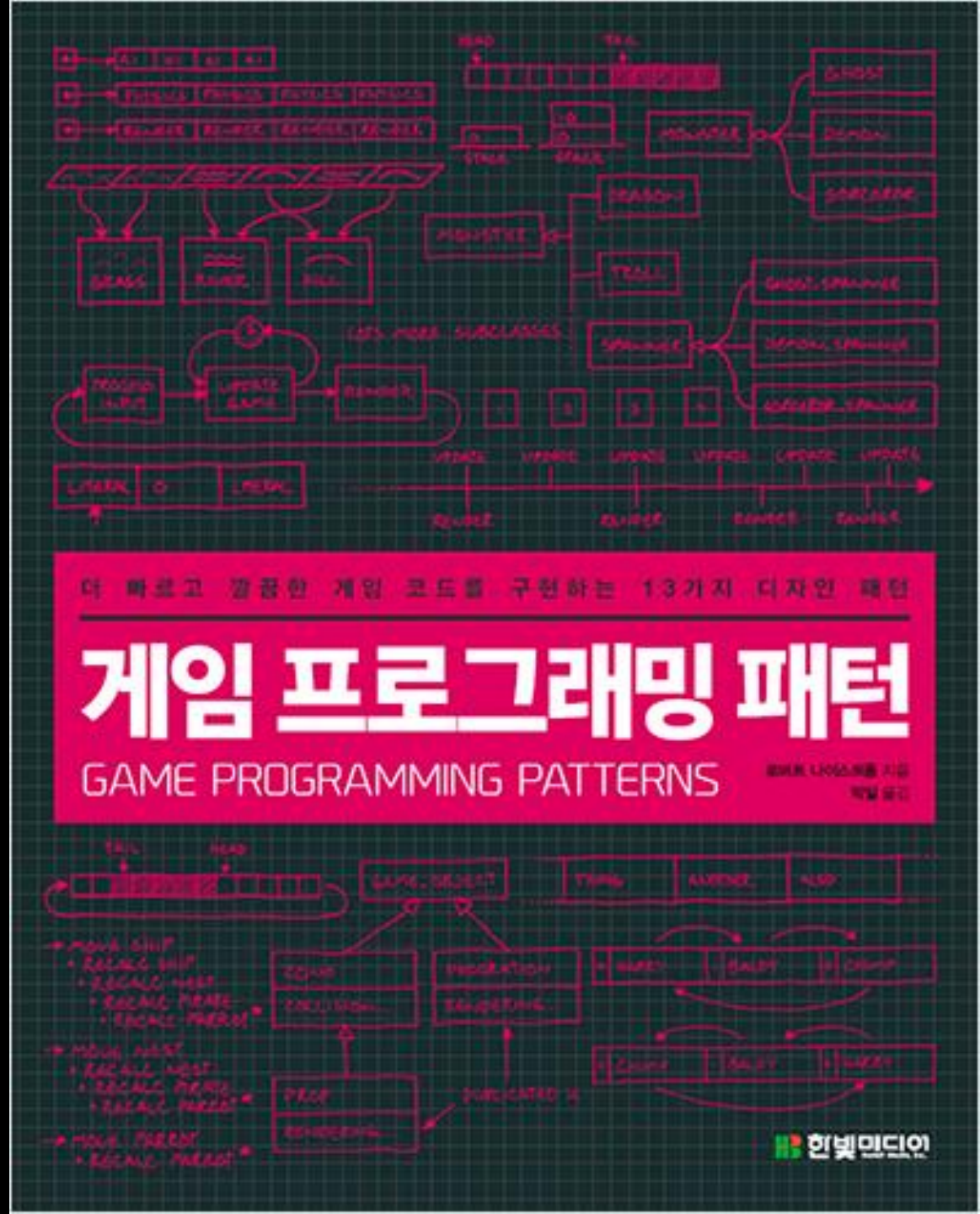
게임 프로그래밍 패턴

쿠재아이
김재경

주제 선정 이유

- 어떻게 해야 유지 보수가 좋게 할까?
- 만약 처음부터 설계를 한다면 어떻게 해야 할까?
- 코딩 할 때마다 항상 하던 생각이지만 공부한 적은 없음
- 그래서 공부했습니다

- 서점에서 우연히 발견한 책
- 인터넷에서 요약 글을 봤었는데 내용이 괜찮았던 걸로 기억
- 관심있는 주제이기도 해서 구입

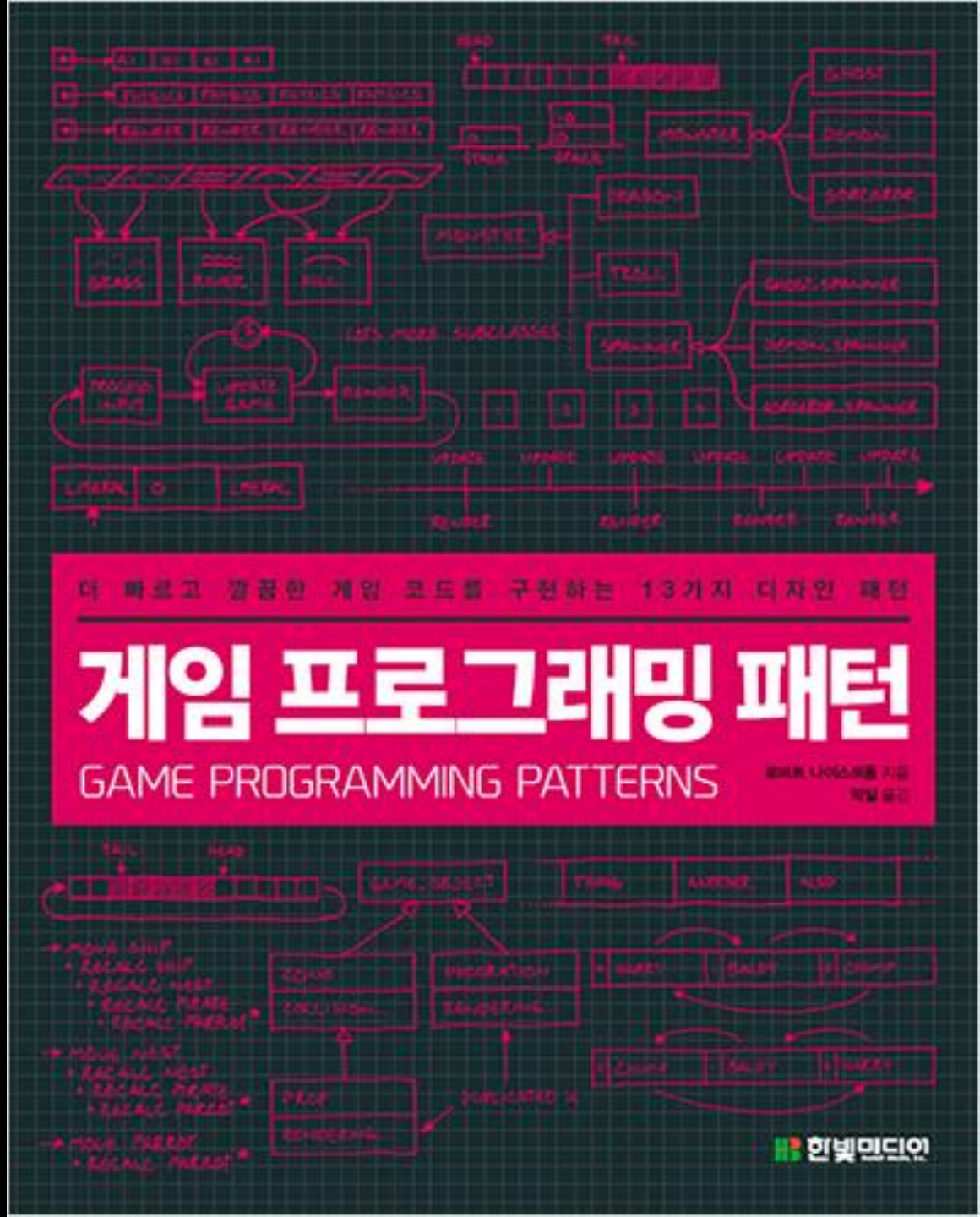


- 이런 사람에게 추천

1. 디자인 패턴 공부를 시작하고 싶은 분
2. 아직 디자인 패턴에 익숙하지 않은 분

- 책 내용이 많지 않고(400p)
  문체가 딱딱하지 않아서 읽기 편함

- 디자인 패턴 입문서로 적절할 듯
  단, 다른 디자인 패턴 책과 같이 볼 것



목차

싱글턴
Singletom

싱글턴을 사용하면서 느낀점

- 싱글턴을 너무 남발하는거 아닌가?
- 싱글턴의 문제점은 무엇인가?
- 싱글턴의 대안은 어떤게 있는가?

싱글턴이란?

1. 한 개의 인스턴스만을 갖도록 보장

- 인스턴스가 여러 개면 제대로 작동하지 않는 상황이 종종 있다
  (외부 시스템과 상호작용하면서 전역 상태를 관리하는 클래스)

- ex) 파일 시스템 API

싱글턴이란?

2. 전역적인 접근점을 제공

- 보통 생성자를 private로 하여 인스턴스를 따로 생성할 수 없으므로 인스턴스를 전역에서 접근할 수 있는 메서드를 제공

싱글턴이란?

- 생성자가 private라 밖에서 생성 X (한 개의 인스턴스를 보장)
- instance static 함수로 어디서나 접근 가능
- 필요할 때까지 초기화를 늦출 수 있음 (Lazy Initialization)

```
class FileSystem {
public:
    static FileSystem& instance()
    {
        // 게으른 초기화
        if(instance_ == NULL)
        {
            instance_ = new FileSystem();
        }

        return *instance_;
    }

private:
    FileSystem() {}
    static FileSystem* instance_;
};
```

싱글턴의 장점

1. 필요할 때 인스턴스를 생성한다

- 해당 싱글턴을 한 번도 사용하지 않는다면 메모리와 CPU 사용량을 줄일 수 있다

2. 런타임에 초기화된다

- 프로그램이 실행된 후에 알 수 있는 정보를 활용할 수 있다

싱글턴의 장점

3. 상속할 수 있다

- 만약 파일 시스템 싱글턴이 크로스 플랫폼을 지원해야 한다면 추상 인터페이스를 만든 뒤 구체 클래스를 만들면 된다

싱글턴의 장점

3-1. 상위 클래스를 만든다

```
class FileSystem
{
public:
    virtual ~FileSystem() {}
    virtual char* readFile(char* path) = 0;
    virtual void writeFile(char* path, char* contents) = 0;
};
```

싱글턴의 장점

3-2. 플랫폼별로 하위 클래스를 만든다

```
class PS3FileSystem : public FileSystem
{
public:
    virtual char* readFile(char* path)
    {
        // 소니의 파일 IO API를 사용한다...
    }
    virtual void writeFile(char* path, char* contents)
    {
        // 소니의 파일 IO API를 사용한다...
    }
};
```

```
class WIIFileSystem : public FileSystem
{
public:
    virtual char* readFile(char* path)
    {
        // 닌텐도의 파일 IO API를 사용한다...
    }
    virtual void writeFile(char* path, char* contents)
    {
        // 닌텐도의 파일 IO API를 사용한다...
    }
};
```

싱글턴의 장점

3-3. FileSystem 클래스를 싱글턴으로 만든다

```
class FileSystem
{
public:
    static FileSystem& instance();

    virtual ~FileSystem() {}
    virtual char* readFile(char* path) = 0;
    virtual void writeFile(char* path, char* contents) = 0;

protected:
    FileSystem() {}
};
```

싱글턴의 장점

3-4. 인스턴스 생성하는 부분이 핵심

```
FileSystem& FileSystem::instance()
{
#if PLATFORM == PLAYSTATION3
    static FileSystem* instance = new PS3FileSystem();
#elif PLATFROM == WII
    static FileSystem* instance = new WiiFileSystem();
#endif
    return *instance;
}
```

와! 짱 좋네! 자주 쓰면 좋을 듯 ㅎㅎ 완벽하네

싱글턴의 단점

- 전역 변수이다

```
class FileSystem {
public:
    static FileSystem& instance()
    {
        // 게으른 초기화
        if(instance_ == NULL)
        {
            instance_ = new FileSystem();
        }

        return *instance_;
    }

private:
    FileSystem() {}
    static FileSystem* instance_;
};
```

싱글턴의 단점 1

1. 전역 변수는 코드를 이해하기 어렵게 한다

- SomeClass::getSomeGlobalData() 같은 코드가 있다면 전체 코드에서 SomeGlobalData에 접근하는 곳을 다 살펴봐야 상황을 파악할 수 있다

- 만약 살펴봐야 한다면?

싱글턴의 단점 1

2. 전역 변수는 커플링을 조장한다

- 싱글턴을 사용하기 위해 #include 한 줄만 추가해도
  신중하게 만들어놓은 아키텍처를 더럽힐 수 있다

- 인스턴스에 대한 접근을 통제함으로써 커플링을 통제할 수 있다

싱글턴의 단점 1

3. 전역 변수는 멀티스레딩 같은 동시성 프로그래밍에 알맞지 않다

- 모든 스레드가 읽고 쓸 수 있는 메모리 영역이 생긴 것이다

- 교착 상태(Dead Lock), 경쟁 상태(Race Condition) 등
  스레드 동기화 버그가 생기기 쉽다

싱글턴의 단점 2

- 문제가 하나뿐일 때도 두 가지 문제를 풀려 든다
  (전역 접근 + 인스턴스 제한)

- Log 클래스가 싱글턴으로 되어있다고 하자. 전역으로 접근하여 편하게 로그를 남길 수 있지만 만약 로그를 여러 파일에 나눠쓸 상황이 온다면? 인스턴스를 한 개만 만들어야는 제약이 문제가 된다

싱글턴의 단점 3

- 게으른 초기화는 제어할 수가 없다
- 처음 초기화를 할 때 프레임이 떨어지고 렉이 걸린다면?

싱글턴의 단점 3

- 이렇게 하면 게으른 초기화 문제를 해결할 수 있다

- 단점

  1. 다형성 사용 못 함
  2. 메모리 해제 불가능

```
class FileSystem
{
public:
    static FileSystem& instance() { return instance_; }

private:
    FileSystem() {}

    static FileSystem instance_;
};
```

대안

- 클래스가 꼭 필요한가?

```
class Bullet
{
public:
    int getX() const { return x_; }
    int getY() const { return y_; }
    void setX(int x) { x_ = x; }
    void setY(int y) { y_ = y; }

private:
    int x_;
    int y_;
};
```

```
class BulletManager
{
public:
    Bullet* create(int x, int y)
    {
        Bullet* bullet = new Bullet();
        bullet->setX(x);
        bullet->setY(y);
        return bullet;
    }

    bool isOnScreen(Bullet& bullet)
    {
        return  bullet.getX() >= 0 &&
                bullet.getY() >= 0 &&
                bullet.getX() < SCREEN_WIDTH &&
                bullet.getY() < SCREEN_HEIGHT;
    }

    void move(Bullet& bullet)
    {
        bullet.setX(bullet.getX() + 5);
    }
};
```

대안

```
class Bullet
{
private:
    Bullet(int x, int y) : x_(x), y_(y) { }

bool isOnScreen()
{
    return  x_ >= 0 && x_ < SCREEN_WIDTH &&
        y_ >= 0 && y_ < SCREEN_HEIGHT;
}

void move() { x_ += 5; }

private:
    int x_;
    int y_;
};
```

대안

- 한 개의 인스턴스만 갖도록 보장하기
- 전역 접근 X

```
class FileSystem
{
public:
    FileSystem()
    {
        assert(!instantiated_);
        instantiated_ = true;
    }
    ~FileSystem()
    {
        instantiated_ = false;
    }

private:
    static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

대안

- 인스턴스에 쉽게 접근하기 (전역 접근 대체)

1. 넘겨주기

- 함수에 객체가 필요하다면 인수로 넘겨주는게 가장 쉬우면서도 최선인 경우가 많다.

- 인수로 넘겨주는 게 적절하지 않은 경우도 있다. 이러면 다른 방법을 찾아보자

대안

- 인스턴스에 쉽게 접근하기

2. 상위 클래스로부터 얻기

```
class GameObject
{
protected:
    Log& getLog() { return log_; }

private:
    static Log& log_;
};

class Enemy : public GameObject
{
    void doSomething()
    {
        getLog().write("I can log!");
    }
};
```

대안

- 인스턴스에 쉽게 접근하기

3. 이미 전역인 객체로부터 얻기

```
class Game
{
public:
    static Game& instance() { return instance_; }

    Log& getLog() { return *log_; }
    FileSystem& getFileSystem() { return *fileSystem_; }
    AudioPlayer& getAudioPlayer() { return *audioPlayer_; }

    // log_ 등을 설정하는 함수들..

private:
    static Game instance_;
    Log* log_;
    FileSystem* fileSystem_;
    AudioPlayer* audioPlayer_;
};
```

```
Game::instance().getAudioPlayer().play(VERY_LOUD_BANG);
```

대안

- 인스턴스에 쉽게 접근하기

3. 이미 전역인 객체로부터 얻기

- 나중에 Game 인스턴스를 여러 개 지원하도록 구조를 바꿔도
  Log, FileSystem, AudioPlayer는 영향을 받지 않는다

- Game 클래스에 커플링되는 단점이 있다.
  사운드를 출력하고 싶어도 Game 클래스를 알아야 한다

결론

- 싱클턴을 쓰지 말자는 게 아니다
- 싱글턴을 사용해야 하면 단점과 대안을 한 번 생각해 보자

상태
State

상태 패턴을 사용하면서 느낀점

- 객체 상태별로 클래스가 분리되기 때문에 이해하기 쉽다
- State가 매우 많아진다면?
- 여러 개의 State가 공존해야 한다면?

추억의 게임 만들기

- 간단한 횡스크롤 플랫포머를 만든다고 하자.
  먼저 B버튼을 누르면 점프해야 한다

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
    }
}
```

- 버그를 눈치 챘는가?

추억의 게임 만들기

- 공중에 있는 동안 B를 연타하면 계속 떠 있을 수 있다
- isJumping_ bool 값을 추가하면 간단히 고칠 수 있다

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
    }
}
```

추억의 게임 만들기

- 무한 점프 버그를 해결했다
- 플레이어가 땅에 있을 때 아래 버튼을 누르면 엎드리고 버튼을 떼면 일어서는 기능을 추가해보자

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        if (!isJumping_)
        {
            isJumping_ = true;
            // 점프 관련 코드...
        }
    }
}
```

추억의 게임 만들기

- 버그를 찾아 냈는가?

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        // 점프 중이 아니라면 점프한다.
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            setGraphics(IMAGE_DUCK);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        setGraphics(IMAGE_STAND);
    }
}
```

추억의 게임 만들기

1. 아래 버튼을 누른 뒤
2. B 버튼을 눌러 엎드린 상태에서 점프하고 나서
3. 공중에서 아래 버튼을 떼면
4. 점프 중인데도 땅에 서 있는 모습으로 보인다

- 플래그 변수가 더 필요하다

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        // 점프 중이 아니라면 점프한다.
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            setGraphics(IMAGE_DUCK);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        setGraphics(IMAGE_STAND);
    }
}
```

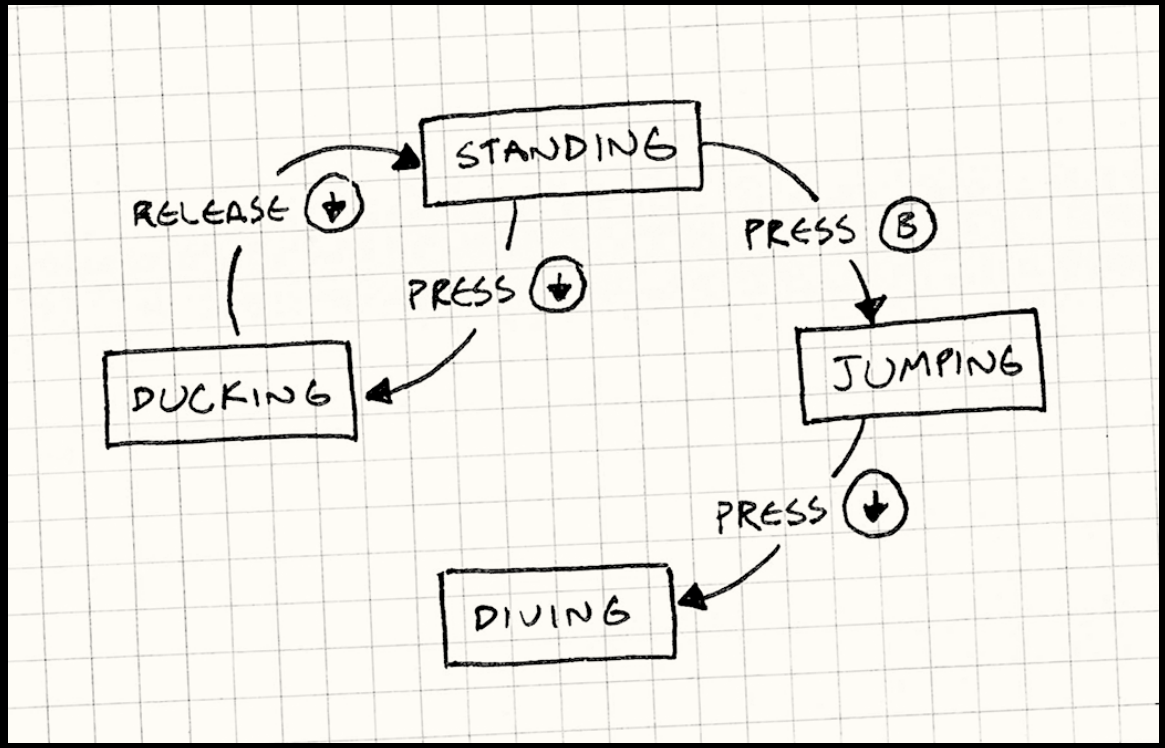
추억의 게임 만들기

- 뭔가 방향을 잘못 잡은 게 분명하다
- 코드가 얼마 없는데도 조금만 건드리면 망가진다
- 더 많은 상태가 추가된다면?

```
void Heroine::handleInput(Input input)
{
    if (input == PRESS_B)
    {
        if (!isJumping_ && !isDucking_)
        {
            // 점프 관련 코드...
        }
    }
    else if (input == PRESS_DOWN)
    {
        if (!isJumping_)
        {
            isDucking_ = true;
            setGraphics(IMAGE_DUCK);
        }
    }
    else if (input == RELEASE_DOWN)
    {
        if (isDucking_)
        {
            isDucking_ = false;
            setGraphics(IMAGE_STAND);
        }
    }
}
```

추억의 게임 만들기

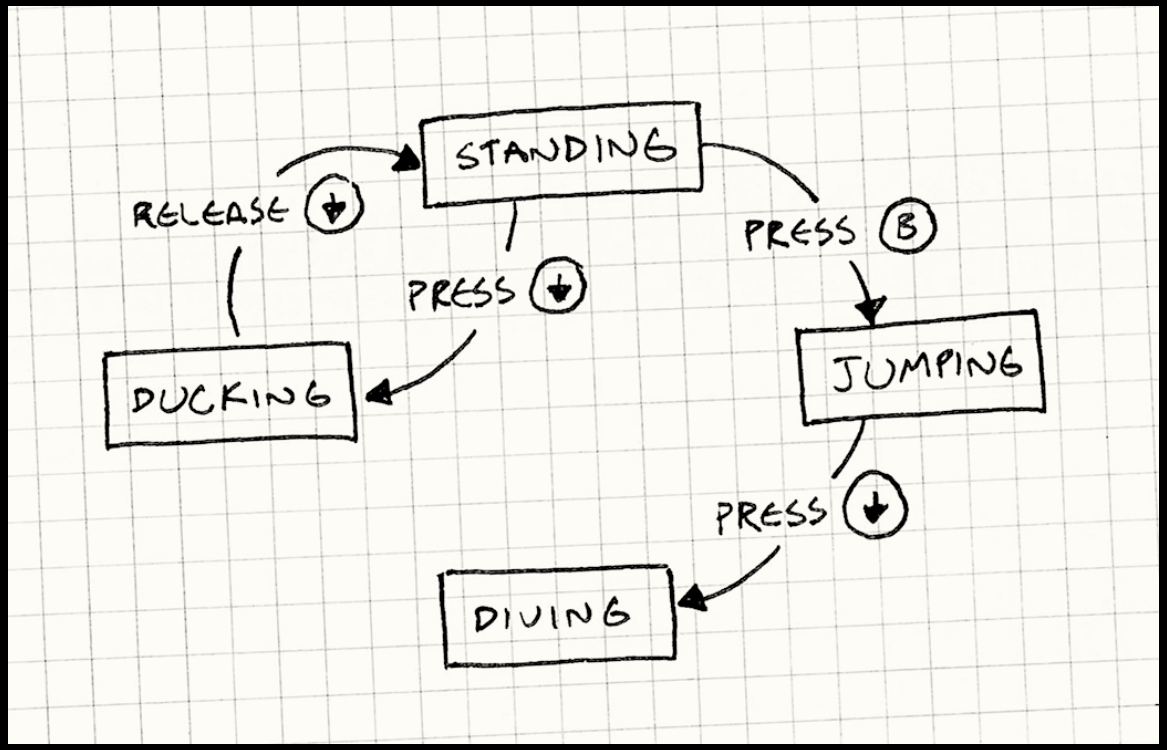
- Flowchart를 그려보자
- 유한 상태 기계(FSM)가 만들어졌다



추억의 게임 만들기

- FSM 요점

1. 가질 수 있는 '상태'가 한정된다
2. 한 번에 '한 가지' 상태만 될 수 있다
3. '입력'이나 '이벤트'가 기계에 전달된다
4. 각 상태에서는 입력에 따라 다른 상태로 바뀌는 '전이'가 있다



열거형과 다중 선택문

- bool 값 조합이 유효하지 않을 수 있다
  ex) isJumping과 isDucking은 동시에 참이 될 수 없다

- 여러 플래그 변수 중에서 하나가 참일 때가 많다면 열거형(Enum)이 필요하다는 신호다

- 예제에서는 FSM 상태를 열거형으로 정의할 수 있다

열거형과 다중 선택문

```
void Heroine::handleInput(Input input)
{
    switch (state_)
    {
    case STATE_STANDING:
        if (input == PRESS_B)
        {
            state_ = STATE_JUMPING;
            yVelocity_ = JUMP_VELOCITY;
            setGraphics(IMAGE_JUMP);
        }
        else if (input == PRESS_DOWN)
        {
            state_ = STATE_DUCKING;
            setGraphics(IMAGE_DUCK);
        }
        break;
    case STATE_JUMPING:
        if (input == PRESS_DOWN)
        {
            state_ = STATE_DIVING;
            setGraphics(IMAGE_DIVE);
        }
        break;
    case STATE_DUCKING:
        if (input == RELEASE_DOWN)
        {
            state_ = STATE_STANDING;
            setGraphics(IMAGE_STAND);
        }
        break;
    }
}
```

열거형과 다중 선택문

- 분기문을 다 없애진 못했지만 업데이트해야 할 상태 변수를 하나로 줄였다(state_ 변수)

- 열거형 만으로는 부족할 수도 있다. 이동을 구현하되, 엎드려 있으면 기가 모여서 놓는 순간 특수 공격을 쏠 수 있게 만든다고 해보자

열거형과 다중 선택문

```
void Heroine::update()
{
    if (state_ == STATE_DUCKING)
    {
        chargeTime_++;
        if (chargeTime_ > MAX_CHARGE)
        {
            superBomb();
        }
    }
}
```

```
void Heroine::handleInput(Input input)
{
    switch (state_)
    {
    case STATE_STANDING:
        if (input == PRESS_DOWN)
        {
            state_ = STATE_DUCKING;
            chargeTime_ = 0;
            setGraphics(IMAGE_DUCK);
        }

        // 다른 입력 처리
        break;

        // 다른 상태 처리...
    }
}
```

열거형과 다중 선택문

1. 기 모으기 공격을 추가하기 위해 함수 2개를 수정했다

2. 엎드리기 상태에서만 의미 있는 chargeTime 변수를 추가해야 했다

- 이것보다는 모든 코드와 데이터를 한 곳에 모아둘 수 있는 게 낫다

상태 패턴

- 상태에 의존하는 모든 코드를 인터페이스의 가상 함수로 만든다

```
class HeroineState
{
public:
    virtual ~HeroineState() {}
    virtual void handleInput(Heroine& heroine, Input input) {}
    virtual void update(Heroine& heroine) {}
};
```

상태 패턴

- 상태별로 인터페이스를 구현하는 클래스를 정의한다

```
class DuckingState : public HeroineState
{
public:
    DuckingState() : chargeTime_(0) {}

    virtual void handleInput(Heroine& heroine, Input input)
    {
        if (input == RELEASE_DOWN)
        {
            // 일어선 상태로 바꾼다...
            heroine.setGraphics(IMAGE_STAND);
        }
    }
```

```
    virtual void update(Heroine& heroine)
    {
        chargeTime_++;
        if (chargeTime_ > MAX_CHARGE)
        {
            heroine.superBomb();
        }
    }

private:
    int chargeTime_;
};
```

- chargeTIme_은 엎드리기 상태에서만 의미 있다는 점을 분명하게 보여준다

상태 패턴

- Heroine 클래스에 현재 상태 객체 포인터를 추가한다
- 상태를 바꾸려면 state_에 HeroineState를 상속받은 다른 객체를 할당하면 된다

```
class Heroine
{
public:
    virtual void handleInput(Input input)
    {
        state_->handleInput(*this, input);
    }
    virtual void update()
    {
        state_->update(*this);
    }
    // 다른 메서드들...

private:
    HeroineState* state_;
};
```

상태 객체는 어디에 둬야 할까?

- 정적 객체
- 상태 객체에 필드가 따로 없다면 인스턴스는 하나만 있으면 된다

```
class HeroineState
{
public:
    static StandingState standing;
    static DuckingState ducking;
    static JumpingState jumping;
    static DivingState diving;
    // 다른 코드들...
};
```

```
if (input == PRESS_B)
{
    heroine.state_ = &HeroineState::jumping;
    heroine.setGraphics(IMAGE_JUMP);
}
```

상태 객체는 어디에 둬야 할까?

- 전이 할 때마다 상태 객체를 만든다

```
HeroineState* StandingState::handleInput(Heroine& heroine, Input input)
{
    if (input == PRESS_DOWN)
    {
        // 다른 코드들...
        return new DuckingState();
    }
    // 지금 상태를 유지한다
    return NULL;
}
```

```
void Heroine::handleInput(Input input)
{
    HeroineState* state = state_->handleInput(*this, input);
    if (state != NULL)
    {
        delete state_;
        state_ = state;
    }
}
```

입장과 퇴장

- 상태 패턴의 목표는 같은 상태에 대한 모든 동작과 데이터를 클래스 하나에 캡슐화하는 것이다

- 지금까지는 이전 상태에서 스프라이트를 변경했다 (DuckingState에서 IMAGE_STAND로 변경, 51p 좌측 사진)

- 이보다는 같은 상태에서 그래픽까지 제어하는 게 바람직하다 이를 위해 입장 기능을 추가하자

입장과 퇴장

```
class StandingState : public HeroineState
{
public:
    virtual void enter(Heroine& heroine)
    {
        heroine.setGraphics(IMAGE_STAND);
    }
    // 다른 코드들...
};
```

```
void Heroine::handleInput(Input input)
{
    HeroineState* state = state_->handleInput(*this, input);
    if (state != NULL)
    {
        delete state_;
        state_ = state;

        // 새로운 상태의 입장 함수를 호출한다.
        state_->enter(*this);
    }
}
```

```
HeroineState* DuckingState::handleInput(Heroine& heroine, Input input)
{
    if (input == RELEASE_DOWN)
    {
        return new StandingState();
    }
}
```

입장과 퇴장

- Stand 상태로 변경하기만 하면 Stand 상태가 알아서 그래픽까지 챙긴다
- 이전 상태와는 상관없이 항상 같은 입장 코드가 실행된다는 것도 장점
- 새로운 상태로 교체되기 직전에 호출되는 퇴장도 이런 식으로 하면 된다

병행 상태 기계

- 플레이어가 총을 들 수 있게 만든다고 해보자
- 총을 장착한 후에도 달리기, 점프, 엎드리기 동작을 할 수 있어야 한다
  그러면서 총도 쏠 수 있어야 한다
- FSM 방식은 모든 상태를 서기, 무장한 채로 서기, 점프, 무장한 채로 점프
  같은 식으로 무장, 비무장에 맞춰 2개씩 만들어야 한다
- 이 문제는 상태 기계를 둘로 나누면 된다

병행 상태 기계

```
class Heroine
{
    // 다른 코드들...

private:
    HeroineState* state_;
    HeroineState* equipment_;
};
```

```
void Heroine::handleInput(Input input)
{
    state_->handleInput(*this, input);
    equipment_->handleInput(*this, input);
}
```

병행 상태 기계

- 두 상태 기계가 서로 전혀 연관이 없다면 좋은 방법이 될 수 있다
- 현실적으로는 점프 도중에는 총을 못 쏜다는 식으로 복수의 상태 기계가 상호작용해야 할 수도 있다
- 이를 위해 어떤 상태 코드(총)에서 다른 상태 기계의 상태(점프)가 무엇인지를 검사하는 지저분한 코드를 만들 일이 생길 수도 있다

계층형 상태 기계

- 플레이어가 서기, 걷기, 달리기, 미끄러지기 같은 상태가 있다고 하자
- 위 상태에서는 모두 B 버튼을 누르면 점프하고, 아래 버튼을 누르면 엎드려야 한다
- FSM 방식은 각 상태마다 코드를 중복해 넣어야 한다. 그보다는 한 번만 구현하고 다른 상태에서 재사용하는 게 낫다
- 상속을 이용하면 된다

계층형 상태 기계

```
class OnGroundState : public HeroineState
{
public:
    virtual void handleInput(Heroine& heroine, Input input)
    {
        if (input == PRESS_B)
        {
            // 점프...
        }
        else if (input == PRESS_DOWN)
        {
            // 엎드리기...
        }
    }
};
```

```
class DuckingState : public OnGroundState
{
public:
    virtual void handleInput(Heroine& heroine, Input input)
    {
        if (input == RELEASE_DOWN)
        {
            // 서기..
        }
        else
        {
            // 따로 입력을 처리하지 않고, 상위 상태로 보낸다.
            OnGroundState::handleInput(heroine, input);
        }
    }
};
```

Push-Down Automata

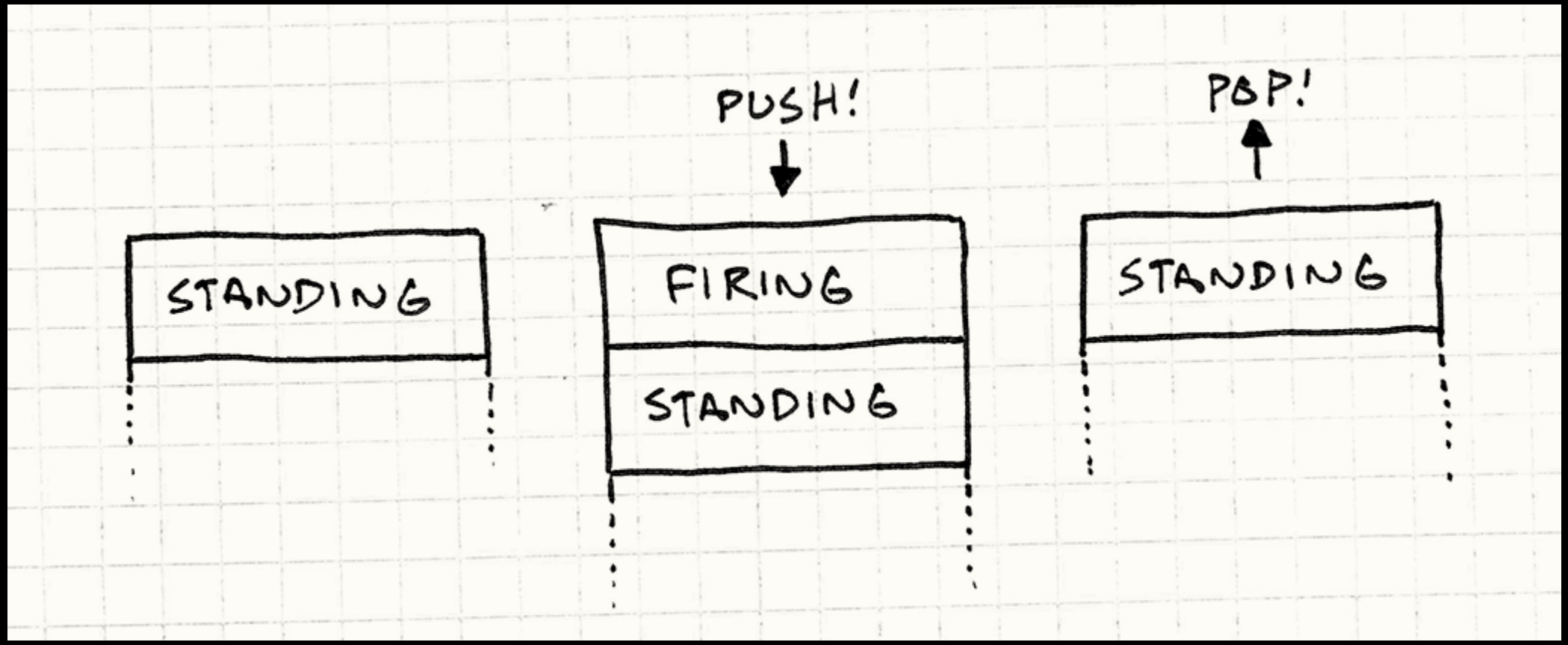
- FSM에서는 현재 상태는 알 수 있지만 이전 상태가 무엇인지 알 수 없다

- 사진의 스테이지 화면을 로비에서도 접근할 수 있고 퀘스트 화면에서도 접근할 수 있다면?

- 이전 화면으로 어떻게 돌아갈 것인가?

Push-Down Automata

- 상태를 스택으로 관리하면 된다



언제 사용해야 하는가?

- 내부 상태에 따라 객체 동작이 바뀔 때
- 이런 상태가 많지 않은 선택지로 분명하게 구분될 수 있을 때
- 객체가 입력이나 이벤트에 따라 반응할 때

게임 AI에서는?

- 사실 FSM을 처음 접한 건 게임 AI를 공부했을 때
- 근데 요즘은 안 쓴다고 하더라…
- Behavior Tree 나 Planning System을 더 많이 쓴다고 한다

게임 AI에서는?

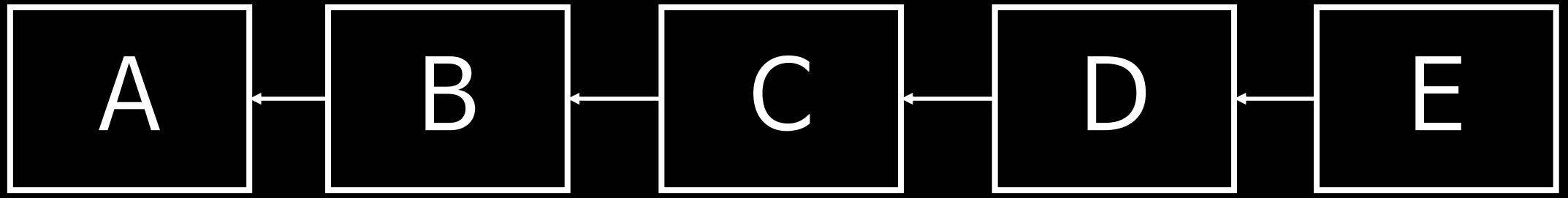
https://www.slideshare.net/yonghakim900/2009-ndc

컴포넌트
Component

컴포넌트 패턴을 선택한 이유

- 상속을 사용해보니 구조가 복잡해 질수록 유지보수 하기가 힘들었음
- 상속을 사용하지 않고 코드를 재사용하는 방법이 없을까?

컴포넌트 패턴을 선택한 이유



- 이런 상속 구조에서 코드를 보기 힘들 때가 있다

동기

- 플레이어 클래스에서는 AI, 물리, 렌더링, 사운드 등의 클래스들이 실타래처럼 얽혀 있다

```
if (collidingWithFloor() && (getRenderState() != INVISIBLE))
{
    playSound(HIT_FLOOR);
}
```

- 이 코드를 고치려면 물리(collidingWithFloor), 그래픽(getRenderState), 사운드(playSound)를 전부 알아야 한다 (커플링)

패턴

- 여러 분야를 다루는 하나의 개체가 있다
- 분야별로 격리하기 위해, 각각의 코드를 별도의 컴포넌트 클래스에 둔다
- 이제 개체 클래스는 단순히 컴포넌트들의 컨테이너 역할만 한다

언제 쓸 것인가?

- 다음 조건 중 하나라도 만족한다면 유용하게 쓸 수 있다

1. 한 클래스에서 여러 분야를 건드리고 있어서 이들을 서로 디커플링 하고 싶다

2. 클래스가 거대해져서 작업하기가 어렵다

3. 여러 다른 기능을 공유하는 다양한 객체를 정의하고 싶다
   단, 상속으로는 딱 원하는 부분만 골라서 재사용할 수가 없다

주의 사항

- 클래스 하나에 코드를 모아놨을 때보다 더 복잡해질 가능성이 높다
- 한 무리의 객체를 생성하고 초기화하고 알맞게 묶어줘야 하나의 개념적인 '객체'를 만들 수 있기 때문이다.
- 컴포넌트끼리 통신하기도 더 어렵고 컴포넌트들을 메모리 어디에 둘지 제어하는 것도 더 복잡하다

주의 사항

- 코드베이스 규모가 크면 이런 복잡성에서 오는 손해보다 디커플링과 컴포넌트를 통한 코드 재사용에서 얻는 이득이 더 클 수 있다

- 하지만 컴포넌트 패턴을 적용하기 전에 아직 있지도 않은 문제에 대한 '해결책'을 오버엔지니어링하려는 것은 아닌지 주의해야 한다

주의 사항

- 또 다른 문제는 무엇이든지 하려면 한 단계를 거쳐야 할 때가 많다는 점이다
- 무슨 일이든 컨테이너 객체에서 원하는 컴포넌트부터 얻어야 할 수 있다
- 성능이 민감한 내부 루프 코드에서 이런 식으로 포인터를 따라가다 보면 성능이 떨어질 수 있다

통짜 클래스

- 컴포넌트 패턴을 아직 적용하지 않아 모든 기능이 클래스에 다 들어있다

```
class Character
{
public:
    Character() : velocity_(0), x_(0), y_(0) {}
    void update(World& world, Graphics& graphics);

private:
    static const int WALK_ACCELERATION = 1;

    int velocity_;
    int x_, y_;

    Volume volume_;

    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};
```

```
void Character::update(World& world, Graphics& graphics)
{
    // 입력에 따라 주인공의 속도를 조절한다.
    switch (Controller::getJoystickDirection())
    {
    case DIR_LEFT:
        velocity_ -= WALK_ACCELERATION;
        break;

    case DIR_RIGHT:
        velocity_ += WALK_ACCELERATION;
        break;
    }

    // 속도에 따라 위치를 바꾼다
    x_ += velocity_;
    world.resolveCollision(volume_, x_, y_, velocity_);

    // 알맞은 스프라이트를 그린다.
    Sprite* sprite = &spriteStand_;
    if (velocity_ < 0) { sprite = &spriteWalkLeft_; }
    else if (velocity_ > 0) { sprite = &spriteWalkRight_; }
    graphics.draw(*sprite, x_, y_);
}
```

분야별로 나누기

```
class InputComponent
{
public:
    void update(Character& character)
    {
        switch (Controller::getJoystickDirection())
        {
        case DIR_LEFT:
            character.velocity -= WALK_ACCELERATION;
            break;

        case DIR_RIGHT:
            character.velocity += WALK_ACCELERATION;
            break;
        }
    }

private:
    static const int WALK_ACCELERATION = 1;
};
```

```
class Character
{
public:
    int velocity;
    int x, y;

    void update(World& world, Graphics& graphics)
    {
        input_.update(*this);

        // 속도에 따라 위치를 바꾼다.
        x += velocity;
        world.resolveCollision(volume_, x, y, velocity);

        // 알맞은 스프라이트를 그린다.
        Sprite* sprite = &spriteStand_;
        if(velocity < 0) { sprite = &spriteWalkLeft_; }
        else if(velocity > 0) { sprite = &spriteWalkRight_; }
        graphics.draw(*sprite, x, y);
    }

private:
    InputComponent input_;

    Volume volume_;

    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};
```

분야별로 나누기

```
class PhysicsComponent
{
public:
    void update(Character& character, World& world)
    {
        character.x += character.velocity;
        world.resolveCollision(volume_, character.x, character.y, character.velocity);
    }

private:
    Volume volume_;
};
```

분야별로 나누기

```
class GraphicsComponent
{
public:
    void update(Character& character, Graphics& graphics)
    {
        Sprite* sprite = &spriteStand_;
        if (character.velocity < 0)
        {
            sprite = &spriteWalkLeft_;
        }
        else if (character.velocity > 0)
        {
            sprite = &spriteWalkRight_;
        }
        graphics.draw(*sprite, character.x, character.y);
    }

private:
    Sprite spriteStand_;
    Sprite spriteWalkLeft_;
    Sprite spriteWalkRight_;
};
```

분야별로 나누기

```
class Character
{
public:
    int velocity;
    int x, y;

    void update(World& world, Graphics& graphics)
    {
        input_.update(*this);
        physics_.update(*this, world);
        graphics_.update(*this, graphics);
    }

private:
    InputComponent input_;
    PhysicsComponent physics_;
    GraphicsComponent graphics_;
};
```

오토- 캐릭터

- 컴포넌트 클래스를 추상화한다

```
class InputComponent
{
public:
    virtual ~InputComponent() {}
    virtual void udpate(Character& character) = 0;
};

class PlayerInputComponent : public InputComponent
{
public:
    void update(Character& character)
    {
        switch (Controller::getJoystickDirection())
        {
        case DIR_LEFT:
            character.velocity -= WALK_ACCELERATION;
            break;

        case DIR_RIGHT:
            character.velocity += WALK_ACCELERATION;
            break;
        }
    }

private:
    static const int WALK_ACCELERATION = 1;
};
```

오토- 캐릭터

```
class Character
{
public:
    int velocity;
    int x, y;

    Character(InputComponent* input) : input_(input) {}

    void update(World& world, Graphics& graphics)
    {
        input_->update(*this);
        physics_.update(*this, world);
        graphics_.update(*this, graphics);
    }

private:
    InputComponent* input_;
    PhysicsComponent physics_;
    GraphicsComponent graphics_;
};
```

```
Character* character = new Character(new PlayerInputComponent());
```

오토- 캐릭터

```
class DemoInputComponent : public InputComponent
{
public:
    virtual void update(Character& character)
    {
        // AI가 알아서 캐릭터를 조정한다...
    }
};
```

```
Character* character = new Character(new DemoInputComponent());
```

캐릭터일 필요는 없다

- 이제 캐릭터 클래스는 컴포넌트 묶음일 뿐 캐릭터와 관련된 코드가 없다
- 게임에서 모든 객체가 기본으로 사용하는 GameObject 클래스로 바꾸는 게 더 좋을 것 같다

캐릭터일 필요는 없다

```
class PhysicsComponent
{
public:
    virtual ~PhysicsComponent() {}
    virtual void update(GameObject& obj, World& world) = 0;
};

class CharacterPhysicsComponent : public PhysicsComponent
{
public:
    virtual void update(GameObject& obj, World& world)
    {
        // 물리 코드...
    }
};
```

캐릭터일 필요는 없다

```
class GraphicsComponent
{
public:
    virtual ~GraphicsComponent() {}
    virtual void update(GameObject& obj, Graphics& graphics) = 0;
};

class CharacterGraphicsComponent : public GraphicsComponent
{
public:
    virtual void update(GameObject& obj, Graphics& graphics)
    {
        // 그래픽스 코드...
    }
};
```

캐릭터일 필요는 없다

```
class GameObject
{
public:
    int velocity;
    int x, y;

    GameObject(InputComponent* input,
               PhysicsComponent* physics,
               GraphicsComponent* graphics)
    : input_(input),
      physics_(physics),
      graphics_(graphics)
    { }

    void update(World& world, Graphics& graphics)
    {
        input_->update(*this);
        physics_->update(*this, world);
        graphics_->update(*this, graphics);
    }

private:
    InputComponent* input_;
    PhysicsComponent* physics_;
    GraphicsComponent* graphics_;
};
```

```
GameObject* createCharacter()
{
    return new GameObject(
        new PlayerInputComponent(),
        new CharacterPhysicsComponent(),
        new CharacterGraphicsComponent());
}
```