화할 때

2) 객체를 풀 밖에서 초기화 한다면

- 외부 코드에서는 객체 생성이 실패할 때의 처리를 챙겨야 할 수 있다
- create()가 성공하면 객체 포인터를 반환하지만 풀이 꽉차 있다면 NULL을 반환할 수 있다