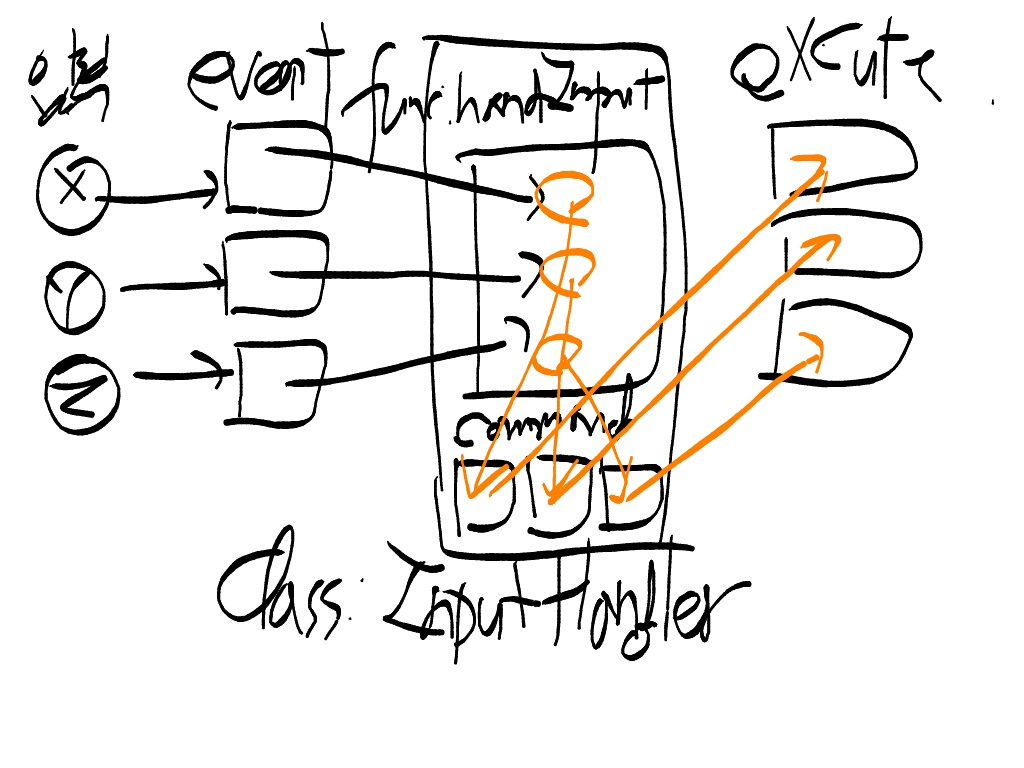
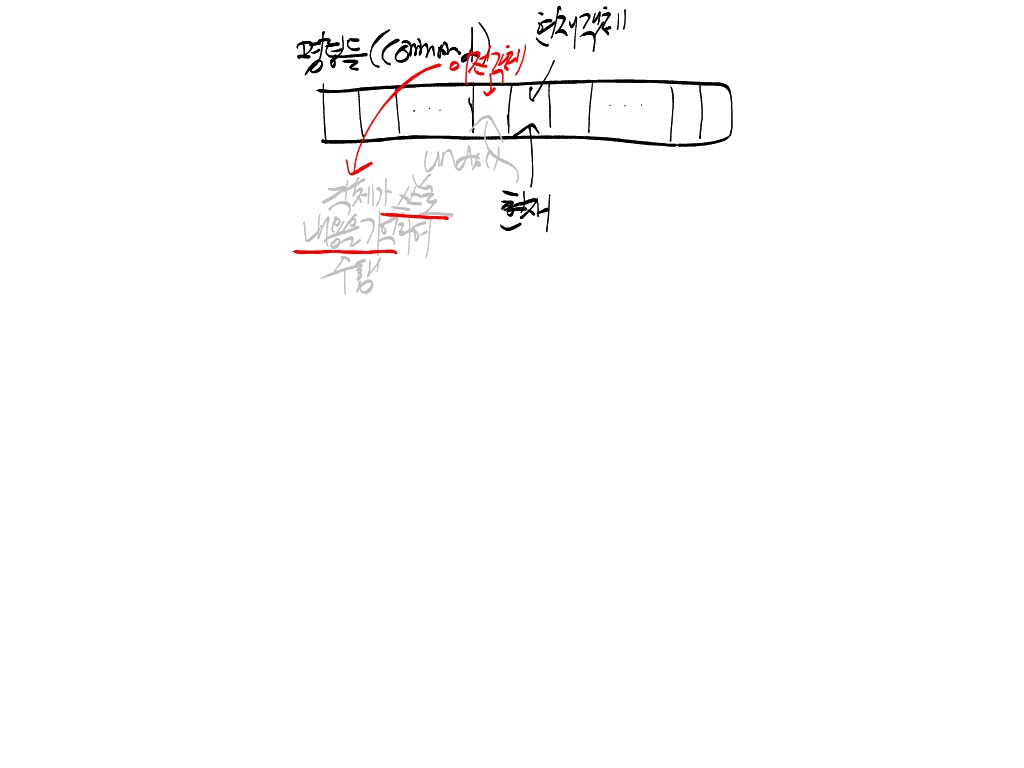
2. 명령 패턴

## 동작과정



명령을 command클래스로 만듭니다. execute()를 재정의하여 모든 입력에 대해 일관성 있게 작동합니다.



명령은 객체에서 실행합니다. 정보를 담을 수 있으므로 이전 상태로 되돌리거나 재실행할 수 있습니다.

## 세부사항

### 디커플링: 어느 한 코드를 변경했을 때, 다른 코드를 변경하지 않아도 된다.

액터와 입력은 디커플링되어야 한다. 이로써 PC와 NPC를 일반화할 수 있다. 입출력장치뿐만 아니라 다양한 AI를 입력으로써 액터에게 명령할 수 있다.

### 직렬화

명령들을 직렬화할 수 있게 만들면, 네트워크로도 전달할 수 있다.

참고: <http://www.devtimes.com/53>

**Serialization(직렬화)**  
직렬화는 객체를 저장매체(파일이나 메모리와 같은)에 저장하거나 바이너리(이진) 형태로 네트워크를 통해 전달하기 위해 저장하는 과정이다. 이(객체를 직렬화한) 연속된 바이트나 포맷은 다시 원래 객체가 가지고 있던 상태로 객체를 재생성하는데 사용할 수 있다. 이렇게 객체를 직렬화하는 과정은 디플레이팅(deflating) 또는 마샬링(marshalling)이라고도 부른다. 직렬화와는 반대로 연속된 바이트로부터 어떤 데이터 구조를 추출해내는 과정을 역직렬화(deserialization)라고 한다. 역직렬화는 인플레이팅(inflating) 또는 언마샬링(unmarshalling)이라 불리기도 한다.

**직렬화의 장단점**

**직렬화의 주요한 장점은**  
DOM을 이용하지 않고 XML 문서를 수정할 수 있다.   
어떤 어플리케이션으로부터 다른 어플리케이션으로 객체를 전달할 수 있다.   
어떤 도메인으로부터 다른 도메인으로 객체를 전달할 수 있다.   
객체를 XML 문자열의 형태로 방화벽을 거쳐(통과해) 전달할 수 있다(방화벽에 제약을 받지 않고 객체를 전달할 수 있다).

**반면 직렬화의 주요한 단점은**  
직렬화와 역직렬화 과정에서 자원(CPU와 IO 장치)의 소모가 많다.   
네트워크를 통해 객체를 전달할 경우 지연 문제가 발생할 수 있다(직렬화 과정은 상당히 느리다).   
XML로 직렬화하는 것은 안전하지 않으며(XML 문서의 형태로 저장할 경우 다른 프로그램이나 사용자가 실수나 고의로 문서의 내용을 변경할 수도 있다.), 많은 저장 공간을 차지하고, public 클래스에 대해서만 직렬화가 이루어질 수 있으므로 private 클래스 또는 internal 클래스는 직렬화할 수 없다(프로그래머로 하여금 직렬화를 지원하는 클래스를 만들기 위해 불필요한 public 클래스를 양산할 수도 있다).

## 구현

#include *<iostream>*

#include *<memory>*

#define BUTTON\_TYPE unsigned int

#define BUTTON\_X 0

#define BUTTON\_Y 1

**class** **Unit**{

**public**:

**inline** void moveTo(int \_x, int \_y)

{ x = \_x, y = \_y; }

**inline** std::pair<int, int> getLocation() **const**

{ **return** std::make\_pair(x, y); }

**private**:

int x;

int y;

};

**class** **Command**;

**class** **InputHandler**;

**class** **GameMode**{

**public**:

GameMode& getInstance();

**static** bool isPressed(BUTTON\_TYPE bt);

**static** std::shared\_ptr<Unit> getSelectedUnit();

**private**:

**struct** Keyboard{

**static** **const** int MAX\_KEY = 2;

**static** bool onButton[MAX\_KEY];

};

**struct** PlayerController{

**static** std::shared\_ptr<Unit> selected;

};

};

GameMode& GameMode::getInstance(){

**static** GameMode gm;

**return** gm;

}

bool GameMode::isPressed(BUTTON\_TYPE bt){

**if** (Keyboard::onButton[bt]) **return** true;

**return** false;

}

std::shared\_ptr<Unit> GameMode::getSelectedUnit(){

**return** PlayerController::selected;

}

bool GameMode::Keyboard::onButton[] = { false };

std::shared\_ptr<Unit> GameMode::PlayerController::selected = **nullptr**;

*//user interface*

bool isPressed(BUTTON\_TYPE bt){

**return** GameMode::isPressed(bt);

}

**const** Unit\* getSelectedUnit(){

**const** Unit\* **const** unit = GameMode::getSelectedUnit().get();

**return** unit;

}

**class** **Command**{

**public**:

**virtual** ~Command(){}

**virtual** void execute(**const** Unit&) = 0;

**virtual** void undo() = 0;

};

**class** **JumpCommand** : **public** Command{

**public**:

**virtual** void execute(**const** Unit&);

**virtual** void undo() {}

};

void JumpCommand::execute(**const** Unit& actor){

std::cout << "jump**\n**";

}

**class** **MoveUnitCommand** : **public** Command{

**public**:

MoveUnitCommand(Unit\* unit, int x, int y)

: unit\_(unit), x\_(x), y\_(y),

beforeX\_(0), beforeY\_(0) {}

**virtual** void execute(**const** Unit&);

**virtual** void undo();

**private**:

std::shared\_ptr<Unit> unit\_;

*//Unit\* unit\_;*

int x\_;

int y\_;

int beforeX\_;

int beforeY\_;

};

void MoveUnitCommand::execute(**const** Unit& unit){

unit\_ = std::shared\_ptr<Unit>(**const\_cast**<Unit\*>(&unit));

beforeX\_ = unit.getLocation().first;

beforeY\_ = unit.getLocation().second;

unit\_->moveTo(x\_, y\_);

}

void MoveUnitCommand::undo(){

unit\_->moveTo(x\_, y\_);

}

**class** **InputHandler**{

**public**:

Command\* handleInput();

**private**:

Command\* buttonX;

Command\* buttonY;

};

*//입력을 받아서 그에 맞는 Command 객체 반환*

Command\* InputHandler::handleInput(){

**if** (isPressed(BUTTON\_X)) **return** buttonX;

**if** (isPressed(BUTTON\_Y)) **return** buttonY;

**return** **nullptr**;

}

int main(){

InputHandler inputHandler;

**auto** actor = getSelectedUnit();

**auto** command = inputHandler.handleInput();

**if** (command && actor){

command->execute(\*actor);

}

**return** 0;

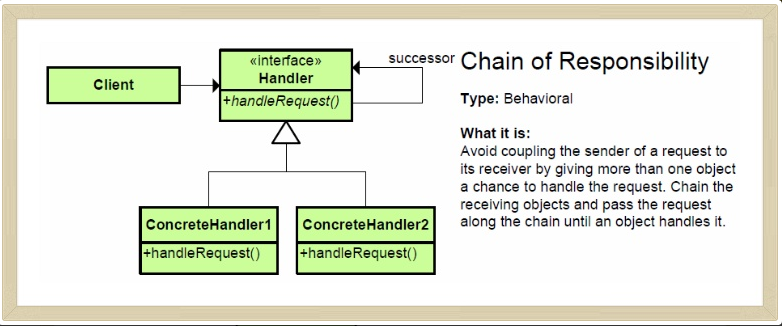
}

## 관련자료

### 수많은 Command

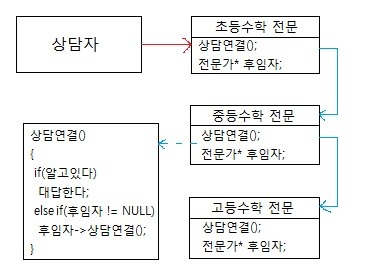
구체 상위 클래스를 여러 개 만들어놓고, 하위 클래스에서 필요한 것만 재정의하며 사용한다. – [12. 샌드박스 패턴]

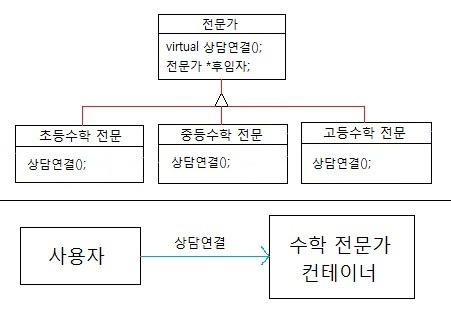
### 명령 받는 액터가 명시적이지 않은 경우

*GoF의 책임 연쇄 패턴*

명령을 처리하는 객체들이 체인으로 연결되어 있다. 현재 객체가 처리하지 못하면, 다음 객체가 처리하도록 명령을 떠넘긴다.

예제를 하나 봅시다. <http://newsdu.tistory.com/entry/>





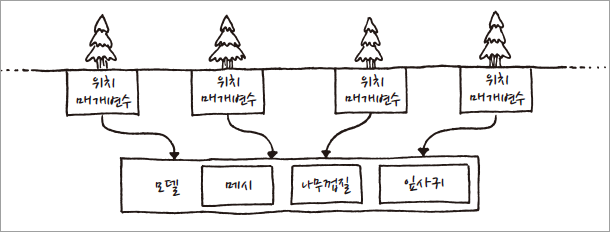
책임 연쇄 패턴은 정적으로 어떤 기능에 대한 처리의 연결이 하드코딩 되어 있을 시 기능 처리의 연결 변경이 불가능한데 이것을 동적으로 연결되어있는 경우에 따라서 다르게 처리될 수 있도록 연결한 패턴입니다. 이는 기능과 처리에 대한 결합도를 낮추게 되고 서로가 서로를 몰라도 전혀 상관없게 됩니다. 그리고 후에 필요한 추가적인 처리도 완전히 바꿀 필요 없이 책임 연쇄의 바톤을 받게끔 링크만 연결하게 하면 깔끔하게 추가가 완료됩니다.

### 데이터가 없는 Command

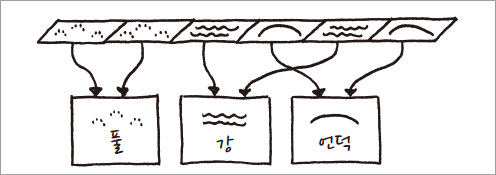
여러 명령이 들어오다 보면 command객체의 인스턴스는 많아질 것이다. 그런데 데이터가 없는 객체는 여러 인스턴스를 만들 필요가 없다. 모두 같기 때문이다. 이런 경우, 경량 패턴으로 해결할 수 있다. 싱글턴은 권장되지 않는다. – [3. 경량 패턴]

3. 경량 패턴

## 동작과정



***공유하는 data***(=*고유상태, 자유문맥*)는 위 그림과 같이 하나의 인스턴스를 참조합시다.



위의 과정으로 만들어진 terrain

## 세부사항

### 무엇이 이를 가능하게 하는가?

하나의 객체를 참조 -> 공통되는 모든 data는 공유하는 하나의 객체에 캡슐화되어 있으므로 모든 객체를 검사할 필요는 없다.

### 성능 하락

포인터로 따라가면 캐시 미스로 인한 성능 하락이 있을 수 있다. 하지만 객체 지향 방식의 장점을 가지며 상황에 따라서 성능 하락이 없을 수도 있다.

## 구현

#include *<iostream>*

#include *<vector>*

#include *<map>*

**class** **Terrain**{

**public**:

Terrain(int \_type = -1, bool \_water = false)

: type(\_type), isWater(\_water) {}

**private**:

int type;

bool isWater;

};

**const** int MAX\_TEXTURE = 5;

**class** **Factory**{

**public**:

Factory()

: terrains() {}

Terrain& createTerrain(int \_type);

**private**:

std::map<int, Terrain> terrains;

};

Terrain& Factory::createTerrain(int \_type){

**if** (terrains.find(\_type) != terrains.end())

**return** terrains[\_type];

**auto**& terrain = Terrain(\_type, false);

terrains.insert(std::pair<int, Terrain>(\_type, terrain));

**return** terrain;

}

**const** int WIDTH = 10;

**const** int HEIGHT = 10;

**class** **World**{

**public**:

World() :

grass(Terrain(1, false)),

water(Terrain(2, true)),

hill(Terrain(3, false)),

tiles(std::vector<std::vector<Terrain\*> >

(HEIGHT, std::vector<Terrain\*>(WIDTH))) {}

void generateTerrain();

**const** Terrain& getTile(int y, int x) **const**;

**private**:

std::vector<std::vector<Terrain\*> > tiles;

Terrain grass;

Terrain water;

Terrain hill;

};

void World::generateTerrain(){

**for** (int y = 0; y < HEIGHT; y++)

**for** (int x = 0; x < WIDTH; x++){

**if** (rand() % 10 == 0) tiles[y][x] = &hill;

**else** tiles[y][x] = &grass;

}

}

**const** Terrain& World::getTile(int y, int x) **const**{

**return** \*tiles[y][x];

}

int main(){

World world;

world.generateTerrain();

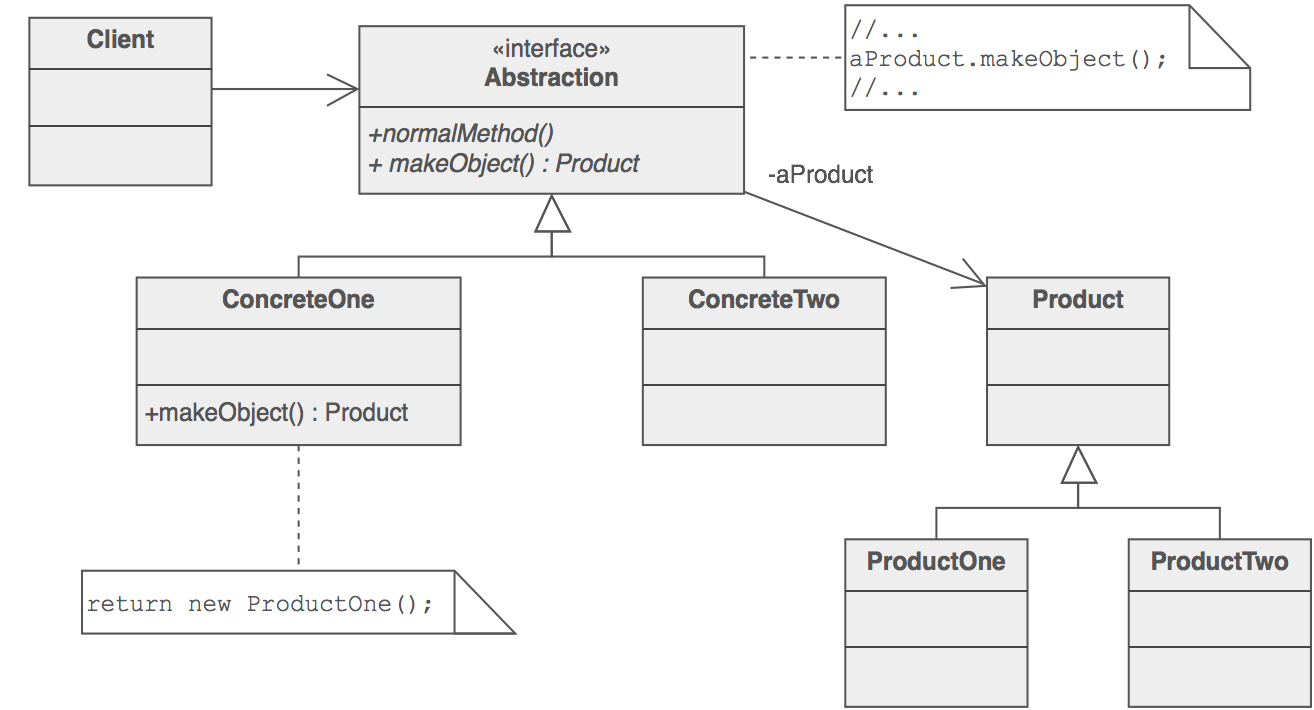
**auto** myTile = world.getTile(3, 3);

**return** 0;

}

## 관련자료

### 팩토리 메서드 패턴



생성 함수를 갖는 생성 객체를 만들어 인터페이스를 분리한다.

***이걸 왜 쓰지?***

하나의 구체에서 객체를 동적으로 결정하여 생성하는 방법이다.   
생성이 유연하다는 장점이 있다. 무슨 말이냐 하면, 어떤 자료를 가지고 동적으로 생성될 객체를 결정하고 싶을 때 사용할 수 있다. 하나의 함수에서 여러 객체를 생성할 수 있기 때문에 가능하다.  
생성의 사후작업이 용이하다. 생성자가 아닌 함수에서 생성하기 때문에 가능하다. 혹은 생성을 제어할 수 있다.

정리하면, 객체 생성 시에 생성하는 일 외에 다른 일이 동시에 필요하다면 이를 관리하는 또 다른 객체를 만들어 생성하는 방법인 셈이다.

### 동적으로 객체를 생성하고 싶다면

위의 펙토리 함수를 다른 생성 객체에 넣고 그 함수 안에서 유무확인을 하고 객체 값을 반환한다.

유무 확인 시, 이 객체들을 관리하려면 – [19. 객체 풀 패턴]

### 상태 객체 재활용

[7. 상태 패턴] – 상태 기계에서 사용되는 상태 객체에 멤버변수가 하나도 없는 경우, 경량 패턴을 적용하면 객체 하나를 여러 상태 기계에서 재사용할 수 있다.

4. 관찰자

흔히들 말하는 event개념과 비슷하다. 하지만 관점이 다르다.

## 사전지식

### 이벤트(event) 개념

**상황가정**

어떤 객체 Archer가 있다. 이 객체가 다리에서 떨어져 죽으면 어떠한 업적이 갱신되어야 한다. 이는 어떻게 구현될 것인가?

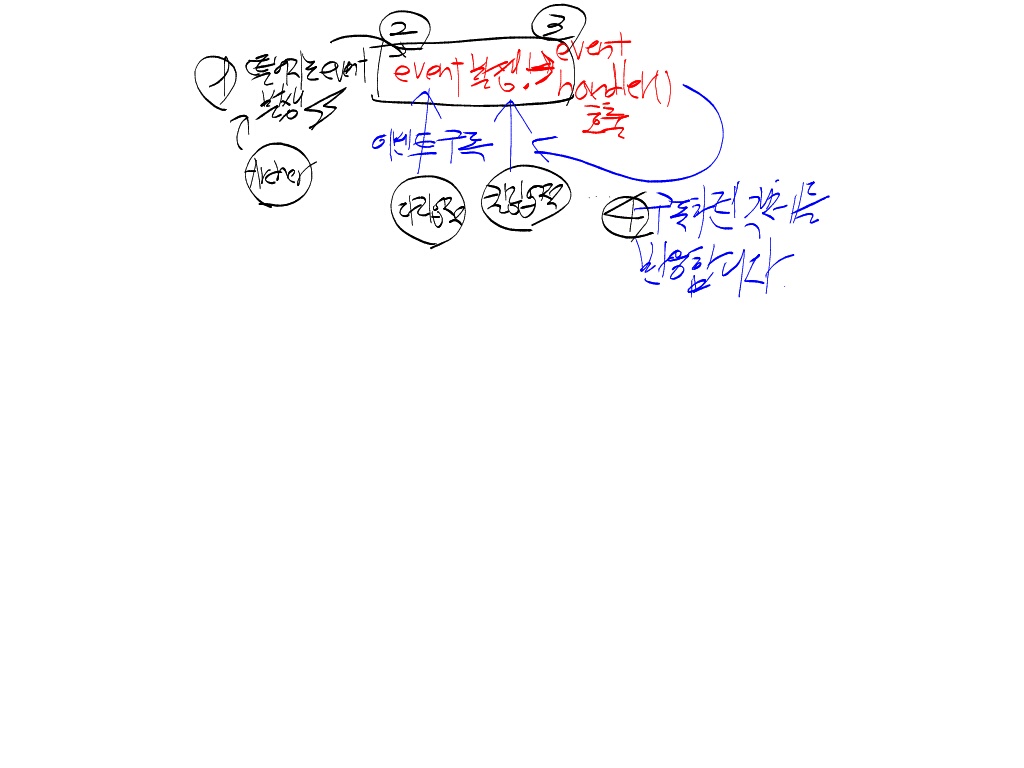
가장 기본적인 방법은

Archer의 이동 -> 지형에서 떨어짐 -> 물리엔진 작동 -> 죽음 -> 업적 관련 함수 호출

이 방법은 물리엔진 내부에서 함수를 호출하므로 Archer객체 자체의 수정을 요구하며 각 객체에 대하여 어떻게 반응할지 고려해야만 한다. 하지만 이는 그리 바람직하지 않다.

**객체와 객체간의 관심사를 분리한다.**

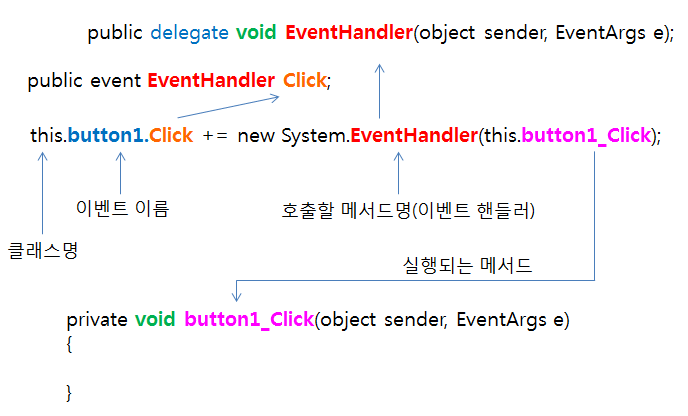
Archer는 움직이는 객체일 뿐이다. 업적을 관리하는 객체가 아니다. 이들을 분리하여 죽었을 시, 함수를 호출하게 할 수 있을까?



Archer가 event를 전달하며, 업적 관련 객체들은 그 event를 구독할 뿐이다. **누가 구독하든 Archer은 아무런 관련이 없다.** 100만개의 객체가 구독해도 Archer가 하는 행동은 딱 하나, “이벤트가 발생했다!”이다. 구독자들이 이에 반응해야 한다. 이는 분명하게 관심사가 분리됨을 보인다.

### C#에서의 event

C#언어에서는 event가 어떻게 쓰이는지 살펴보자.



EventHandler Click는 함수 객체이다.

button1.Click = event(event발생)

button1\_Click() = event구독자

EventHandler += button1\_Click() -> 이는 역할이 핸들러에 추가된다는 의미다. button1.Click이라는 이벤트가 발생하면 이를 구독하는 button1\_Click()이 실행될 것이다.

### C++에서의 event

원리는 위와 같다. 구현은 C++14에 근거한다.

출처: <http://codereview.stackexchange.com/questions/118628/c14-event-system>

#ifndef \_\_PD\_EVENT\_\_

#define \_\_PD\_EVENT\_\_

#define PD\_EVENT\_VER 8

#include *<functional>*

#include *<iostream>*

#include *<stdint.h>*

#include *<unordered\_map>*

**using** **namespace** std::placeholders;

*//*

*// @brief Describes additional parameters for handler binding.*

*//*

**enum** **class** **EventFlag** {

DEFAULT = 0,

ONLY\_UNIQUE = 1

};

*// Maximum allowed number of event arguments is 4.*

**template**<class... Args>

**class** **Event**{

**using** Address = uint64\_t;

**using** Handler = std::function<void(Args...)>;

**using** Identifier = uint64\_t;

*// @brief Maximum supported event handler arguments.*

**static** **constexpr** **auto** \_MAX\_EVENT\_ARGS = 4;

**static\_assert**(**sizeof**...(Args) <= \_MAX\_EVENT\_ARGS, "Too many arguments");

**public**:

*// @param lambda - the lambda expression to bind.*

void **operator**+=(Handler lambda) {

*// TODO find a way to unbind*

\_addToList(\_identify(&lambda, 0), lambda, EventFlag::DEFAULT);

}

void **operator**()(Args... e) {

fire(e...);

}

*// @brief Calls every subscriber in container with given arguments.*

void fire(Args... e) {

**for** (**auto** **subscriber** : subscribers)

subscriber.second(e...);

}

bool hasSubscriber(Address a) **const** {

**return** subscribers.find(a) != subscribers.end();

}

#pragma mark - bind(...) overloads

*//*

*// @brief Subscribes a member function with no arguments.*

*// @param member - member function pointer &C::M.*

*// @param instance - instance of structure housing the member function.*

*// @usage <i>event.bind(&MyClass::member, myClassInstance)</i>.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**>

void bind(C (M::\*member)(), T \*instance, EventFlag flag = EventFlag::DEFAULT) {

\_addToList(\_identify(instance, member), std::bind(member, instance), flag);

}

*//*

*// @brief Subscribes a member function with 1 argument.*

*// Event handler argument A1.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1>

void bind(C (M::\*member)(A1), T \*instance, EventFlag flag = EventFlag::DEFAULT) {

\_addToList(\_identify(instance, member), std::bind(member, instance, \_1), flag);

}

*//*

*// @brief Subscribes a member function with 2 arguments.*

*// Event handler arguments: A1, A2.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1, **typename** A2>

void bind(C (M::\*member)(A1, A2), T \*instance, EventFlag flag = EventFlag::DEFAULT) {

\_addToList(\_identify(instance, member), std::bind(member, instance, \_1, \_2), flag);

}

*//*

*// @brief Subscribes a member function with 3 arguments.*

*// Event handler arguments: A1, A2, A3.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1, **typename** A2, **typename** A3>

void bind(C (M::\*member)(A1, A2, A3), T \*instance, EventFlag flag = EventFlag::DEFAULT) {

\_addToList(\_identify(instance, member), std::bind(member, instance, \_1, \_2, \_3), flag);

}

*//*

*// @brief Subscribes a member function with 4 arguments.*

*// Event handler arguments: A1, A2, A3, A4.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1, **typename** A2, **typename** A3, **typename** A4>

void bind(C (M::\*member)(A1, A2, A3, A4), T \*instance, EventFlag flag = EventFlag::DEFAULT) {

\_addToList(\_identify(instance, member), std::bind(member, instance, \_1, \_2, \_3, \_4), flag);

}

#pragma mark - unbind(...) overloads

*//*

*// @brief Unsubscribes a member function by its function pointer address as key.*

*// @param member - member function pointer &C::M.*

*// @param instance - instance of structure housing the member function.*

*// @usage <i>event.unbind(&MyClass::member, myClassInstance)</i>.*

*//*

**template**<**typename** C, **typename** M, **typename** T>

void unbind(C (M::\*member)(), T \*instance) {

\_removeFromList(\_identify(instance, member));

}

*//*

*// @brief Unsubscribes a member function by its function pointer address as key.*

*// Event handler argument A1.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1>

void unbind(C (M::\*member)(A1), T \*instance) {

\_removeFromList(\_identify(instance, member));

}

*//*

*// @brief Unsubscribes a member function by its function pointer address as key.*

*// Event handler arguments: A1, A2.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1, **typename** A2>

void unbind(C (M::\*member)(A1, A2), T \*instance) {

\_removeFromList(\_identify(instance, member));

}

*//*

*// @brief Unsubscribes a member function by its function pointer address as key.*

*// Event handler arguments: A1, A2, A3.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1, **typename** A2, **typename** A3>

void unbind(C (M::\*member)(A1, A2, A3), T \*instance) {

\_removeFromList(\_identify(instance, member));

}

*//*

*// @brief Unsubscribes a member function by its function pointer address as key.*

*// Event handler arguments: A1, A2, A3, A4.*

*//*

**template**<**class** **C**, **class** **M**, **class** **T**, **typename** A1, **typename** A2, **typename** A3, **typename** A4>

void unbind(C (M::\*member)(A1, A2, A3, A4), T \*instance) {

\_removeFromList(\_identify(instance, member));

}

**private**:

#pragma mark - Address conversion

*//*

*// @brief Unions a generic lvalue object address and an integral identifier.*

*//*

**template**<**typename** T>

**union** AddressCast {

**explicit** AddressCast(T \_type) : type(\_type) { }

T type;

Identifier address;

};

*//*

*// @brief Returns a unique identifier for a given member function pointer and instance pointer.*

*//*

**template**<**class** **C**, **class** **M**>

**inline** **static** Identifier \_identify(C \_class, M \_member) {

**return** AddressCast<C>(\_class).address \* 10 + AddressCast<M>(\_member).address;

}

#pragma mark - Private inline functions

*//*

*// @brief Validates prepared function object and adds it to the container.*

*// Contract: handlers with matching identifiers present in the container are silently ignored.*

*//*

**inline** void \_addToList(Address a, Handler h, EventFlag flag) {

*// disallow non-unique handlers*

**if** (flag == EventFlag::ONLY\_UNIQUE && hasSubscriber(a)) **return**;

subscribers.insert(std::make\_pair(a, h));

}

*//*

*// @brief Removes bound function object from the container by its address.*

*// Contract: non-existent member identifier is silently ignored.*

*//*

**inline** void \_removeFromList(Address a) {

**auto** it = subscribers.find(a);

**if** (it != subscribers.end())

subscribers.erase(it);

**else** {

*// TODO handle*

}

}

*//*

*// @brief Pairs bound subscribers with their unique memory address.*

*// Container has to be unordered for the handlers to be called sequentially.*

*//*

std::unordered\_map<Identifier, Handler> subscribers;

};

#endif

#### 사용법

**struct** WidgetEventArgs {

**explicit** WidgetEventArgs(std::string \_reversedString) : reversedString(\_reversedString) { }

std::string reversedString;

};

**class** **Widget** {

**public**:

void reverseString(std::string s) {

**auto** reversedS = std::string(s.rbegin(), s.rend());

*// fire stringReversed event*

*// method 1:*

stringReversed.fire(WidgetEventArgs(reversedS));

*// method 2:*

stringReversed(WidgetEventArgs(reversedS));

}

**Event**<WidgetEventArgs> stringReversed;

};

**class** **Consumer** {

**public**:

Consumer() {

*// event subscription to a member function*

\_widget.stringReversed.bind(&Consumer::\_onStringReversed1, **this**);

\_widget.stringReversed.bind(&Consumer::\_onStringReversed2, **this**);

*// ignore subscription if handler function already subscribed:*

\_widget.stringReversed.bind(&Consumer::\_onStringReversed2, **this**, EventFlag::ONLY\_UNIQUE);

*// unsubscribe event*

\_widget.stringReversed.unbind(&Consumer::\_onStringReversed2, **this**);

*// subscribe a lambda*

\_widget.stringReversed += [](**auto** e) {

std::cout << e.reversedString << " from lambda**\n**";

};

\_widget.reverseString("Hello");

}

**private**:

Widget \_widget;

void \_onStringReversed1(WidgetEventArgs e) {

std::cout << e.reversedString << 1 << std::endl;

}

void \_onStringReversed2(WidgetEventArgs e) {

std::cout << e.reversedString << 2 << std::endl;

}

};

int main() {

Consumer c;

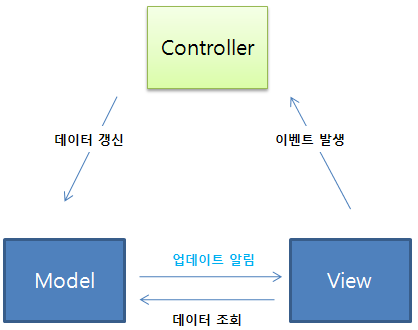
**return** 0;

}

Widget이라는 event class를 관리한다. 이 안에는 EventHandler(stringReversed)가 있어 함수를 추가하고 제거, 또 구독하는 함수들을 수행할 수 있다.

### MVC패턴

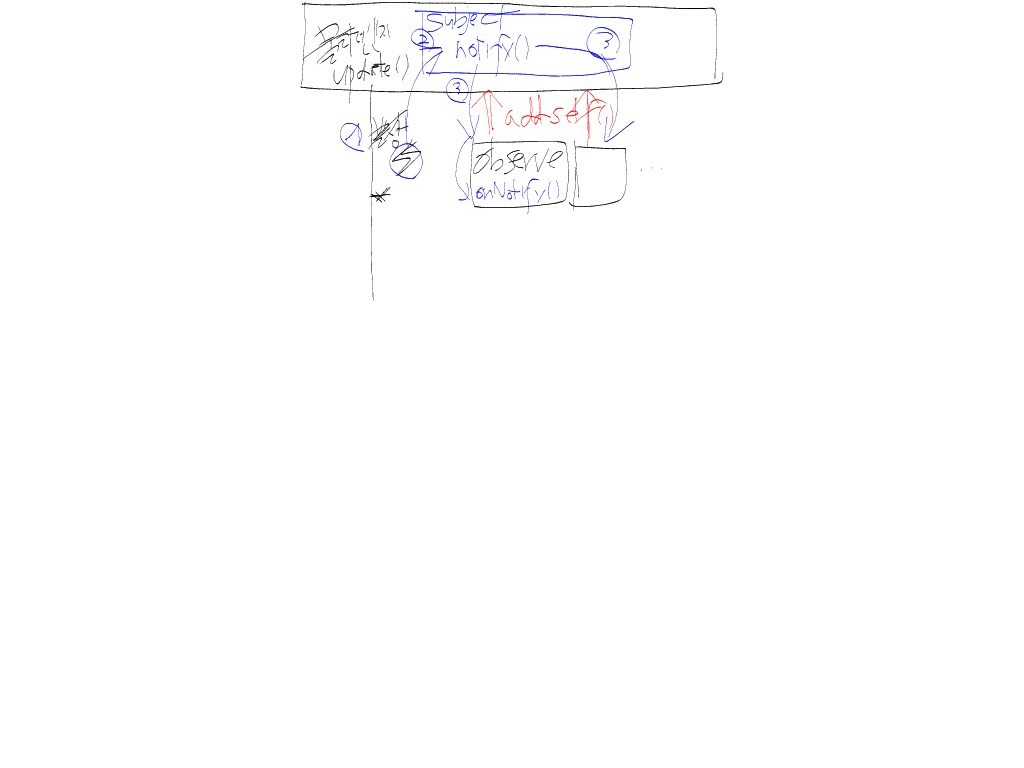
모델-뷰-컨트롤러(Model-View-Controller, MVC)의 기반은 관찰자 패턴이다. 아래와 같이 정의된다.



MVC패턴은 기본적으로 모든 활동을 컨트롤러가 통제한다.

## 동작과정

전반적인 과정이 event와 비슷하다. 하지만 관찰자 패턴은 관찰자로서 직접 핸들러를 추가하지만, event에서는 핸들러가 관찰자들을 수용한다는 관점의 차이가 있다.



## 세부사항

### 동기화

관찰자 패턴은 동기적이다. 대상이 관찰자를 직접 호출하기 때문이다. 그러므로 멀티스레드를 사용하는 엔진이라면 [15. 이벤트 큐]를 이용해 비동기적 상호작용을 고려해야 한다.

### 동적 할당

실은 이 패턴은 동적 할당을 하지 않는다. 포인터로 가리키는 대상의 함수만 호출할 뿐, 메모리 할당은 일어나지 않는다.

### 리스트 노드 풀

관찰자를 감싸는 간단한 객체를 만들어(RAII) 이 객체가 여러 관찰자를 가리킬 수 있게 한다. 같은 관찰자를 동시에 여러 대상에 추가할 수 있다는 의미다. 추가적인 동적 할당의 문제를 피하려면 – [19. 객체 풀 패턴]

### 대상과 관찰자 제거

**대상 제거**

어떠한 관찰자도 대상을 가리키지는 않기 때문에 특별히 심각한 문제는 없지만, 대상이 삭제되었는데 계속해서 관찰자들이 알림을 기다릴 수 있다. 이럴 경우에는 대상이 소멸할 시, 관찰자들에게 알림을 주어 더 이상 기다리지 않도록 해결한다.

**관찰자 제거**

관찰자가 삭제될 때, 스스로 등록을 취소하는 RAII방식을 취한다. 더 나아가서 모든 대상에게 해당 관찰자를 삭제하라고 명령하는 것은 실수를 막을 수 있다.

**사라진 리스너 문제(Lapsed listener problem)**

From Wikipedia, the free encyclopedia

The **lapsed listener problem** is a common source of [memory leaks](https://en.wikipedia.org/wiki/Memory_leak) for [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) programming languages, among the most common ones for [garbage collected](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) languages.[[1]](https://en.wikipedia.org/wiki/Lapsed_listener_problem#cite_note-1)[[2]](https://en.wikipedia.org/wiki/Lapsed_listener_problem#cite_note-2)

It originates in the [observer pattern](https://en.wikipedia.org/wiki/Observer_pattern), where observers (or listeners) register with a subject (or publisher) to receive events. In basic implementation, this requires both explicit registration and explicit deregistration, as in the [dispose pattern](https://en.wikipedia.org/wiki/Dispose_pattern), because the publisher holds strong references to the observers, keeping them alive. The leak happens when a listener fails to unsubscribe from the publisher when it no longer needs to listen. Consequently, the publisher still holds a reference to the observer which prevents it from being garbage collected — including all other objects it is referring to — for as long as the publisher is alive, which could be until the end of the application.

This causes not only a memory leak, but also a performance degradation with an "uninterested" observer receiving and acting on unwanted events. **This can be prevented by the subject holding**[**weak references**](https://en.wikipedia.org/wiki/Weak_reference)**to the observers, allowing them to be garbage collected as normal without needing to be unregistered.**

이를 막기 위해서는 std::shared\_ptr처럼 weak참조(객체를 참조만 할 뿐, reference count를 올리지 않는 참조. 참조 하는 객체가 있다 해도 필요 없다면 삭제할 수 있다.)를 관리하여 좀비 객체가 생기는 것을 막는다.

## 구현

#include *<iostream>*

#include *<list>*

#include *<unordered\_set>*

**class** **Event**{

**public**:

Event(int n) : type(n) {}

**static** Event EVENT\_OBJECT\_HELL(){ **return** Event(1); }

bool **operator**==(**const** Event& other){

**return** type == other.type;

}

**private**:

int type;

};

**class** **Object**{

**public**:

**virtual** ~Object() {}

**virtual** void update() = 0;

};

**class** **Observer**{

**public**:

**virtual** ~Observer(){}

**virtual** void onNotify(**const** Object& object, Event e) = 0;

};

**class** **Subject**{

**public**:

void **operator**+=(Observer\* observer){

addObserver(observer);

}

void **operator**-=(Observer\* observer){

removeObserver(observer);

}

void notify(**const** Object& object, Event e){

**for** (**auto** iter = observer\_list.begin();

iter != observer\_list.end(); iter++){

(\*iter)->onNotify(object, e);

}

}

**private**:

void addObserver(Observer\* observer){

observer\_list.insert(observer);

}

void removeObserver(Observer\* observer){

observer\_list.erase(observer);

}

std::unordered\_set<Observer\*> observer\_list;

int countObserver;

};

**class** **Physics**{

**public**:

Subject& objectFell() { **return** subject; }

void update(Object& object){

object.update();

**if** (!onSurface){

subject.notify(object, Event::EVENT\_OBJECT\_HELL());

}

}

**private**:

bool onSurface;

Subject subject;

};

**template**<**typename** Observer>

**class** **Node**{

**public**:

Node(Observer\* \_observer, Subject\* \_subject)

: observer(\_observer), subject(\_subject) {}

~Node(){

**if** (subject != **nullptr**)

\*subject -= observer;

**delete** observer;

}

Observer\* getObserver() { **return** observer; }

**private**:

Observer\* observer;

Subject\* subject;

};

**class** **Achievement** : **public** Observer{

**public**:

**template**<**typename** Target>

void addSelf(Target& target){

target.objectFell() += **this**;

}

**template**<**typename** Target>

void removeSelf(Target& target){

target.objectFell() -= **this**;

}

**virtual** void onNotify(**const** Object& object, Event e){

**if** (e == Event::EVENT\_OBJECT\_HELL()){

unlock();

}

}

**private**:

void unlock(){

std::cout << "achievement unlock.**\n**";

}

};

**class** **Archer** : **public** Object{

**public**:

Archer() {}

**virtual** void update(){

std::cout << "do something.**\n**";

}

**private**:

};

int main(){

Physics physics;

Node<Achievement> achievement(**new** Achievement, &physics.objectFell());

*//Physics(target)에 나 자신(Observer)을 구독자로 등록한다.*

achievement.getObserver()->addSelf(physics);

Archer archer;

*//물리 엔진으로 archer를 작동시킨다.*

physics.update(archer);

**return** 0;

}

## 관련자료

### 최신 방식: 클로저

위의 코드에서는 Observer라는 객체를 추가하고 제거하지만, 최신 event(C#)에서는 클로저(함수 객체)를 추가, 제거한다. 이는 훨씬 더 가볍고 융통성 있게 만든다.

### 데이터 흐름 프로그래밍

언리얼’s 블루프린터 같은 것. 어떤 값이 변경되면 관련 UI요소나 속성을 바꿔야 하는 작업들을 알아서 해준다.

### 함수형 반응형 프로그래밍

Function Reative Programming(FRP)는 다양한 시각에서 바라볼 수 있다.

* 널리 사용되는 관찰자 패턴(혹은 리스너나 콜백)의 대용품
* 이벤트기반 로직을 코딩하는 합성 가능한 모듈식 방법
* 다른 사고방식. 프로그램을 입력에 대한 응답 혹은 데이터의 흐름으로 표현한다.
* 프로그램 상태(state) 관리에 질서를 가져다 준다.
* 뭔가 근본적인 것. 관찰자 패턴으로 문제를 해결하려던 사람이라면 누구나 결국에는 FRP를 창시하게 됐을 것이다.
* 표준적인 프로그래밍 언어의 경량 라이브러리로 구현이 가능하다.
* 상태유지(stateful) 로직에 사용하기 위한 튜링완전 임베디드 언어로 볼 수 있다.

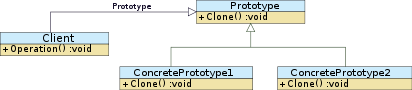
입출력 부분을 빼면, 가령 어떤 복잡한 게임이든 전적으로 FRP로 작성할 수 있다. 그만큼 강력하고 표현력이 있다. 그렇다고 FRP가 ‘모 아니면 도’라는 말은 아니다. 기존 프로젝트에 쉽게 도입할 수 있고, 도입하는 규모도 원하는 만큼 정할 수 있다. (2~3쪽)

<http://feelyou.tistory.com/entry/%ED%95%A8%EC%88%98%ED%98%95-%EB%B0%98%EC%9D%91%ED%98%95-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8DFRP%EC%9D%B4%EB%9E%80>

5. 프로토 타입

## 사전지식

[소프트웨어 디자인 패턴](https://ko.wikipedia.org/wiki/%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4_%EB%94%94%EC%9E%90%EC%9D%B8_%ED%8C%A8%ED%84%B4)에서 **프로토타입 패턴**(prototype pattern)은 생성할 객체들의 타입이 프로토타입인 인스턴스로부터 결정되도록 하며, 인스턴스는 새 객체를 만들기 위해 자신을 복제(clone)하게 된다.

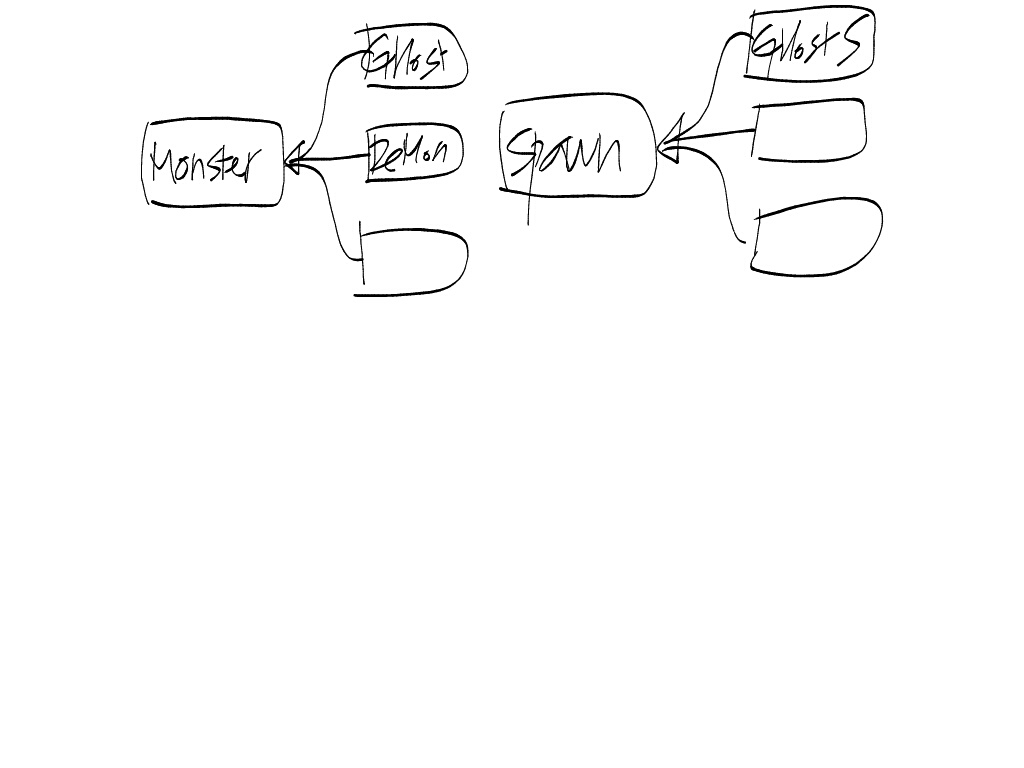


## 동작과정

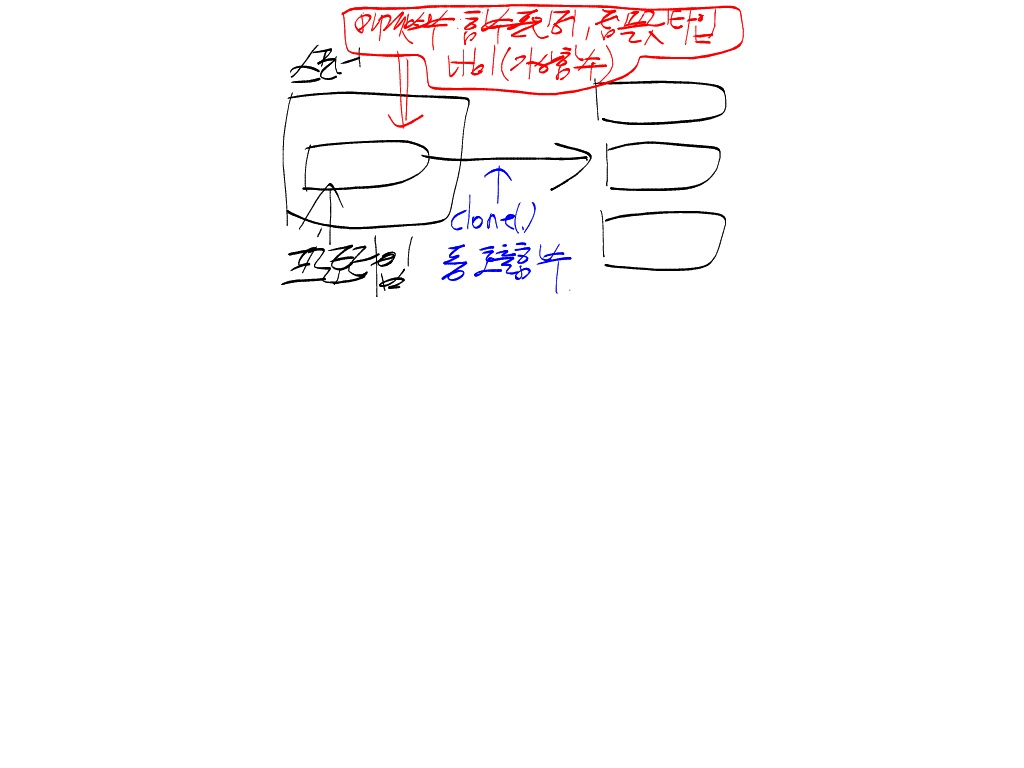
### 클래스 vs. 프로토타입

동작과정을 보면 상속받는 과정과 비슷해 보이는데 프로토타입은 객체자체를 복사하여 담고 있는 것이다. 상속은 그 형틀만 가지고 있다.

몬스터들과 각 몬스터들을 생성하는 클래스들(클래스 6개)



프로토타입 패턴을 이용하면 아래처럼 구현할 수 있다.



스포너는 몬스터를 만들어낸다. 클래스가 형틀을 가지고 새로운 객체를 만들어냈다면 포로토타입은 객체 자체를 그대로 복사하는 것이다. 스포너가 프로토타입에 초기화시킨 객체의 Clone()을 호출해서 말이다. 그러니 타입은 하나만 있으면 된다. 그의 여러 인스턴스를 만들어서 다른 객체를 복사하게 만든다.

핵심은 어떤 객체가 자기와 비슷한 객체를 스폰할 수 있다는 점이다. 이 복사 함수는 각 몬스터 객체에 들어있어야 하므로 상태도 같이 복사할 수 있다.

이 프로토타입 자리에 인자를 **함수 포인터나 템플릿**으로 처리할 수 있다.

## 구현

이 장에서는 구현이 그다지 필요할 것 같지 않다. 개념만 이해하고 넘어간다.

**class** **Archer**{

**public**:

Archer() {}

**virtual** ~Archer() {}

**virtual** Archer\* clone() { **return** **new** Archer; }

};

**class** **BladeMaster** : **public** Archer{

**public**:

BladeMaster() {}

**virtual** Archer\* clone() { **return** **new** BladeMaster; }

};

**class** **Factory**{

**public**:

Factory(Archer\* \_archer) : archer(\_archer) {}

Archer\* makeUnit() { **return** archer->clone(); }

**private**:

Archer\* archer;

};

*/\*클래스로는 각 객체마다 상속하여 구현해야 한다.*

*class Factory\_BladeMaster : public Factory{*

*public:*

*//각 클래스가 자신의 유닛을 생성한다.*

*virtual Archer\* makeUnit();*

*};\*/*

**template**<**typename** TA>

**class** **ItemFactory**{

**public**:

Factory() {}

TA\* makeUnit() { **return** **new** TA; }

};

**class** **Skill**{};

Skill\* makeSkill(){

**return** **new** Skill;

}

std::function<Skill\*()> f\_makeSkill = makeSkill;

Skill\*(\*raw\_func)() = makeSkill;

**typedef** Skill\* (\*f\_p)();

**class** **SkillLab**{

**public**:

SkillLab(std::function<Skill\*()> f) : func(f) {}

Skill\* buildSkill() { **return** func(); }

**private**:

std::function<Skill\*()> func;

Skill\* (\*raw\_func);

f\_p raw\_func2;

};

위에서 스킬을 만들기 위해서 **SkillLab**객체를 생성해야 한다. 이 하나의 클래스만으로도 내부의 func인스턴스만 교체하여 다른 스킬을 생성하는 인스턴스를 만들어낼 수 있다.

## 참고자료

프로토타입보다는 [14. 컴포넌트]나 [13. 타입 객체]를 사용하므로 참고하도록 한다.

### OOP: class vs. self

***class***

개념: 클래스는 인스턴스를 찍어내는 거푸집이다. 어떤 동작이나 의미를 갖는 것은 클래스지만 구체적인 상태는 인스턴스 안에 있다.

상속: 다형성, 코드 재사용의 핵심 개념이다.

***self***

개념: 객체에 모든 것이 들어있다. 모든 데이터를 묶어서 하나로 묶었다고 보면 된다.

위임: 해당 객체에서 어떤 필드나 행위가 있는지 찾아본다. 없으면, 상위 객체에서 찾도록 보낸다. 이것이 위임이다.

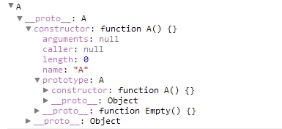
### 자바스크립트: 프로토타입 기반 방식

#### 자바 스크립트 간략한 특징

1. 자바스크립트에서는 자료형을 정의하는 객체로부터 new를 호출하는 생성자 함수를 통해 객체를 생성한다.  
2. 상태는 인스턴스 그 자체에 저장된다.  
3. 동작은 자료형이 같은 객체 모두가 공유하는 메서드 집합을 대표하는 별도 객체인 프로토타입에 저장되고 위임을 통해 간접 접근된다.

#### 자바스크립트의 프로토타입(Prototype) 란?

자바스크립트의 모든 객체는 자신을 생성한 객체 원형에 대한 숨겨진 연결을 갖는다. 이때 자기 자신을 생성하기 위해 사용된 객체원형을 프로토타입이란 한다. 자바스크립트의 모든 객체는 Object 객체의 프로토타입을 기반으로 확장 되었기때문에 이 연결의 끝은 Object 객체의 프로토타입 Object 다.



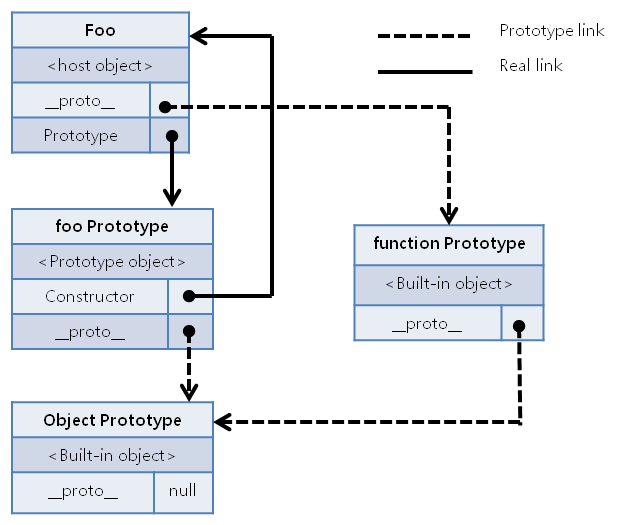
#### 자바스크립트의 프로토타입(Prototype) 프로퍼티 란?

모든 함수 객체의 Constructor는 prototype 이란 프로퍼티를 가지고 있다. 이 prototype 프로퍼티는 객체가 생성될 당시 만들어지는 객체 자신의 원형이 될 prototype 객체를 가리킨다. 즉 자신을 만든 원형이 아닌 자신을 통해 만들어질 객체들이 원형으로 사용할 객체를 말한다. prototype object는 default로 empty Object 를 가리킨다.

function foo() {}

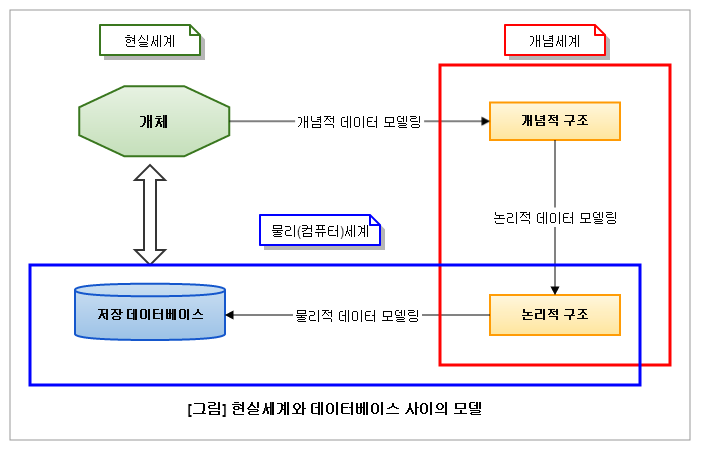
var foo = new foo();

위 예제코드를 통해 만들어지는 Prototype Link 와 Prototype Property가 가리키는 Prototype Object의 Real Link 에 대한 관계도



### 데이터 모델링

**데이터 모델링**(data modeling)이란 주어진 개념으로부터 논리적인 [데이터 모델](https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%84%B0_%EB%AA%A8%EB%8D%B8)을 구성하는 작업을 말하며, 일반적으로 이를 물리적인 [데이터베이스 모델](https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%B2%A0%EC%9D%B4%EC%8A%A4_%EB%AA%A8%EB%8D%B8)로 환원하여 고객의 요구에 따라 특정 [정보 시스템](https://ko.wikipedia.org/wiki/%EC%A0%95%EB%B3%B4_%EC%8B%9C%EC%8A%A4%ED%85%9C)의 [데이터베이스](https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%B2%A0%EC%9D%B4%EC%8A%A4)에 반영하는 작업을 포함한다. 후자의 의미로 흔히 **데이터베이스 모델링**으로 불리기도 한다.



거창한 개념이다. 여기서는 간략하게 프로토타입과 위임을 활용해 데이터를 재활용하여 데이터 모델링하는 기법을 소개한다.

#### 프로토타입과 위임을 활용해 데이터 모델링하는 기법: JSON의 사용

출처: <http://luckyyowu.tistory.com/303> [요우의 내맘대로 블로그]

[JSON](http://json.org/json-ko.html)은 **J**ava**S**cript **O**bject **N**otation의 약자로서 다른 언어간에 객체를 전달하는 포맷으로 많이 사용된다. 주로 클라이언트 사이드에서 서버 사이드로 객체 정보를 전송할 때 쓰인다고 한다. JSON의 개념이 없는 상태면 어려울 수도 있지만 막상 공부해보니 하나도 안 어렵다.

다음 소스코드는 일반적인 자바스크립트에서의 **배열**과 **객체**의 사용방법 예제다.

|  |  |
| --- | --- |
|  | <head> |
|  | <title>JSON Basic</title> |
|  | <meta charset="utf-8"/> |
|  | </head> |
|  | <body> |
|  | <script> |
|  | /\* Array and Object Definition \*/ |
|  | var arr = [ "Yongwoo", 25, "Catholic Univ, of Korea", "Computer Science" ]; |
|  | var obj = { "NAME" : "Yongwoo", "AGE" : 25, "UNIVERSITY" : "Catholic Univ, of Korea", "MAJOR" : "Computer Science" }; |
|  |  |
|  | /\* Array and Object Value Check \*/ |
|  | document.write("<b>Normal String Arrary :</b> ", arr, "<br/>"); |
|  | document.write("<br/><b>Normal Object</b><br/>"); |
|  | for(var objVarName in obj) { |
|  | document.write(objVarName, " : ", obj[objVarName], "<br/>"); |
|  | } |
|  | </script> |
|  | </body> |

JSON의 표현방법도 위와 크게 다르지 않지만**객체 자체를 문자열로 바꿀 수 있다는 특징**이 있다. 이 객체 정보를 가진 문자열을 다른 언어나 서버로 던져 다시 사용할 수 있다. 다음은 기초적인 JSON 사용 예제다.

|  |  |
| --- | --- |
|  | <head> |
|  | <title>JSON Basic</title> |
|  | <meta charset="utf-8"/> |
|  | </head> |
|  | <body> |
|  | <script> |
|  | /\* JSON Definition \*/ |
|  | var json = '{ "NAME" : "Yongwoo", "AGE" : 25, "UNIVERSITY" : "Catholic Univ, of Korea", "MAJOR" : "Computer Science" }'; |
|  |  |
|  | /\* JSON Parse and Check \*/ |
|  | var jsonObj = JSON.parse(json); |
|  | document.write("<br/><b>JSON Object</b><br/>"); |
|  | for(var objVarName in jsonObj) { |
|  | document.write(objVarName, " : ", jsonObj[objVarName], "<br/>"); |
|  | } |
|  |  |
|  | /\* JON Object to JSON String \*/ |
|  | var jsonStr = JSON.stringify(jsonObj); |
|  | document.write("<br/><b>JSON String Format :</b> ", jsonStr, "<br/>"); |
|  | </script> |
|  | </body> |

사실 클라이언트에서만 사용하기에는 큰 메리트가 없지만 다른 언어로 객체의 정보를 보내거나 Ajax를 사용할 때 그 진가가 드러난다고 한다.

6. 싱글톤

## 사전지식

<https://kldp.org/files/static____________160.htm>

**static을 이해하자**

**static 용어의 개념 및 성질**

**static이라는 용어에 대한 사전적 뜻**

static

1 정지(靜止)하고 있는, 변화하지 않는; 정적인, 움직임이 없는(반:dynamic). (또는 statical)

2 〈물리〉 정적인, 정압(靜壓)의.

~ pressure 정압.

3 〈전기〉 정전(靜電)(기(氣))의; 공전(空電)의.

[출처 : 야후 영어사전]

**C/C++에서 static이 갖는 성질들**

1.       정적 지속성(static storage duration): static 객체들은 일단 한번 생성이 되면 프로그램이 종료될 때까지(C++표준에서는 main()함수에서 리턴 될 때 혹은 exit()를 호출했을 때 소멸된다고 명시되어있습니다.) 유지됩니다. 즉, 객체의 소멸 시점이 scope에 영향 받지 않고 항상 일정합니다.

2.       유일성(singleton): 어떤 모듈 단위(function, class, file)에서든지 static 객체는 단 한번만 생성됩니다.

3.       내부 연결성(internal linkage): 전역 static 객체나 함수는 link 단계에서 외부 바인딩이 일어나지 않습니다. 즉, 외부 파일에서는 내부 전역 static 객체/함수를 참조하거나 호출할 수 없습니다.

그렇다면 왜 static 키워드가 붙은 변수나 함수가 이런 성질을 가지게 되는지에 대해서 차근차근 알아보도록 하겠습니다.

**binding(바인딩)**

static에 대해서 이해하기 위해서는 우선 binding이라는 개념을 이해하여야 합니다. 바인딩이라는 단어 역시 다양한 의미로 사용되는 용어인데 프로그래밍 언어에서 말하는 바인딩이란 어떤 심볼(변수나 함수)의 속성을 결정짓는 것을 의미합니다.

int main()

  {

  int x;

  x = 3;

  return 0;

  }

위와 같은 C 소스가 있을 때 이것을 컴파일러가 컴파일 하게 되면 우선 컴파일러는 int x; 라는 선언문을 보고서 x라는 이름의 변수를 자신의 심볼 테이블에 기록을 하며 이 때 그 변수의 속성(type, scope, value 등)을 저장하게 됩니다. 그리고 그 다음에 있는 x = 3; 이라는 구문에서 x라는 변수가 자신의 심볼 테이블에 있는지 살펴보고, 있으면 그 속성을 참고하여 해당 구문의 타입 체크와 같은 문법 검사를 수행하게 됩니다. (물론 x가 심볼 테이블에 없다면 그런 변수를 찾을 수 없다는 컴파일 에러를 발생시킵니다.)

이제 좀 더 복잡한 예를 살펴보겠습니다.

/// test1.cpp

#include <iostream>

int global; // ---> 1)

void extern\_fun(int a);

int main()

  {

  global = 5; // ---> 2)

  extern\_fun(3);

  std::cout << global << '\n';

  return 0;

  }

/// test2.cpp

extern int global;  // ---> 3)

void extern\_fun(int a)

  {

  global += a;      // ---> 4)

  }

 위 소스에서는 global이라는 전역 변수를 두 개의 파일에서 참조하고 있습니다. 위에서 언급했듯이 컴파일러는 변수 선언문이 나타나면 해당 변수를 심볼 테이블에 등록하고 이후에 변수를 사용하는 구문을 만날 때마다 해당 변수가 자신의 심볼 테이블에 등록되어 있는지 살펴보고 그 정보에 따라 문법 검사를 수행하게 됩니다.

 그런데 컴파일러는 항상 파일 단위로 컴파일을 수행하며 새로운 파일을 컴파일 할 때 이전에 이미 컴파일 한 파일에 대한 정보 같은 것들은 따로 기록하지 않습니다. 즉, 항상 새로운 마음가짐으로 각각의 파일들을 독립적으로 컴파일 하게 됩니다.

그래서 위의 경우 test1.cpp를 먼저 컴파일 했다고 해서 **test2.cpp에서 global변수를 사용하는 구문을 컴파일 할 때 test1.cpp에서 만든 심볼 테이블을 참조하지는 않습니다.**

 때문에 두 개 이상의 파일에서 같은 전역 변수를 참조하게 되면 그런 사실을 미리 소스 코드를 통해 컴파일러에게 알려줘야 합니다. 만약 그렇지 않으면 test2.cpp에서 4)구문을 컴파일 할 때 global이라는 변수가 선언되어 있지 않다 라고 에러를 발생시키거나 혹은 test1.cpp와 test2.cpp의 global변수를 별개의 변수로 취급하게 될 것입니다.

 그러므로 프로그래머는 사전에 test1.cpp와 test2.cpp에서 사용하는 global 변수가 동일한 것이다라는 사실을 컴파일러에게 알려주게 되는데 위의 소스에서 3)에 있는 extern은 바로 그런 역할을 수행하는 키워드입니다.

extern의 의미는 '해당 변수는 외부에서 참조하여 사용하겠다'라는 뜻입니다. 만약 3)이 없다면 test2.cpp를 컴파일 할 때 4)에서 global이라는 변수를 찾을 수 없다는 에러가 발생할 것이며, 3)에서 extern이 없다면 test1.cpp와 test2.cpp의 global은 별개의 변수로 취급되어 프로그램 결과가 8이 아니라 5가 나오게 될 것입니다.

 그렇다면 컴파일러는 각각의 파일을 컴파일 할 때 다른 파일의 심볼 값들을 참조하지 않음에도 불구하고 **어떻게 전역 변수를 공유할 수 있을까요?** 그것은 바로 링크 과정이 있기 때문에 가능합니다.

컴파일 시 **컴파일러는** 전역 변수나 함수에 대한 참조/호출을 직접적인 어셈블리어 구문으로 변환하는 것이 아니라(다시 말하면 해당 변수의 메모리 주소나 함수의 코드 주소로 바로 변환하지 않고) **특정한 이름을 부여하여 해당 이름의 전역 변수나 함수에 대한 참조/호출 구문으로 바뀌게 되는 것**입니다.

 예를 들어 위의 소스의 경우에 test2.cpp에서 참조하는 global이 외부에서 참조되는 변수이다라는 사실을 컴파일 한 목적 코드(object code)에 기록해 두면 링커는 해당 변수와 동일한 이름으로 정의된 파일을 검색하고 test1.cpp에서 일치된 이름을 발견하면 해당 전역 변수 참조 구문들을 올바른 속성(상대 주소)값으로 바꾸게 됩니다.

 이렇게 어떤 변수나 함수의 속성값을 결정짓는 과정을 바인딩이라고 합니다. 그리고 위의 경우처럼 링크 과정에서 그런 속성값을 결정짓는 것을 링크 바인딩(link binding), 혹은 동적 바인딩(dynamic binding)이라 합니다.

 위의 야후 사전에서도 나와 있듯이 컴퓨터 분야에서 dynamic에 반대되는 용어는 static이며 거의 항상 이 두 존재는 양립합니다. 즉, 앞에 dynamic이라는 단어가 붙는 어떤 용어가 있다면 거의 항상 static이 앞에 붙는 반대되는 의미의 용어가 존재합니다.

바인딩 역시 예외가 아니어서 동적 바인딩에 반대되는 개념으로 정적 바인딩(static binding)이 있습니다. 이것은 바인딩이 컴파일 시점에 이루어 지는 것을 의미합니다. 즉, 해당 변수나 함수의 속성이 컴파일 과정에서 결정됩니다.

정적 바인딩에 의해 정의되는 변수나 함수는 링크 과정 이전에 이미 속성이 결정되기 때문에 아래와 같은 특징을 가지게 됩니다.

  1. 정적 바인딩 변수는 extern 키워드를 통해 외부 파일에서 참조가 불가능하다.

  2. 정적 바인딩 함수는 외부 파일에서 호출이 불가능하다.

static 전역 변수나 함수를 사용해 보신 분들은 아시겠지만 위의 특징은 바로 static 전역 변수와 함수의 특징과 일치합니다. 즉, static 키워드는 해당 전역 변수나 함수를 컴파일러가 정적으로 바인딩을 하도록 프로그래머가 '지시'하는 역할을 합니다.

컴파일 타임 바인딩을 정적(static) 바인딩이라고 말하는 이유는 아마도 한번 컴파일 시에 속성이 결정되고 나면 '변하지 않는다' 라고 하는 불변성 때문이 아닌가 추측됩니다. 어쨌든 이런 특징을 가지고 있기에 static 변수는 독특한 성질을 지니고 있습니다.

 아래의 예제는 초보자가 흔히 하게 되는 실수입니다.

/// header.h

int global;

/// a.cpp

#include <iostream>

#include "header.h"

void b\_func();

int main()

  {

  global = 3;

  b\_func();

  std::cout << global << std::endl;

  return 0;

  }

/// b.cpp

#include "header.h"

void b\_func()

  {

  global = 5;

  }

 위 소스는 정상적으로 컴파일 됩니다.

 앞에서 언급했듯이 컴파일러는 파일 별로 독립적으로 컴파일을 수행합니다. 따라서 a.cpp 컴파일 시 header.h에 있는 global 선언문을 통해 global 변수를 하나 생성하여 3을 할당합니다. 또한 b.cpp 역시 header.h에 있는 global 선언문을 통해 global 변수를 하나 생성하여 5를 할당합니다. 따라서 둘 다 적법한 소스입니다.

 하지만 링크 과정에서 링커는 a.cpp와 b.cpp가 동일한 이름(global)의 전역 변수를 두 개 생성한 것을 보고 링크 에러를 발생시킵니다. (이 때 발생하는 에러는 redefinition error입니다. 아마 초보자들이 가장 많이 접하는 에러 중 하나일 것입니다.)

 그러면 위 소스를 아래와 같이 수정해 보겠습니다.

// header.h

static int global; /// int global을 static 전역 변수로 수정

// a.cpp

///원 소스와 동일...

// b.cpp

/// 원 소스와 동일....

이렇게 수정하면 정상적으로 컴파일 및 링크가 수행됩니다. 왜냐하면 static 전역 변수는 컴파일 시점에 바인딩이 완료되고 따라서 링크 과정에서 해당 변수에 대한 바인딩 처리를 하지 않으므로 중복 정의를 검사하지 않기 때문입니다.

그러나 대신 a.cpp와 b.cpp에서 사용하는 global 변수는 완전히 별도의 객체로 처리됩니다. 따라서 위 프로그램에서 a.cpp가 출력하는 global값은 - b\_func()함수를 호출함으로써 바뀌는 값- 5가 아니라 -원래 a.cpp에서 할당한 값 - 3이 됩니다. 즉, a.cpp와 b.cpp에서 사용하는 global 전역 변수는 사실 이름만 똑같을 뿐 별개로 취급되는 다른 변수가 됩니다. (따라서 이런 식의 사용은 프로그래머에게 혼란을 줄 뿐이며 버그를 야기시키는 좋지 않은 코딩 방식입니다. 그러므로 static 전역 변수를 선언할 때는 항상 header 파일이 아닌 c/cpp 파일에 해줘야 합니다.)

어쨌든 static 전역 변수는 항상 해당 변수가 선언된 파일 내부에서만 참조가 가능합니다. 그리고 이것은 static 전역 함수 역시 마찬가지이며 static으로 선언된 전역 함수는 외부 파일에서는 사용이 불가능합니다. (만약 외부에서 호출하려고 하면 링크 에러가 발생합니다.)

때문에 보통 C프로그래머들은 외부 파일에서 사용할 필요가 없는 혹은 외부에서 사용하기를 원치 않는 함수들은 static으로 정의하곤 합니다. 이렇게 함으로써 이름 중복에 의한 혼란 등을 피할 수 있는 부수적인 장점을 얻을 수도 있습니다. (C++에서는 namespace와 class가 생기면서 이런 장점이 많이 사라졌습니다.)

그리고 **이렇게 파일 내부에서만 참조 가능한 static 의 성질을 internal linkage(내부 연결성)이라고 부릅니다.**

**storage duration and scope**

다 아시는 이야기 하나 해보겠습니다. 변수는 관점에 따라 다양하게 분류가 될 수 있습니다. 우선 공간(scope)범위에 따라 전역 변수와 지역 변수로 분류가 가능합니다. 그리고 메모리 할당 주체에 따라 정적 할당 변수와 동적 할당 변수로 분류할 수도 있습니다. 또한 메모리 할당 위치에 따라 스택(stack) 변수와 정적 영역 변수, 동적 영역(heap) 변수로 나눌 수 있으며 그 외에도 정적(static)변수, 상수(const)변수,포인터 변수 등등 여러 종류로 구분이 가능합니다.

게다가 이러한 분류는 복합적으로 정의될 수 있습니다. 가령, 정적 전역 변수(static global variable), 정적 지역 변수(static local variable), 전역 상수 변수(global const variable) 등등이 있습니다.

심지어 어떤 분류 정의는 서로 밀접한 관계(혹은 동등한 의미)를 가집니다. 스택 변수와 비정적(non-static) 지역 변수는 사실 같은 의미를 가지고 있으며, 전역(global) 변수와 정적(static) 변수는 정적 영역 변수에 해당합니다.

 이렇게 변수의 종류가 다양하게 분류되는 것은 프로그래밍 시 여러 가지 상황이 발생하게 되고 이때 이런 상황에 맞는 변수 설정을 위한 여러 가지 개념들이 필요했기 때문이며 이러한 여러 개념들이 서로 복합적인 관계를 맺으며 사용되는 것은 그러한 개념들을 구체화 시키는 과정에서 구현 상의 이유로 만들어진 부수효과(side-effect)라 할 수 있습니다.

가령 포트란 같은 경우 모든 변수가 전역 변수로 지정됩니다(참고로 저는 포트란을 직접 사용해 본적은 없습니다.). 즉, 지역 변수라는 개념이 없습니다. 따라서 파라미터 전달이나 스택 변수 생성에 따른 함수 호출 오버헤드가 없기 때문에 간단한 프로그래밍 시 좋은 성능을 보여줍니다.

그러나 대신 일회성 변수를 사용하더라도 지속적인 변수 유지가 필요하기 때문에 이름 짓기(naming) 문제나 혹은 같은 변수를 다른 용도로 계속 재사용하게 되고 따라서 복잡한 프로그래밍 시 코드가 난해해지는 단점이 있습니다.

그래서 C와 같은 프로그래밍 언어에서는 지역 변수라는 개념을 도입했으며 그에 따라 scope라는 개념이 생겨났습니다.

그리고 이런 scope라는 개념이 들어가면서 동시에 storage duration이라는 개념이 생겨났습니다. 더 이상 참조가 필요 없는, 다시 말하면 scope를 벗어난 지역 변수에 대해서 프로그래머가 일일이 지정하지 않아도 자동적으로 해당 지역 변수의 메모리를 컴파일러가 알아서 해제해줌으로써 보다 추상화된 프로그래밍이 가능하게 되는 것입니다.

그래서 이런 지역 변수의 특징을 구체화하는 과정에서 가장 손쉽고 오버헤드가 적게 드는 기법을 구현한 것이 바로 스택을 통한 지역 변수 관리 기법인 것입니다. (따라서 스택 변수는 곧 비정적 지역 변수가 됩니다.)

어쨌든 점점 프로그래밍이 고난이도 작업이 되고 그에 따라 요구 조건이 까다로워지면서 보다 다양한 개념의 메모리 및 변수 관리가 필요하였고 그에 따라 C/C++와 같은 언어에서는 직접 사용자가 메모리를 관리하는 포인터 및 동적 할당 개념이 생겨났습니다.

그런 와중에 '특정 scope를 가지면서 해당 변수의 storage duration은 scope에 상관없이 프로그램 실행 시간 내내 유지 될 수 있는' 그런 기법이 필요하게 되었습니다. 전역 변수는 프로그램 실행 시간 내내 유지되지만 대신 다른 블럭이나 모듈, 파일들에서 해당 변수를 참조할 수 있게 되고 그러면 프로그램의 안정성이 떨어질 수 있기 때문에, 가급적 참조 범위를 최소화하는 것을 미덕으로 여기는 프로그래밍 세계에서는 다른 대안을 필요로 한 것입니다.

한편, static 변수는 앞서 말한 바와 같이 컴파일 시점에 바인딩이 되는 변수입니다. 그런데 이 변수를 전역이 아닌 지역 변수로 선언을 한다면 뭔가 비 논리적인 상황이 발생합니다. 왜냐하면 지역 변수는 스택을 통해 관리되므로 그 속성(여러 속성들이 있겠지만 여기서는 메모리 주소값)이 수시로 바뀌게 되는데 이는 static의 원칙에 어긋나는 동작입니다.

**결국 static 지역 변수는** 허용을 하지 말거나 **혹은 지역 변수와는 독립적인 처리가 필요하게 되었습니다.**그리고 C/C++ 에서는 후자를 선택하였습니다. 그 이유는 아마도 앞서 언급한 필요성 때문이 아닐까 생각합니다. (사실 위의 이야기들은 다 제가 추측한 이야기입니다. 실제로 어떤 이유로 static 지역 변수가 생겨났는지는 K&R만이 알겠죠...)

따라서 static으로 선언된 지역 변수는 다른 지역 변수와 달리 스택이 아니라 정적 영역에 할당이 됨으로써 해당 scope를 벗어나 스택이 해제되더라도 그 상태를 유지할 수 있게 되었습니다. 어차피 static 전역 변수 역시 정적 영역에 할당되므로 구현 상으로도 통일된 처리가 가능하여 이는 충분히 합리적인 결정이라 생각됩니다.

어쨌든 **결국 static 지역 변수는 지역 변수의 일부 특성(scope)과 전역 변수의 일부 특성(static storage duration)을 갖는 독특한 존재가 됐습니다.** 예를 들자면,

#include <iostream>

int sum(int a)

  {

  static int x = 0; /// 최초 sum 호출 시 한번 만 생성&초기화됨

  x += a;

  return x;         /// sum()함수 종료 시에도 소멸 안됨

  }

/\* 1부터 100까지 합을 출력하는 프로그램\*/

int main()

  {

  int i = 0;

  for (; i < 100; ++i)

    sum(i);

  std::cout << sum(100) << '\n';

  return 0;       /// main()함수 종료 시 sum()함수 내에 있는 x 변수 소멸

  }

이런 식의 사용이 가능합니다. 또,

#include <iostream>

int\* local\_fun(int a)

  {

  int x = 0;

  x += a;

  return &x;

int\* static\_fun(int a)

  {

  static int x = 0;

  x += a;

  return &x;

  }

int main()

  {

  int\* p = NULL;

  p = local\_fun(3);

  std::cout << \*p << '\n';   /// 비정상적인 값 출력

  p = static\_fun(3);

  std::cout << \*p << '\n';   /// 정상값(3) 출력

  return 0;

  }

이처럼 지역 변수의 경우 스택을 통해 메모리가 관리되므로 외부 참조가 불가능하지만 **static 지역 변수의 경우 정적 영역에서 메모리가 관리되므로 포인터 혹은 레퍼런스를 통한 참조가 가능합니다.** 여기서scope에 대한 제약은 어떤 실행 메카니즘에 의한 것이 아니라 단지 컴파일러에 의한 문법 검사에 의한 것임을 알 수 있습니다. (이런 특징 때문에 static 객체는 singleton 패턴을 구현하는데 사용되기도 합니다.)

어쨌든 **이렇게 static 객체는 전역 객체로 선언되면 internal linkage 특성을 가지지만 지역 객체로 선언될 경우 static storage duration 특성을 가지게 됩니다.** 그리고 이런 특성이 모두 static이라고 하는 용어가 가진 개념에 의해 발생된 것임을 알 수 있습니다.

**C++에서의 static**

C에서 static이 가진 의미는 위에서 말한 두 가지가 전부입니다. 그러나 C++에서는 객체지향 패러다임이 도입되었고 그에 따라 클래스라고 하는 개념이 도입되었습니다.

클래스는 다 아시다시피 '무언가 공통된 책임을 수행하기 위해 같이 모아 놓으면 좋을 만한 변수와 함수들을 하나의 모듈로 캡슐화시킨 사용자 정의 타입'입니다. 그리고 이런 클래스 타입에 의해 생성된 변수들을 '객체(Object)'라고 합니다.

사실 이런 클래스나 객체라는 것은 이름은 그럴 듯 하지만 막상 그 세부 구조를 밑바닥까지 파헤쳐 보면 C에서 사용하는 구조체(structure)와 큰 차이가 없습니다. 멤버 변수들은 단지 구조체 멤버에 사용자 권한이라고 하는 컴파일 타임 제약 사항만을 추가한 것에 불과하며, 멤버 함수라고 하는 것은 실상 해당 클래스 객체의 포인터를 파라미터로 자동 추가해 주는 편이성 높은 함수에 불과합니다. 즉,

class CppClass

  {

  public:

    CppClass() : x\_(0) {}

    void SetX(int a) { x\_ = a; }

    int x\_;

  }

위의 클래스를 C언어로 바꾸게 되면

struct CStruct

  {

  int x\_;

  }

void CStruct\_ctor(CStruct\* this)

  {

  this->x\_ = 0;

  }

void CStruct\_SetX(CStruct\* this, int a)

  {

  this->x\_ = a;

  }

이렇게 됩니다. 그래서 실제 사용시에

CStruct tmp;

CStruct\_ctor(&tmp);

CStruct\_SetX(&tmp, 3);

이렇게 사용할 것을 클래스를 이용해서

CppClass tmp;

tmp.SetX(3);

이런 식으로 사용함으로써 생성자나 SetX()와 같은 멤버 함수가 CppClass라는 클래스의 객체에 밀접하게 관련되었다는 것을 프로그래머가 보기에 보다 직관적이 될 수 있도록 해준 장치에 불과합니다.

물론 이렇게 맥 빠지게 말을 하였지만 C++, JAVA, Smalltalk들과 같은 객체 지향 언어들은 객체 지향 패러다임을 통해 상속, 다형성, 캡슐화라는 다양한 장치를 제공하여 현재 가장 널리 사용되는 프로그래밍 언어인 것은 틀림없습니다. (적어도 이제 사람들이 더 이상 C를 반드시 배우려고 하지는 않습니다.)

어쨋든 객체 지향 언어들은 모든 프로그래밍을 객체 단위로 구현하고 조직화하게 되며 따라서 모든 변수들은 자신이 속한 객체와 그 생명 주기(life time)를 같이 합니다. 다시 말하면 멤버 변수의 scope와storage duration은 자신이 속한 객체에 영향을 받는다는 뜻입니다.

그런데 여기서도 지역 변수에서와 유사한 문제가 발생합니다. 즉, 클래스 멤버 변수 선언 시에 static을 앞에 붙히게 되면 어떻게 되느냐 하는 것입니다.

클래스 멤버 변수는 객체가 생성될 때 같이 생성되고 객체가 소멸될 때 같이 소멸됩니다. 게다가 객체가 지역 변수로 선언되면 멤버 역시 스택에 위치하게 되며 객체가 동적 할당되면 멤버 역시 동적 영역에 위치합니다.

그런데 앞서 언급했듯이 static은 지역 변수든 전역 변수든 상관없이 static storage duration을 갖습니다. 따라서 아무리 객체 지향이라 하더라도 이러한 기존의 법칙을 거스르는 행동을 하는 것은 논리적이지 못합니다.

따라서 이것 역시 두 가지 선택이 필요하게 되었습니다. **멤버 변수에 대한 static을 허용하느냐, 그렇지 않느냐...그리고 C++은 전자**를 선택하였습니다. 단 이렇게 하고 나니 한 가지 문제가 발생하였습니다. static 멤버 변수를 허용하게 되면 기존의 static 특성을 따르기 위해서 static storage duration을 가져야 하고 그러려면 이 변수는 정적 영역에 위치하여야 하는데 객체는 반드시 정적 영역에 위치하리라는 보장이 없기 때문입니다.

**결국 이런 모순을 해결하기 위해서는 static 멤버 변수를 객체에서 분리하여야 합니다.** 즉, 객체의 부분 집합으로써가 아니라 단지 클래스의 이름 공간에 한정 받는 전역 변수처럼 취급이 되는 것입니다. 다행스럽게도 이것은 기존의 static 정의에서 크게 벗어나지 않는 개념입니다. 즉,

  1. static 변수는 한 번 생성되면 프로그램 종료 시까지 계속 유지된다. (static storage duration)

  2. static 변수는 scope에 영향을 받는다.

    - 전역 객체는 정의된 파일 scope에 한정되어 참조 가능하며(internal linkage), 지역 객체는 정의된 local scope에 한정되어 참조 가능하다. (no linkage)

static 클래스 멤버 변수는 위의 정의에서 1번과 일치하며 2번의 경우 영향 받는 scope를 클래스 이름공간(namespace)라는 것으로 확장하여 생각하면 역시 문제될 것이 없습니다.

**결국 static 클래스 멤버 변수는 객체와 상관없이 클래스 범위 연산자(::)를 통해서 참조가 가능한 독특한 특징을 가진 전역 객체**가 되었으며 동시에 **'객체가 몇 개가 생성되든 오직 하나만 존재함(singleton)'이라는 성질**을 가지게 되었습니다.

실제 프로그래밍에서 static 멤버 변수는 어떤 클래스에 관련되지만 특정 객체에 영향을 받지 않는 성질을 가진 데이터를 취급하는데 아주 유용하게 사용됩니다. reference counter가 그 대표적인 예라 할 수 있습니다.

한편 클래스 멤버 함수는 다른 경우가 됩니다. **함수라는 것은 어차피 storage duration이라는 것이 존재하지 않기 때문에 일반 함수의 경우 static, non-static의 구분이 'internal linkage인가 아니면 external linkage인가'로 결정이 됩니다.**

여기서 클래스 멤버 함수의 경우는 internal linkage라는 것이 큰 의미가 없습니다. 어차피 private이나 protected와 같은 권한 지정 키워드가 있기 때문에 얼마든지 사용 범위에 제한을 가할 수 있기 때문입니다. 따라서 이것 역시 두 가지 선택 사항이 존재하게 됩니다. 멤버 함수에 대해서 static을 허용하느냐 그렇지 않느냐...C++는 역시 허용하는 쪽에 손을 들어 줍니다.

사실 static이라는 키워드는 기본적으로 모든 선언문 형식에 들어갈 수 있는 storage class specifier라고 하는 지정자의 일종이기 때문에(C에서 그렇게 정의했기 때문에) 자꾸 이런 저런 제약을 가하는 것은 C의 자유로운 문법을 계승한 C++ 입장에서 그다지 바람직하지 않다라고 생각했을 것입니다.

어쨌든 멤버 함수에 static을 허용하였고 그에 따른 어떤 의미 부여가 필요하였습니다. (앞서 말한 대로 internal linkage라는 성질은 큰 의미가 없으므로...)

여기서 C++은 멤버 변수가 객체에 영향을 받지 않는 클래스 이름 공간에 속한 전역 객체로써 취급되었듯이 멤버 함수 역시 동일한 성질을 부여하게 됩니다. 즉, static 멤버 함수는 객체가 없어도 호출이 가능하며 단지 클래스 이름 공간에 한정을 받는 함수가 된 것입니다. 따라서 static 멤버 함수는 다른 멤버 함수처럼 this 포인터를 암시적으로 받는 특성이 없이 일반 함수와 똑같은 호출 구조를 가지게 되었습니다. 결국

class StaticClass

  {

  public:

    static int x\_;

    static int func() {};

  }

이것은

namespace gimmesilver

  {

  extern int x;

  int func() {};

  }

이것과 거의 동일한 특성을 가집니다. 따라서 non-static 멤버 함수들은 암묵적으로 넘겨받은 this 포인터를 통해서 해당 객체의 멤버 변수나 다른 non-static 멤버 함수를 참조/호출할 수 있지만 static 멤버 함수들은 그러한 참조할 만한 객체 포인터가 없으므로 객체의 멤버 변수나 non-static 멤버 함수를 참조/호출할 수 없는 것입니다. (이것 역시 초보자들이 흔히 하는 실수입니다.)

**Post face**

정리 들어갑니다...

static 변수

  1. static storage duration을 갖는다.

  2. scope의 영향을 받는다.

           - 전역 변수 : file scope의 영향을 받으며 internal linkage라고 한다.

           - 지역 변수 : block scope의 영향을 받는다.

           - 클래스 멤버 변수 : 클래스 이름공간의 영향을 받는 전역 객체이다.

static 함수

  1. 일반 함수 : internal linkage를 갖는다. 즉, 외부 파일에서 해당 함수를 호출하지 못한다.

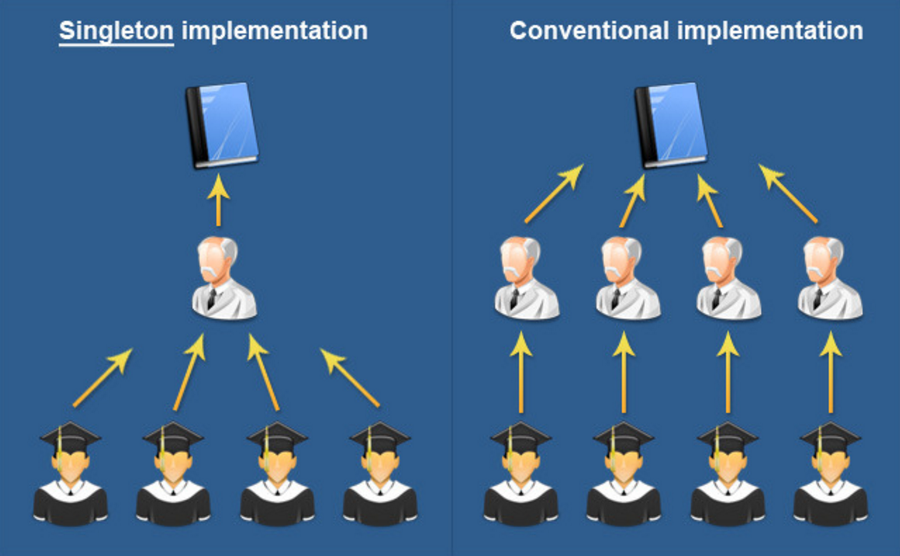
2. 클래스 멤버 함수 : this 포인터를 갖지 않는다. 따라서 객체의 멤버 변수나 멤버 함수를 직접 참조/호출할 수 없다. 단, 같은 클래스의 static 멤버 변수나 static 멤버 함수는 직접 참조/호출이 가능하다.

쓰다 보니 주저리 주저리 글이 너무 길어졌습니다. 부디 이 글을 통해서 static에 대한 개념 이해에 도움이 되셨으면 좋겠습니다.

혹시 내용 중에 잘못된 내용이나 이상한 부분이 있으면 귀찮으시더라도 메일 주시면 감사 드리겠습니다.

글쓴이: 이은조([gimmesilver@hanmail.net](mailto:gimmesilver@hanmail.net))

## 동작과정



위 그림에서처럼 임의의 클래스에 하나의 객체만 만들어서 관리하는 기법. 동작 과정은 그리 복잡하지 않다. 하지만 static이 갖는 의미는 이해할 필요가 있다.

## 구현

class GameMode

{

public:

static GameMode& getInstance()

{

static GameMode gm;

return gm;

}

void intro()

{

std::cout << "fire\n";

}

~GameMode();

private:

GameMode();

};

#include "GameMode.h"

int main()

{

auto gm = GameMode::getInstance();

gm.intro();

return 0;

}

## 세부사항

### 싱글톤의 의의

싱글톤 패턴을 쓰는 이유는 다음과 같다.

오직 1개 객체 + 전역 접근

### 장점

한번도 사용하지 않는다면 아예 생성하지 않는다.

런타임에 초기화된다.

싱글톤을 상속할 수 있다.

간단하다.

### 단점

어쨌든 전역변수이다. 이는 혼란을 가중시킨다.

싱글톤의 2가지 의의로 인해 오히려 단점이 될 수 있다.

게으른 초기화로 성능이 저하될 수 있다.

### 대안

싱글톤은 가능한 사용하지 않는 것이 좋다. 어떻게 대체할 수 있을지 생각해보자.

#### 객체가 스스로 챙기게 하자.

Manager클래스가 있다면, 그가 관리하는 객체가 스스로 할 수 있는 일인지 확인하고 가능하면 객체가 스스로 작동하도록 한다.

#### 1개의 객체만 얻기

유일한 객체이기를 원하지만 전역 접근은 원하지 않는 경우다. 이는 간단한 존재 확인 여부만 넣어주면 해결할 수 있다.

static bool isInstaniated;

#### 인스턴스에 쉽게 접근하는 법

전역 접근과 같이 쉬운 접근을 원한다. 여러 가지 방법이 있다.  
1. 넘겨주기

//...  
int getPlayer(GameMode& gm)

{

std::cout << "player.\n";

return gm.getPlayerCounter();

}

효율적인 방법이지만, 적절하게 GameMode를 넘겨주기가 쉽지 않아 보인다.

2. 상위 클래스로부터 얻기

상속하여 부모 클래스의 멤버 함수에 접근한다.

class Player : public GameMode{

public:

Player()

{

std::cout <<

GameMode::getPlayerCounter() << '\n';

}

//...

};

설계만 적절하다면 굉장히 효율적으로 보인다.

3. 이미 전역인 객체로부터 얻기

이는 여러 시스템들을 각각 전역을 만들기보다 주축이 되는 전역 객체를 한 개 만들고 나머지를 이를 통해 접근할 수 있도록 하는 것이다.

class AudioSys{};

class FileSys{};

class GameMode

{

public:

GameMode();

virtual ~GameMode();

static GameMode& getInstance();

AudioSys\* getAudioSys();

FileSys\* getFileSys();

//...

void intro();

int getPlayerCounter();

private:

static int playerCounter;

AudioSys \*audio;

FileSys \*file;

};

전역의 피해를 최소화하며 적절할 수 있다.

4. 서비스 중개자로부터 얻기  
[16. 서비스 중개자 패턴]: 전역 접근을 가능하게 한다.

8. 이중 버퍼

## 동작과정

## 구현

Actor.h

#pragma once

#include <iostream>

class Actor

{

public:

Actor();

virtual ~Actor();

virtual void update() = 0;

void slap();

bool wasSlapped();

private:

static int currentSlapped;

static int nextSlapped;

bool slapped[2];

public:

static void init();

static void swap();

static int next();

};

Actor.cpp

#include "Actor.h"

int Actor::currentSlapped;

int Actor::nextSlapped;

void Actor::slap()

{

std::cout << "slap!\n";

slapped[next()] = true;

}

bool Actor::wasSlapped()

{

return slapped[currentSlapped];

}

void Actor::init()

{

currentSlapped = 0;

}

void Actor::swap()

{

currentSlapped = next();

}

int Actor::next()

{

return 1 - currentSlapped;

//return (currentSlapped + 1) % 2;

}

## 세부사항

더 빠르고 정확하게 교체하기

### 포인터나 레퍼런스 교체

아주 단순하고 빠르다. pimpl관용구를 적용해도 되겠다.

### 객체 상대적 오프셋을 이용한다.

현재의 값이 1이라고 하면, 다음과 다다음은 현재+1, 현재+2임을 이용하여 모두 복사하지 않는다.

### static 함수로 모든 객체를 한번에 교체한다.

어떤 순간의 교체시기는 모든 객체가 동일하므로 아주 효율적이다.

9. 게임 루프

## 동작과정

## 구현

전반적인 구성은 동작과정과 같다. 하지만 이에 대한 몇 가지 제어로 다양한 구현이 나올 수 있다.

### 최대한 빨리 달리기

while (true)

{

processInput();

update();

render();

}

위에 보이는 대로다. 아무것도 제어하지 않고 미친 듯이 실행한다.

### 한숨 돌리기

const long long PER\_FRAME = 16;

int main()

{

while (true)

{

long long start = clock();

processInput();

update();

render();

std::this\_thread::sleep\_for(

std::chrono::milliseconds(

start + PER\_FRAME - clock()));

}

return 0;

}

FPS=60: 1초에 60번 update를 실행한다.  
일정수의 update를 수행하며, 시간이 남으면 쉬어서 게임진행이 빨라지지 않도록 막는다.

### 한번은 짧게, 한번은 길게

int main()

{

long long lastTime = clock();

while (true)

{

long long currentTime = clock();

long long elapsed = currentTime - lastTime;

processInput();

update(elapsed);

render();

lastTime = currentTime;

}

return 0;

}

update에 프레임당 시간차인 elapsed를 넘긴 것이 인상 깊다. 이는 업데이트할 때, 이 시간만큼 업데이트한다. 즉, 1초에 5만큼의 이동을 한다고 할 때, elapsed가 0.1이면 0.5만큼만 이동시키는 것이다.

이는 네트워크 방식의 게임에서 PC마다 실행 속도가 달라져서 다르게 보이는 문제점이 있다.

### 따라잡기: 고정 시간 간격

const long long PER\_FRAME = 16;

int main()

{

long long lastTime = clock();

long long lag = 0;

while (true)

{

long long currentTime = clock();

long long elapsed = currentTime - lastTime;

lastTime = currentTime;

lag += elapsed;

processInput();

while (lag >= PER\_FRAME)

{

update();

lag -= PER\_FRAME;

}

render();

}

return 0;

}

update부분만 반복 실행한다. lag이란, 한 프레임에 걸리는 시간이다. 이를 PER\_FRAME동안 반복하는 것이다. 좋은 PC에서는 촘촘히 update가 될 뿐, PC간의 게임 진행 속도 차이는 없다.

### 업데이트 사이의 렌더링: 가변 시간 간격

따라잡기 방법에서 렌더링은 한번뿐이 실행한다. 하지만 업데이트 중간에 렌더링이 되면 움직임이 튀어 보인다. 1과 5사이에 렌더링이 되면 3만큼 되어야 하는데 여전히 1만 진행할 것이다. 따라잡기와 코드는 같으며, 렌더링만 인자를 넘겨서 진행한다.

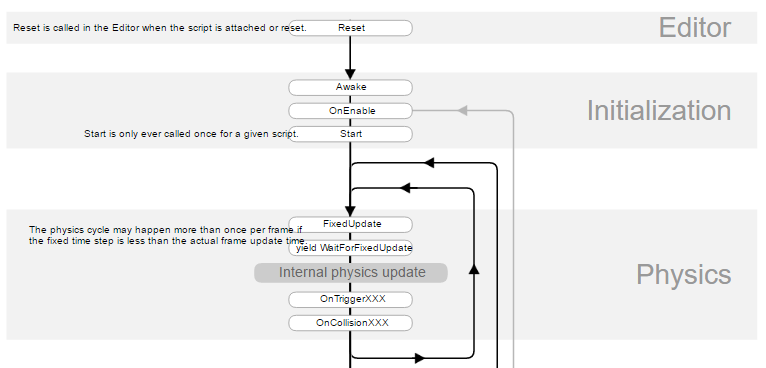
render(lag / PER\_FRAME);

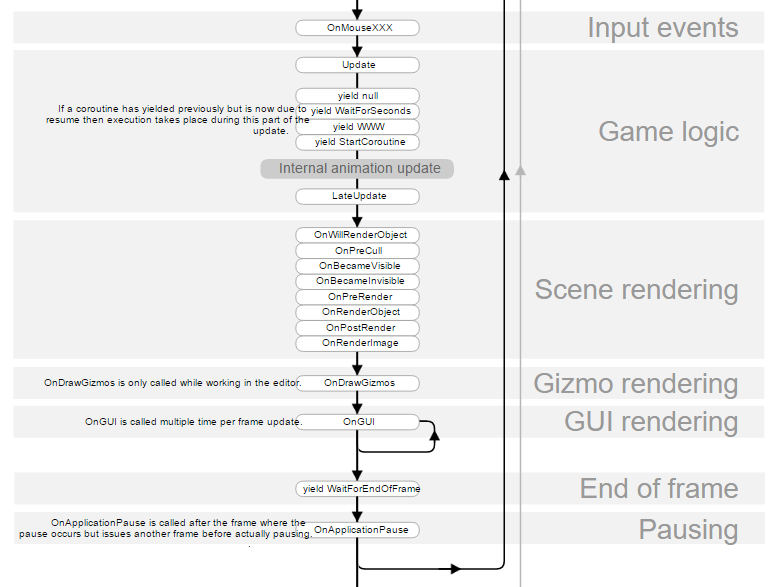
## 세부사항

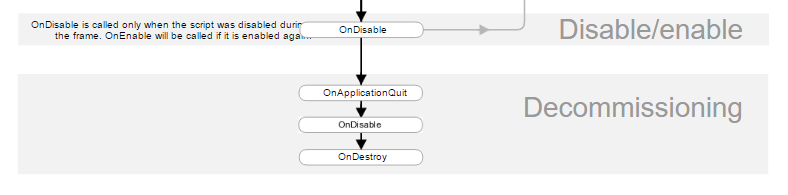
<https://docs.unity3d.com/Manual/ExecutionOrder.html>

### Script Lifecycle Flowchart

The following diagram summarises the ordering and repetition of event functions during a script’s lifetime.







10. 업데이트 메서드

## 사전지식

### 상속: is-a 관계

class Archer : public Unit

{

//Archer이면 Unit이다.

//Unit의 모든 기능을 가져야 한다.

};

### 결합: has-a 관계

class Archer : public Unit

{

public:

void update()

{

}

private:

class Impl

{

//객체 안에 들어있습니다.

//Archer은 Impl을 가지고 있습니다,

};

std::unique\_ptr<Impl> pimpl;

};

### 위임(Delegation)﻿

class Archer : public Unit

{

public:

void update()

{

pimpl->update();

}

private:

class Impl

{

public:

void update()

{

}

};

std::unique\_ptr<Impl> pimpl;

};

위임이란 행동을 전해주는 것이다. Archer은 Impl이라는 인스턴스를 가지고 있다. 결합상태인 것이다. Archer a라는 인스턴스는 Impl pimpl이라는 객체에게 update()를 호출하라고 전해준다. 이것이 위임이다.

### 상속과 결합의 차이

의미상 차이가 있다. **이 의미에서 오는 가장 큰 차이는 결합도이다.** 상속은 기본 클래스의 영역을 침범할 수 있는 강한 결합이며, 결합은 결합된 인스턴스끼리만 공유하는 상대적으로 느슨한 결합이다.

상속에 재정의가 있다면 결합에는 위임이 있는 것이다. 역시 차이는 같다. 결합된 대상의 함수를 호출한다고 하자. 캡슐화의 정도는 어떠한가? **상속된 모든 인스턴스는 어떠한 인스턴스이든 외부에서 자신의 기본 클래스로 침범할 수 있으며, 기본 클래스의 정보를 바꿀 수 있는 반면에, 결합에서는 본인의 내부함수 외에는 결합된 인스턴스로 접근이 불가능하다.**

나와 결합된 대상은 나만 사용할 수 있다. 결국, 더욱 강력한 캡슐화로 해석할 수 있다. **위임의 의미는 그러하다. 결합된 대상에게 행동을 전할 뿐이지, 어떠한 경우로도 외부로부터의 접근은 완벽히 보호된다.**

## 동작과정

아래의 사이클은 한 프레임이다. 모든 객체를 순회함을 볼 수 있다. 그리고 각 객체당 객체 내의 AI, 물리 등 update를 수행한다.

## 구현

### 클래스 상속

int counter;

class Unit

{

public:

Unit() : id(counter++) {}

virtual ~Unit() {}

virtual void update() = 0;

private:

int id;

};

class Builder : public Unit

{

public:

Builder() : workingPower(0), Unit() {}

virtual void update()

{

std::cout << "building...\n";

workingPower++;

}

private:

int workingPower;

};

const long long PER\_FRAME = 1000;

class World

{

public:

void gameLoop()

{

while (true)

{

long long current = clock();

//사용자 입력

for (auto unit : set)

{

unit->update();

}

std::this\_thread::sleep\_for(

std::chrono::milliseconds(PER\_FRAME + current - clock()));

//물리, 렌더링

}

}

void add(Unit\* unit)

{

set.insert(unit);

}

private:

std::unordered\_set<Unit\*> set;

};

### 컴포넌트 클래스

[14. 컴포넌트 패턴] 참고

### 위임 클래스

class Archer : public Unit

{

public:

void update()

{

pimpl->update();

}

private:

class Impl

{

public:

void update()

{

std::cout << "destroy you!\n";

}

};

std::unique\_ptr<Impl> pimpl;

};

## 세부사항

### 휴면 객체 처리

Q. 비활성 하고자 하는 객체들은 update를 할 필요가 없다. 이는 어떻게 처리할 것인가?

A. 살아있는 객체만 따로 모아두면 된다.

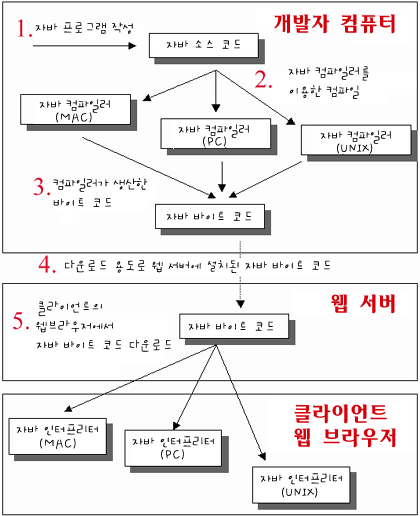
11. 바이트 코드

## 사전지식

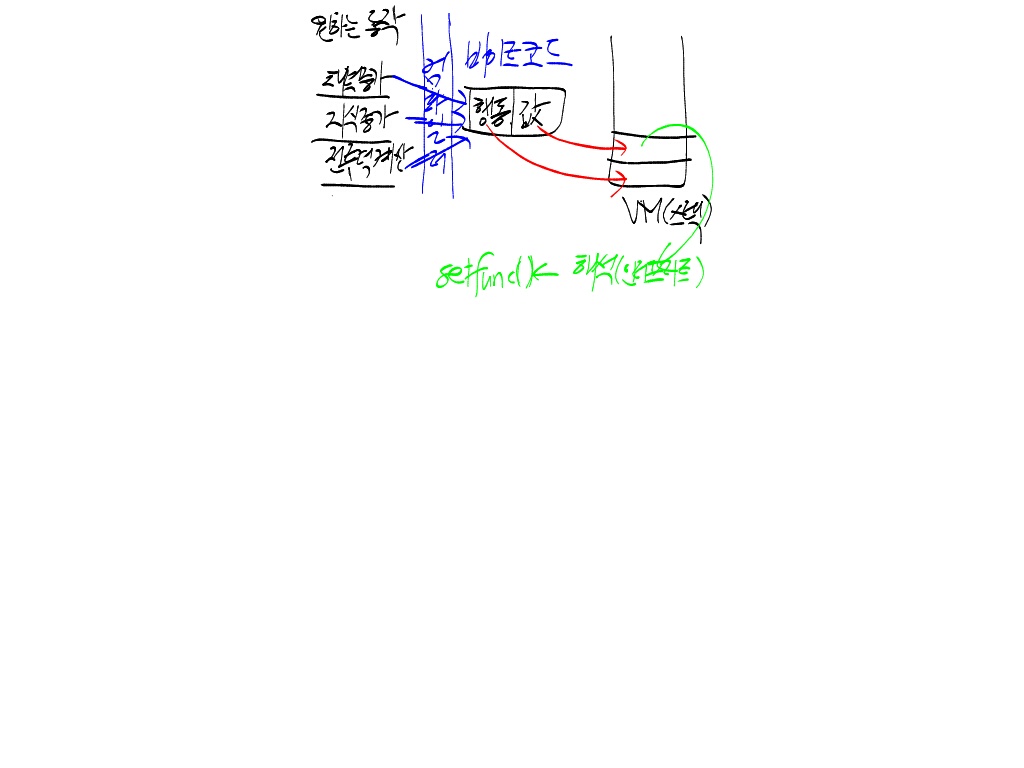
**바이트코드**(Bytecode)는 특정 하드웨어가 아닌 [가상 컴퓨터](https://ko.wikipedia.org/wiki/%EA%B0%80%EC%83%81_%EC%BB%B4%ED%93%A8%ED%84%B0)에서 돌아가는 [실행 프로그램](https://ko.wikipedia.org/wiki/%EC%BB%B4%ED%93%A8%ED%84%B0_%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4)을 위한 [이진](https://ko.wikipedia.org/wiki/%EC%9D%B4%EC%A7%84%EB%B2%95) 표현법이다. 하드웨어가 아닌 소프트웨어에 의해 처리되기 때문에, 보통 [기계어](https://ko.wikipedia.org/wiki/%EA%B8%B0%EA%B3%84%EC%96%B4)보다 더 [추상적](https://ko.wikipedia.org/wiki/%EC%B6%94%EC%83%81%ED%99%94_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99))이다.

### 자바에서의 바이트코드

컴파일한 결과가 어셈블리어가 아닌 바이트코드인 것이다. 이것을 한번 더 해석해서 기계어로 변환한다. 이 해석을 하드웨어가 아닌 JVM이 한다. 어떤 환경에서든 JVM만 있으면 해석할 수 있으므로 이식성이 좋다.



## 동작과정



## 구현

enum Instruction

{

INST\_SET\_HEALTH = 0x00,

INST\_SET\_WISDOM = 0x01,

INST\_SET\_AGILITY = 0x02,

INST\_SET\_SOUND = 0x03,

INST\_SET\_PARTICLE = 0x04,

INST\_LITERAL = 0x05,

INST\_ADD = 0x06,

};

class VM

{

public:

VM() : stackSize(0) { memset(stack, 0, sizeof(stack)); }

void interpret(const std::vector<Instruction>& vinst)

{

for (int i = 0; i < vinst.size(); i += 2)

{

int j = i + 1;

int inst = vinst[i];

switch (inst)

{

case INST\_SET\_HEALTH:

int amount = pop();

int archer = pop();

//setHealth(archer, amount);

break;

case INST\_SET\_WISDOM:

case INST\_SET\_AGILITY:

case INST\_SET\_SOUND:

case INST\_SET\_PARTICLE:

case INST\_LITERAL:

int value = vinst[j];

push(value);

break;

case INST\_ADD:

int b = pop();

int a = pop();

push(a + b);

break;

}

}

}

private:

static const int MAX\_SIZE = 128;

int stackSize;

int stack[MAX\_SIZE];

void push(int value)

{

assert(stackSize < MAX\_SIZE, "There is no space.");

stack[stackSize++] = value;

}

int pop()

{

if (stackSize <= 0)

{

std::cout << "size is 0.\n";

}

else

{

return stack[--stackSize];

}

}

};

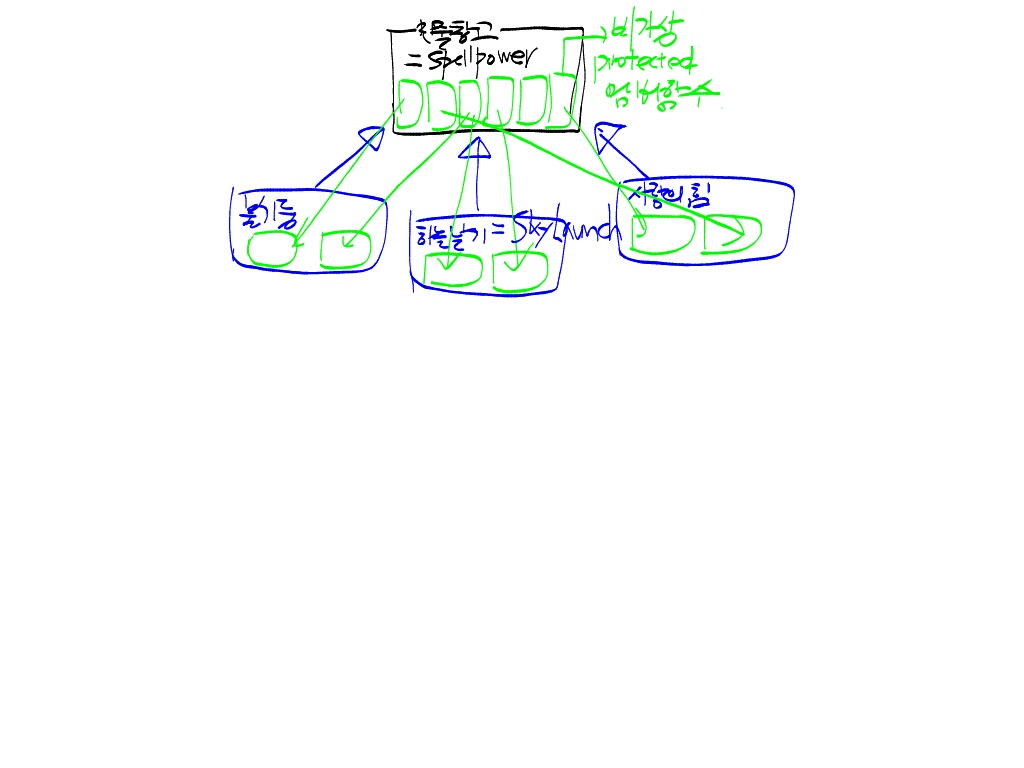
간단하게 구현되어 있다. 전반적인 순서를 설명한다.  
1. VM을 결정한다. 이는 스택 혹은 레지스터 기반이 될 수도 있다.  
2. 명령어 집합(enum Instruction)을 만든다. 명령어들은 유저와 소통하는 외부 행동, VM내부 행동, 흐름 제어, 추상화가 있다.  
3. 값을 어떻게 표현할지 정한다.

## 세부사항

가장 중요한 바이트 코드를 생성하는 방법이다.

12. 하위 클래스 샌드박스

## 동작과정



## 구현

SpellPower.h/.cpp

#pragma once

#include "SoundPlayer.h"

class SoundId;

class ParticleType;

class Hero;

class ParticleSystem;

class SpellPower

{

public:

SpellPower();

virtual ~SpellPower();

//초기화 함수로 외부에서 객체를 가져온다.

//정적 데이터는 정적 함수로 가져온다.

static void init(ParticleSystem\*);

protected:

virtual void activate() = 0;

void move(double x, double y, double z);

// void playSound(SoundId sound, double volume);

void spawnParticles(ParticleType type, int count);

double getHeroX();

double getHeroY();

double getHeroZ();

SoundPlayer& getSoundPlayer();

private:

Hero\* hero;

//다른 객체를 품음으로써 메서드 분리하기

SoundPlayer soundPlayer;

//정적 선언으로 모든 클래스가 공유

static ParticleSystem\* particle;

};

#include "SpellPower.h"

#include "SoundId.h"

#include "ParticleType.h"

#include "Hero.h"

#include <iostream>

void SpellPower::move(double x, double y, double z)

{

std::cout << "move\n";

}

//void SpellPower::playSound(SoundId sound, double volume)

//{

// std::cout << "playSound\n";

//}

void SpellPower::spawnParticles(ParticleType type, int count)

{

std::cout << "spawnParticle\n";

}

double SpellPower::getHeroX()

{

return hero->getX();

}

double SpellPower::getHeroY()

{

return hero->getY();

}

double SpellPower::getHeroZ()

{

return hero->getZ();

}

SoundPlayer& SpellPower::getSoundPlayer()

{

//TODO: insert return statement here

return soundPlayer;

}

ParticleSystem\* SpellPower::particle;

void SpellPower::init(ParticleSystem\* \_particle)

{

particle = \_particle;

}

모든 능력은 이 SpellPower을 상속한다. 각 능력은 activate만 재정의하여 필요한 기능만 골라서 사용하면 된다.   
1. 메서드가 너무 많을 수 있으므로 SoundPlayer의 객체를 품어서 이 객체를 반환하는 방식을 사용할 수 있다.  
2. 모든 클래스가 공용으로 사용하면 정적으로 선언하여 메모리를 아낀다.  
3. 품어야 할 외부 객체는 생성자로 가져올 수 있지만, 개수가 많아지면 인터페이스가 망가질 수 있다. 그러므로 초기화 함수를 따로 마련한다. 이는 파생된 객체의 펙토리 함수에서 호출할 것이다.

SkyLaunch.h/.cpp

#pragma once

#include "SpellPower.h"

class SkyLaunch :

public SpellPower

{

public:

SkyLaunch();

~SkyLaunch();

//초기화와 생성을 묶어줄 캡슐화가 필요하다.

SpellPower\* createSkyLaunch(ParticleSystem\*);

protected:

virtual void activate();

};

#include "SkyLaunch.h"

#include "SoundId.h"

#include "ParticleType.h"

SkyLaunch::SkyLaunch()

{

}

SkyLaunch::~SkyLaunch()

{

}

void SkyLaunch::activate()

{

if (getHeroZ() == 0)

{

getSoundPlayer().playSound(SOUND\_SPROING, 1.0f);

spawnParticles(PARTICLE\_DUST, 10);

move(0, 0, 20);

}

else if (getHeroZ() < 10.0f)

{

getSoundPlayer().playSound(SOUND\_SWOOP, 1.0f);

move(0, 0, 20);

}

else

{

getSoundPlayer().playSound(SOUND\_DIVE, 0.7f);

spawnParticles(PARTICLE\_SPARKLES, 1);

move(0, 0, 20);

}

}

SpellPower\* SkyLaunch::createSkyLaunch(ParticleSystem\* \_particle)

{

SpellPower\* power = new SkyLaunch();

power->init(\_particle);

return power;

}

파생된 객체의 실질적인 행위는 재정의한 activate가 된다. 그리고 펙토리 함수 createSkyLaunch(ParticleSystem\* \_particle)에서는 생성과 init으로 초기화를 묶어서 캡슐화하고 있다.

## 세부사항

위의 구현에서 설명되었지만 다시 한 번 정리한다.

### 어떤 기능을 제공할 것인가?

1. 몇 안 되는 클래스에서 사용하는 메서드라면, 하위 클래스 자체에서 외부 시스템에 직접 접근할 수도 있다.   
2. 다른 시스템의 함수에 접근할 때, 내부 내용을 바꾸지 않는다면 좋지만, 바꾼다면 눈에 잘 들어오도록 상의 클래스의 제공 기능으로 옮겨주는 게 나을 수 있다.  
3. 제공 기능이 단순히 외부 시스템으로 호출을 넘겨주는 일 밖에 하지 않는다면 하위 클래스에서 외부 메서드를 직접 호출하는 것이 나을 수 있다.

### 메서드를 직접 제공할 것인가? 해당 객체를 품을 것인가?

상위 클래스 메서드의 수가 늘어나는 것을 방지하기 위해 이런 기능들을 묶고 있는 객체를 품는 것이 더 좋을 수 있다.

### 필요한 객체는 어떻게 얻는가?

생성자로 얻을 수 있지만, init()함수를 별도로 만들어서 초기화하는 방법이 있다. 이는 객체 생성과 초기화를 하나로 묶어주는 캡슐화된 펙토리 함수를 사용해야 사용자가 빼먹지 않고 init()을 호출할 수 있다.

### 서비스 중개자 이용하기

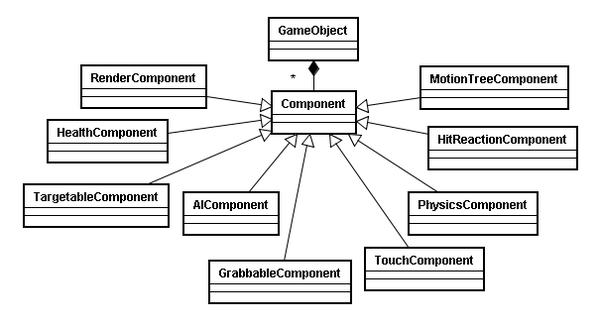
SpellPower::spawnParticles함수에서 상위 클래스에서 필요로 하는 객체를 밖에서 미리 설정해주어야 한다. 즉, 밖에서 초기화시킨 후, 내부에서 설정해야 했다. 하지만 서비스 중개자 패턴을 이용하여 위와 같은 인터페이스로 내부에서 스스로 초기화할 수 있다. – [16. 서비스 중개자 패턴] 참고

13. 타입 객체

이 부분은 값을 변경하지 않고 데이터를 수정한다는 의미를 알 수가 없다. 작성 보류

14. 컴포넌트

## 동작과정



## 구현

#include <iostream>

#include <memory>

World world;

class BladeMaster : public Archer

{

//…

};

class Archer

{

public:

~Archer() { delete input; }

int velocity;

int x, y;

//펙토리 메서드 패턴

static std::unique\_ptr<Archer>

createBladeMaster(InputComponent\* \_input,

PhysicsComponent\* \_physics,

GraphicsComponent\* \_graphics)

{

auto player = std::make\_unique<BladeMaster>();

player->init(\_input, \_physics, \_graphics);

return player;

}

void init(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:

Archer() {}

InputComponent\* input;

PhysicsComponent\* physics;

GraphicsComponent\* graphics;

};

class InputComponent

{

public:

virtual ~InputComponent() {}

virtual void update(Archer& archer) = 0;

};

class PlayerInputComponent : public InputComponent

{

public:

virtual void update(Archer& archer)

{

switch (Controller::getJoystickDirection())

{

case DIR\_LEFT:

archer.velocity -= WALK\_ACCELERATION;

break;

case DIR\_RIGHT:

archer.velocity += WALK\_ACCELERATION;

break;

}

}

private:

static const int WALK\_ACCELERATION = 1;

};

class DemonInputComponent : public InputComponent

{

public:

virtual void update(Archer& archer)

{

std::cout << "random move\n";

}

};

class PhysicsComponent

{

public:

virtual ~PhysicsComponent() {}

virtual void update(Archer& archer, World& \_world) = 0;

};

class ArcherPhysicsComponent : public PhysicsComponent

{

public:

void update(Archer& archer, World& \_world)

{

archer.x += archer.velocity;

\_world.resolveCollision(volume, archer.x, archer.y, archer.velocity);

}

private:

Volume volume;

};

class GraphicsComponent

{

public:

virtual ~GraphicsComponent() {}

virtual void update(Archer& archer, Graphics& \_graphics) = 0;

};

class ArcherGraphicsComponent : public GraphicsComponent

{

public:

void update(Archer& archer, Graphics& \_graphics)

{

Sprite\* sprite = &spriteStand;

if (archer.velocity < 0)

{

sprite = &spriteWalkLeft;

}

else if (archer.velocity > 0)

{

sprite = &spriteWalkRight;

}

\_graphics.draw(\*sprite, archer.x, archer.y);

}

private:

Sprite spriteStand;

Sprite spriteWalkLeft;

Sprite spriteWalkRight;

};

int main()

{

Archer::createArcher(new PlayerInputComponent,

new ArcherPhysicsComponent,

new ArcherGraphicsComponent);

return 0;

}

## 세부사항

### 객체는 컴포넌트를 어떻게 얻는가?

누가 이 컴포넌트를 합쳐줄 것인가?  
1. 객체가 스스로 컴포넌트를 생성 🡪 실수하고 빠뜨릴 일은 없지만 객체를 변경하기가 어렵다.  
2. 외부 코드에서 컴포넌트를 제공하 🡪 객체가 훨씬 유연해지며, 구체 컴포넌트 자료형으로부터 디커플링할 수 있다.

### 컴포넌트끼리 어떻게 통신할 것인가?

다른 컴포넌트간의 통신하는 방법

#### 컨테이너 객체의 상태를 변경하는 방식 1. 서로 디커플링 상태를 유지한다. 2. 컴포넌트들이 공유하는 정보를 컨테이너 객체에 전부 넣어야 한다. 3. 실행 순서에 의존한다.

#### 컴포넌트가 서로 참조하는 방식

다른 컴포넌트를 받아서 인수로 삼는다.

class ArcherGraphicsComponent : public GraphicsComponent

{

public:

ArcherGraphicsComponent(ArcherPhysicsComponent\* \_physics)

: physics(\_physics) {}

private:

ArcherPhysicsComponent\* physics;

Sprite spriteStand;

Sprite spriteWalkLeft;

Sprite spriteWalkRight;

};

#### 메시지를 전달하는 방식

기본 인터페이스인 Component를 정의하여 컨테이너 객체는 이 모든 Component들에게 메시지를 보낸다.

class Component

{

public:

virtual ~Component() {}

virtual void receive(int message) = 0;

};

class ContainerObject

{

public:

void send(int \_message)

{

for (auto c : components)

{

if (c != nullptr)

{

c->receive(\_message);

}

}

}

private:

std::vector<Component\*> components;

};

15. 이벤트 큐

## 사전지식

4. 관찰자 패턴과 연관이 깊다. 이의 비동기 버전이다.

## 동작과정

### 기본 과정

여러 가지 이벤트가 생길 것이다.   
이를 큐 안에 차곡차곡 담는다.   
이벤트가 들어오면 해결해야 한다. 이를 확인하기 위해 무한루프가 있어야 한다. (이벤트 핸들러)  
앞에서부터 이를 해결한다.

여러 상황에서 큐 안에 이벤트를 전해줄 수 있다. 전투 코드, 튜토리얼 코드 등 전투 중에도 튜토리얼에 관련된 메시지를 전해야 한다면 이벤트 큐에서 이를 처리할 수 있다는 것이다.

### 문제해결

요청을 즉시적으로 하나씩 처리하면 같은 소리가 2배가 되는 등 이벤트를 제어할 수가 없다.   
🡪 요청을 모아두어 잃어버리지 않도록 해야 한다.

게임에는 여러 개의 스레드가 작동한다. 그러므로 여러 개의 스레드가 하나의 함수에 접근할 수 있는데, 동기화 과정에서 문제가 될 수 있다.  
🡪 ?

즉시성은 요청을 처리하는 시기를 유연하게 할 수 없다.  
🡪 요청을 받는 부분과 처리하는 부분을 구분해야 한다.

### 주의사항

전역변수

처리하는 동안 상태가 변할 수 있다. 현재 정보와 처리 내용을 함께 담아야 한다.

피드백 루프에 빠질 수 있다. (이해 안 됨)

## 구현

기본배열로 원형버퍼 원리/구현하기

큐에 넣기 전에 요청 취합하기

멀티스레드

## 세부사항

### 디자인 결정

큐에 무엇을 넣을까?

누가 큐를 읽는가?

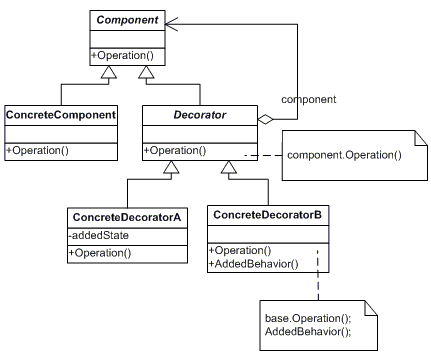
누가 큐에 쓰는가?

객체의 생명주기

16. 서비스 중개자

## 사전지식

**데코레이터 패턴**(Decorator pattern)이란 주어진 상황 및 용도에 따라 어떤 객체에 책임을 덧붙이는 패턴으로, 기능 확장이 필요할 때 서브클래싱 대신 쓸 수 있는 유연한 대안이 될 수 있다.



클래스 전체가 아닌 각각의 객체마다 특정한 기능을 추가 시기고자 하는 경우, 클래스의 수가 폭발적으로 늘어날 가능성이 있고, 새로운 케이스가 발생할 때 마다 새로운 클래스를 만들어야 하는 단점 있다.

이런 상속의 단점을 해결하기 위해 객체의 합성을 이용 하는, **즉, 추가 되고자 하는 특정 기능을 클래스로 구현하고, 그 인터페이스를 user class가 내포 함으로써 동적으로 추가 기능을 변경** 가능토록 하고, 또한 추가 기능 클래스가 다른 추가 기능클래스를 내포하는 재귀적인 형태를 가짐으로써 여러 가지 기능의 추가가 가능토록 할 수 있는 패턴이 Decorator Pattern이다.

/\* Component (interface) \*/

class Widget {

public:

virtual void draw() = 0;

virtual ~Widget() {}

};

/\* ConcreteComponent \*/

class TextField : public Widget {

private:

int width, height;

public:

TextField(int w, int h) {

width = w;

height = h;

}

void draw() {

cout << "TextField: " << width << ", " << height << '\n';

}

};

/\* Decorator (interface) \*/

class Decorator : public Widget {

private:

Widget\* wid; // reference to Widget

public:

Decorator(Widget\* w) {

wid = w;

}

void draw() {

wid->draw();

}

~Decorator() {

delete wid;

}

};

/\* ConcreteDecoratorA \*/

class BorderDecorator : public Decorator {

public:

BorderDecorator(Widget\* w) : Decorator(w) { }

void draw() {

Decorator::draw();

cout << " BorderDecorator" << '\n';

}

};

/\* ConcreteDecoratorB \*/

class ScrollDecorator : public Decorator {

public:

ScrollDecorator(Widget\* w) : Decorator(w) { }

void draw() {

Decorator::draw();

cout << " ScrollDecorator" << '\n';

}

};

int main(void) {

Widget\* aWidget = new BorderDecorator(

new ScrollDecorator(

new TextField(80, 24)));

aWidget->draw();

delete aWidget;

}

Component(interface)인 Widget은 draw()를 실행하는 객체이다. 이를 상속한 객체는 draw()를 실행하지만 기능은 모두 다르다. 결국, Widget을 상속하게 되면 파생 클래스의 draw()가 추가되어 작동한다. 이는 Widget의 draw() 🡪 파생 객체의 draw()를 호출하게 된다.

그러므로 aWidget의 draw()은 BorderDecorator + ScrollDecorator + TextField가 된다.

## 동작과정

[그림]

## 구현

#include <iostream>

#include <memory>

//Audio 서비스

class Audio

{

public:

virtual ~Audio() {}

virtual void playSound(int soundID) = 0;

virtual void stopSound(int soundID) = 0;

virtual void stopAllSounds() = 0;

};

class NullAudio : public Audio

{

public:

virtual void playSound(int soundID) {}

virtual void stopSound(int soundID) {}

virtual void stopAllSounds() {}

};

class ConsoleAudio : public Audio

{

public:

virtual void playSound(int soundID)

{

std::cout << "playSound\n";

}

virtual void stopSound(int soundID)

{

std::cout << "stopSound\n";

}

virtual void stopAllSounds()

{

std::cout << "stopAllSounds\n";

}

};

class LoggedAudio : public Audio

{

public:

LoggedAudio(Audio& \_wrapped) : wrapped(\_wrapped) {}

virtual void playSound(int soundID)

{

wrapped.playSound(soundID);

}

virtual void stopSound(int soundID)

{

wrapped.stopSound(soundID);

}

virtual void stopAllSounds()

{

wrapped.stopAllSounds();

}

private:

void log(const char\* message)

{

std::cout << message + '\n';

}

Audio& wrapped;

};

//중개자

class Locator

{

public:

static void initilize()

{

service = &nullService;

}

//중개 역할

static Audio\* getAudio() { return service; }

//Audio 서비스 등록하기

static void provide(Audio\* \_service)

{

if (\_service == nullptr)

{

service = &nullService;

}

else

{

service = \_service;

}

}

private:

static Audio\* service;

static NullAudio nullService;

};

Audio\* Locator::service;

NullAudio Locator::nullService;

int main()

{

Locator::initilize();

ConsoleAudio\* audio = new ConsoleAudio;

Locator::provide(audio);

LoggedAudio\* service = new LoggedAudio(\*Locator::getAudio());

Locator::provide(service);

return 0;

}

## 세부사항

### 서비스는 어떻게 등록되는가?

외부 코드에서 등록  
1. 빠르고 간단하다.  
2. 서비스 제공자를 어떻게 만들지 제어할 수 있다.  
3. 게임 실행 도중에 서비스를 교체할 수 있다.  
4. 서비스 중개자가 외부 코드에 의존한다는 단점이 있다.

컴파일 시, 바인딩  
1. 빠르다.  
2. 서비스는 항상 사용 가능하다.  
3. 서비스를 변경할 수 없다.

런타임에 설정 값 읽기  
1. 다시 컴파일하지 않고도 서비스를 교체할 수 있다.  
2. 프로그래머가 아니어도 서비스를 바꿀 수 있다.  
3. 등록 과정을 코드에서 완전히 빼냈기 때문에 하나의 코드로 여러 설정을 동시에 지원할 수 있다.  
4. 복잡하다.  
5. 서비스 등록에 시간이 걸린다는 단점이 있다.

### 서비스를 못 찾으면 어떻게 할 것인가?

사용자가 알아서 처리한다: NULL을 반환한다. 사용자가 알아서 적당하게 반응하게 한다.

게임을 멈춘다: 단정문을 사용하여 중개자의 버그라고 말하는 것이다.

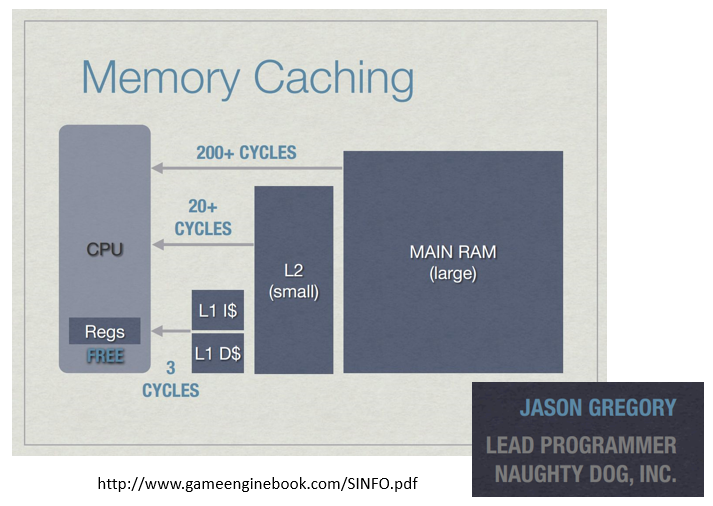
널 서비스를 반환한다: 앞에서 선보인 방법이다. 서비스가 없어도 게임을 진행한다.

### 서비스의 범위를 어떻게 잡을 것인가?

일반적으로는 싱글턴처럼 항상, 전역에 존재하지만 특정 클래스의 접근으로 제한될 수 있다.

17. 데이터 지역성

## 동작과정



위의 그림을 보면 미리 가져다 놓을 시, 이득을 얻을 수 있는 이유를 알 수 있다.

## 구현

### [Data Oriented Design](http://www.dataorienteddesign.com/dodmain/node3.html)(DOD)의 구현

[Data Oriented Design](http://www.dataorienteddesign.com/dodmain/node3.html) (DOD)는 [Object-Oriented Design](https://en.wikipedia.org/wiki/Object-oriented_design)(OOD)와 다른 편에 서 있는 Language Design 개념이다. 말 그대로 객체 지향이 아닌, 데이터 지향적인 프로그래밍 설계 방법이다.

대게 OOD에서는 몬스터 클래스를 만들면 아래와 같이 설계한다.

class Monster

{

Vector3 power;

Vector3 velocity;

Vector3 position;

Quaternion rotation;

void render();

};

vector<Monster> monsterList;

그런데 DOD에서는 아래와 같이 한다.

class MonsterGroup

{

vector<Vector3> power;

vector<Vector3> velocity;

vector<Vector3> position;

vector<Quaternion> rotation;

void render();

};

즉, monster의 객체를 따로 정의하고 생성된 몬스터를 리스트에 집어 넣어서 객체별로 따로 움직임을 주는 것이 아닌, 몬스터의 객체를 없애고 각 Property 데이터들을 처리하는 일을 더 수월하게 구조를 잡는 형태이다. 이렇게 하는 여러가지 이유가 있으나, 특별히 게임 엔진이나 물리 엔진 등에서 Cache Miss를 줄여서 퍼포먼스를 올리기 위해 사용한다.

### 정렬된 데이터

어떤 데이터 그룹을 사용한다고 하자. 여기서 조건에 맞는 일부만 사용한다고 할 때, 모든 데이터를 매번 검사하며 수행하는 것은 비효율적이다. 다음과 같이할 수 있다.

class Particle

{

public:

void update()

{

std::cout << "effect.\n";

}

bool isActive()

{

return activeFlag;

}

private:

bool activeFlag;

};

class ParticleSystem

{

public:

ParticleSystem() : numParticles(0),

particles(std::vector<Particle>(MAX\_PARTICLES))

{

}

void update()

{

//활성화된 파티클만 수행한다.

//모든 파티클을 순회하는데다 추가적인 검사는 비효율적이다.

/\*for (auto& p : particles)

{

if (p.isActive())

{

p.update();

}

}\*/

//활성화된 파티클을 앞쪽에 넣는다.

for (int i = 0; i < numActive; i++)

{

particles[i].update();

}

}

void activateParticle(int index)

{

assert(index >= numActive);

std::swap(particles[numActive], particles[index]);

numActive++;

}

void deactivateParticle(int index)

{

assert(index < numActive);

numActive--;

std::swap(particles[numActive], particles[index]);

}

private:

static const int MAX\_PARTICLES = 10000;

int numParticles;

std::vector<Particle> particles;

int numActive;

};

## 세부사항

### 빈번한 코드와 한산한 코드 나누기

필요 없는 update는 피하는 것이 캐시 적중에 효율적이다. 그래서 자주 쓰는 것과 그렇지 않은 것을 나누어 주는 것이다.

### 다형성은 어떻게 할 것인가?

사용하지 않거나 종류별로 다른 배열에 넣는다.

### 게임 개체는 어떻게 정의할 것인가?