치트키 커맨드를 수행하는 핵심 함수 원형은 이렇다.

BOOL SetGMCommand(const char* commandString);

커맨드 스트링을 함수 인자로 전달해서 수행하는 모양이다. 함수 네이밍에 문제가 있어 보이지만 내용상 문제는 없어 보인다. 함수 내부를 살펴본다.

GM_COMMAND_TYPE CommandType = GetCommandType(commandString);

커맨드 스트링을 분석해서 enum 타입을 얻는 모양이다. 굳이 enum 타입을 써야 했을까? 커맨드가 추가될 때마다 GetCommandType() 함수 내부도 덩달아 수정해야 하지 않는가. 스트링으로도 충분히 구분할 수 있지 않았을까. 어쨌든 ...

실 커맨드 수행코드는 enum 타입으로 분기가 되어 있다. switch-case 문으로. 아래와 같이.

```
switch (commandType)
{
        case GM_COMMAND_USER_KICK:
        ...
}
```

switch-case 문을 활용하는 데 특별한 이견은 없다. 적어도 if-else 문보다는 가독성에서 유리하다. 문제는 유지보수에 있다. 치트키 커맨드가 100개 200개가 넘는다면 한 함수 내에서 모든 코드를 넣어야 할 판이다. 기존 커맨드를 찾기도 어렵고 추가 시점 잡기도 어렵다. 만에 하나 이런 저런 쓸데없는 flag 와 같은 변수들이 덕지덕지 붙어있는 경우엔 분석이 매우 어려워진다. 추가하기도 어렵다.

커맨드 인자들은 어떻게 처리하고 있나...

INT32 Arg = GetCommandTypeArg(CommandType, commandString);

명령어 인자를 얻어오는 함수로 보인다. 그런데 이상하다. 명령어 인자가 정수형 변수 1개 밖에 없다. 좀 더 찾아보니 GetCommandTypeArgList 라는 함수가 있다. 여러 명령어 인자를 추출해주는 함수로 보이는데 비교적 최근에 만들어진 함수이다. 아마도 처음 개발할 때는 여러 개 명령어 인자에 대한 고려가 없었나보다. 또 한가지 이상한 점. 명령어 인자가 정수형 변수라서 다른 변수는 사용할 수 없는 모양이다. 실수형 변수나 스트링이 필요한 경우에는 어떻게 해야하나...

SetGMLobbyCommand 라는 함수가 있다. 로비에서만 명령어를 수행하는 함수로 보인다. 명령어를 수행하는 시기에도 제한이 있어 보인다. 반면 SetGMCommand 함수는 주로 배틀에서 사용된다. 분석 단계에서 이 두 함수의 사용 시기를 정확히 알지 못하면 커맨드를 잘 못 추가하게 되는 상황이 발생할 수도 있을 것이다. Lobby 용 커맨드를 만들어 놓고 SetGMCommand 함수 내부에 구현해 놓으면 말짱 꽝이다.

문제점을 요약해보자.

1. enum 타입을 사용한 switch-case 분기 처리.
2. 커맨드 인자 도출과 사용상의 불편함.
3. 사용시기에 따른 커맨드 종류를 함수로 구분한 점.

설계상의 문제점은 어느정도 이해하지만 가장 좋지 않은 점... 코드가 너무 지저분하다!

하나씩 해결해보자.

enum 타입을 사용한 switch-case 분기 처리는 C 에서 주로 사용하는 스타일이다.

상태별로 구현을 달리하게 되면 switch-case는 그에 맞추어 증가하게 된다. C++ 에서는 상태패턴을 사용한다. 필요한 구현은 파생클래스에 가상함수 오버라이딩을 한다.

인터페이스는 아래와 같다.

```
class ICommand  // 한 번 호출 후 삭제되는 커맨드.
{
public:
    explicit ICommand(ILimitPred* p);
    virtual ~ICommand();

    bool operator()(const i3::vector<i3::string>& params);

private:
    virtual bool _ParseParams(const i3::vector<i3::string>& params) { return true; }
    virtual bool _Execute() { return true; }

    ILimitPred* m_pred;
};
```

파생클래스는 ICommand 클래스를 상속받아 _ParseParams 와 _Execute 함수에서 구현한다.

_ParseParams 함수에서는 커맨드 인자들을 vector 에 스트링으로 담아 파싱하고 실 데이터 타입으로 변환하여 저장한다. 파생 클래스 객체 내에서 수행되기 때문에 코드가 자동으로 분리된다. 인자 개수도 제한이 없으며 파싱만 실수 없이 제대로 하면 문제 없다.

예를 들어 원점으로 캐릭터를 이동하라는 커맨드가 "/warp 0.0, 0.0, 0.0" 라면 _ParseParams 에 "0.0", "0.0", "0.0" 세 스트링이 vector 로 전달될 것이고 파싱하여 실수형 변수로 0.0f 세 개를 파생 클래스 객체에 저장할 것이다. 후에 _Execute 함수에서는 이를 활용하여 캐릭터를 원점으로 이동 시킬 것이다.

아래 실 구현은 좀 더 복잡하지만 원리는 같다.

```
// 워프 : A캐릭터(또는 그 이상의 캐릭터)를 B지점으로 이동
//
// ex) "/wp slot[0] nick[test01]" : 0번 캐릭터를 'test01' 캐릭터 위치로 이동
//     "/wp slot[0] pos[x,y,z]"   : 0번 캐릭터를 (x,y,z) 위치로 이동
//     "/wp slot[0] set[bomb-a]"  : 0번 캐릭터를 파일에 저장된 bomb-a 위치으로 이동
//     "/wp nick[test02] nick[test01]" : 'test02' 캐릭터를 'test01' 캐릭터 위치로 이동
//     "/wp nick[test02] pos[x,y,z]"   : 'test02' 캐릭터를 (x,y,z) 위치로 이동
//     "/wp nick[test02] set[bomb-a]"  : 'test02' 캐릭터를 파일에 저장된 bomb-a 위치으로 이동
//     "/wp slot[0,1,2] set[bomb-a]"  : 1, 2, 3 캐릭터를 파일에 저장된 bomb-a 위치으로 이동
//-------------------------------------------------------------------------------------------
bool Warp::_ParseParams(const i3::vector<i3::string>& params)
{
    if (params.size() < 2) return false;

    WarpHelpers::Parser_SearchMovementPlayer psmp;
    i3::string param_players(params[0]);
    if (psmp.operator()(param_players, m_players) == false) return false;

    WarpHelpers::Parser_SearchMovementDestination psmd;
    i3::string param_destination(params[1]);
    if (psmd.operator()(param_destination, m_destination) == false) return false;

    if (!i3Vector::isValid(&m_destination)) return false;

    return true;
}

bool Warp::_Execute()
{
    QA_COMMAND_TELEPORT_DATA info;

    for (size_t i=0; i<m_players.size(); i++)
    {
        info._i32SlotIdx = m_players[i];

        info._ar32Pos[0] = m_destination.x;
        info._ar32Pos[1] = m_destination.y;
        info._ar32Pos[2] = m_destination.z;

        GameEventSender::i()->SetEvent(EVENT_QA_COMMAND_TELEPORT_SOMEONE, &info);
    }

    return true;
}
```

두 가지 문제점이 동시에 해결되었다. 파생클래스를 활용하여 적절히 분기 처리를 할 수 있게 되었고 명령어 인자 도출도 명확하다.

남은 한 가지... 커맨드 사용 시기에 대한 제한 조건은 어떻게 처리해야 할까? 간단하다. 파생클래

스 객체에 조건을 부여하면 된다. ICommand 생성 시 ILimitPred 를 생성하여 넘겨준다. ILimitPred 는 커맨드 수행 시 인자 파싱 전에 실행되어 커맨드를 수행할지 말지를 결정한다.

```
bool ICommand::operator()(const i3::vector<i3::string>& params)
{
    if ((*m_pred)() == false) return false;

    return _ParseParams(params) ? _Execute() : false;
}
```

ILimitPred 인터페이스는 다음과 같다.

```
class ILimitPred
{
public:
    virtual ~ILimitPred() {}

    virtual bool operator()() = 0;
};
```

파생클래스에서는 operator() 함수를 오버라이딩 하여 true false 만 넘겨주면 된다. 이런 식이다.

```
//------------------------------------------
// 로비 전용
//------------------------------------------
bool LPred_LobbyOnly::operator()()
{
    UIMainFrame* stage = g_pFramework->GetUIMainframe();
    if (stage)
    {
        if (stage->GetCurrentPhaseType() == UIPHASE_LOBBY) return true;
    }

    return false;
}
```

좀 더 복잡한 조건 상황은 어떻게 처리해야 할까? 예를 들어 커맨드가 로비용이고 GM 계정에 한해서만 유효하다면? LPred_LobbyOnlyAndGMAccount 와 같은 파생클래스를 만들어야할까? 아니다. 그렇게 하면 안된다. 경우의 수에 따라 파생클래스가 폭발적으로 늘어나게 된다. 이런 경우 클래스들을 조합하는 방향이 옳다. 데코레이터 패턴이라고도 한다. 이런 식으로 ...

```
class ILimitPredDecorator : public ILimitPred
{
public:
    ILimitPredDecorator(ILimitPred* p);
    virtual ~ILimitPredDecorator();

    virtual bool operator()();

private:
    ILimitPred* m_pred;
};

class LPred_GMOnly : public ILimitPredDecorator
{
public:
    explicit LPred_GMOnly(ILimitPred* p) : ILimitPredDecorator(p) {}

    virtual bool operator()();
};
```

최종적으로 아래와 같이 커맨드를 생성한다.

```
 ICommand* Warp::Creator()
 {
     return new Warp(new LPred_PermittedAccountOnly(new LPred_IngameOnly));
 }
```

Creator 함수는 생성 함수인데 지연 생성을 위해 만들었다. 팩토리에 생성함수를 등록해서 필요할 때 호출하는 방식이다(사용하지도 않을 것을 미리 생성할 필요는 없지 않은가). QA 테스트 용도인 관계로 배포 대상이 아니기 때문에 실시간으로 메모리 할당하는 방식 사용했다.

```
//-------------------------------------------
// 커맨드 팩토리
//-------------------------------------------
class CommandFactory
{
public:
    typedef std::tr1::function<ICommand* ()> CreatorPtr;

    CommandFactory()
    {
        _Regist("/WP",              Warp::Creator);                     // 서버 디버깅 필요.
        _Regist("/WPSAVE",          Warp_SaveLocation::Creator);
        _Regist("/WPDEL",           Warp_DelLocation::Creator);
        _Regist("/WARPALL",         WarpAll::Creator);                  // 서버 디버깅 필요.
        _Regist("/WPSHOW",          Warp_ShowLocations::Creator);
        _Regist("/WPHIDE",          Warp_HideLocations::Creator);
        _Regist("/SPAWNOBJ",        RespawnObj::Creator);
        _Regist("/CAMERA3PFREE",    Toggle3pCamera::Creator);
        _Regist("/FLY",             ToggleFlyCamera::Creator);
```

프로젝트에서 C++11 을 본격적으로 사용하기 전이라서 TR1 을 사용했다.

이제 사용자는 아래 두 함수만 사용해서 커맨드를 실행할 수 있다.

```cpp
//-------------------------------------------
// 커맨드 생성 인터페이스
//-------------------------------------------
i3::shared_ptr<ICommand> CreateCommand(const i3::string& keyword)
{
    CommandFactory::CreatorPtr f = g_cmdFactory.FindCreator(keyword);
    if (f)
        return i3::shared_ptr<ICommand>(f());

    return i3::shared_ptr<ICommand>(new Null(new LPred_Anywhere));
}

i3::shared_ptr<ICommand> CreateCommand(const char* keyword)
{
    const i3::string strKeyword(keyword);
    return CreateCommand(strKeyword);
}

bool ParseCommand(const char* cheatkey, i3::string& outKeyword, i3::vector<i3::string>& outParams)
{
    i3::string text(cheatkey);
    i3::to_upper(text);
    i3::lrtrim(text);

    i3::vector<i3::string> tokens;
    Tokenize(text, tokens,  " ", "'\"[]<>{}:;?()");

    if (tokens.empty()) return false;

    outKeyword = tokens[0];

    for (size_t i=1; i<tokens.size(); i++)
        outParams.push_back(tokens[i]);

    return true;
}
```

이런 식으로

```cpp
rstParse = Cheatkey::ParseCommand(mbcsStr.c_str(), keyword, params);

if(!rstParse)
{
    ctx->SetResult(EI3UICSRST_FAIL_EXEC_PARSE_ERR);
    return false;
}

i3::shared_ptr<Cheatkey::ICommand> cmd = Cheatkey::CreateCommand(keyword.c_str());

if( cmd != nullptr)
    rstExec = (*cmd)(params);
```