C++ 게임 서버

1. 소개

이 프로젝트는 서버의 성능과 안정성, 유지보수에 중점을 두고 설계된 C++ 게임 서버입니다. Winsock2를 사용한 소켓 프로그래밍을 구현하였고, DB는 postgreSQL을, 해당 DB의 공식 C++ 라이브러리인 libpqxx를 사용하였습니다.

주요 기능으로는 다음과 같습니다.

1) 회원가입 및 로그인
2) 실시간 채팅
3) 인벤토리 관리
4) 경매장

서버 구조는 던전앤파이터, 메이플스토리, 월드 오브 워크래프트를 참고하였습니다.

2. 개발 환경 및 사용 도구

- **C++ 표준:** C++ 17
- **운영체제(OS):** Windows 10 64bit
- **개발환경(IDE):** Clion 2023.3
- **파일 인코딩:** UTF-8
- **Cmake 버전:** 3.26.4
- **데이터베이스(DB):** postgreSQL 15.5
- **DB 라이브러리:** libpqxx 7.9.0

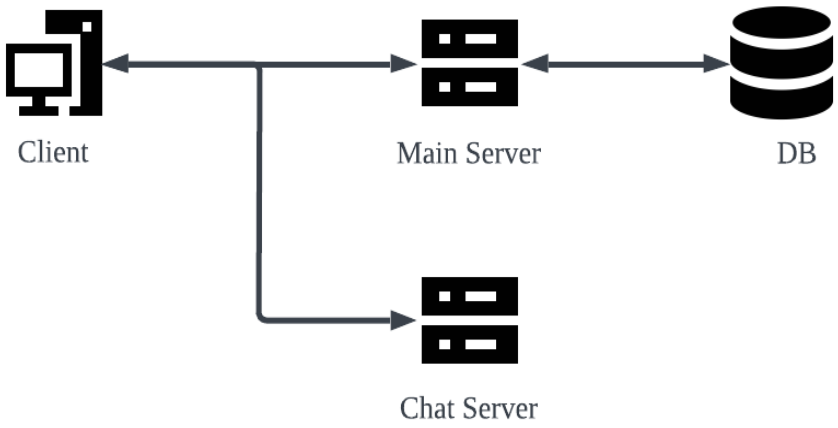
3. 빌드 방법

1) git submodule init / git submodule update를 통해 libpqxx를 설치합니다.
2) Database와 ServerCore를 먼저 빌드 후 나머지 모듈들을 빌드합니다.
   - 각 모듈은 Database와 ServerCore의 정적 라이브러리에 의존하므로 이를 먼저 빌드해야 합니다.
   - libpqxx를 minGW로 빌드할 경우, 스레드 관련 연산자 오버로딩에서 컴파일 에러가 발생합니다. 이 경우 에러가 발생하는 부분을 주석 처리하면 정상적으로 빌드됩니다.
   (issue, solution)
3) postgreSQL 설치 후 GameServer/sql_usersetting.sql, GameServer/sql_schema.sql을 순서대로 실행해야 합니다. 연결 정보는 GameServer/Constants/Constant.h에 하드코딩되어 있습니다.
   (dbname=databasetermproject, user=kosh7707, password=root)
4) 실행 순서는 GameServer -> ChatServer -> GameClient 입니다.

--프로젝트 구조 예시--

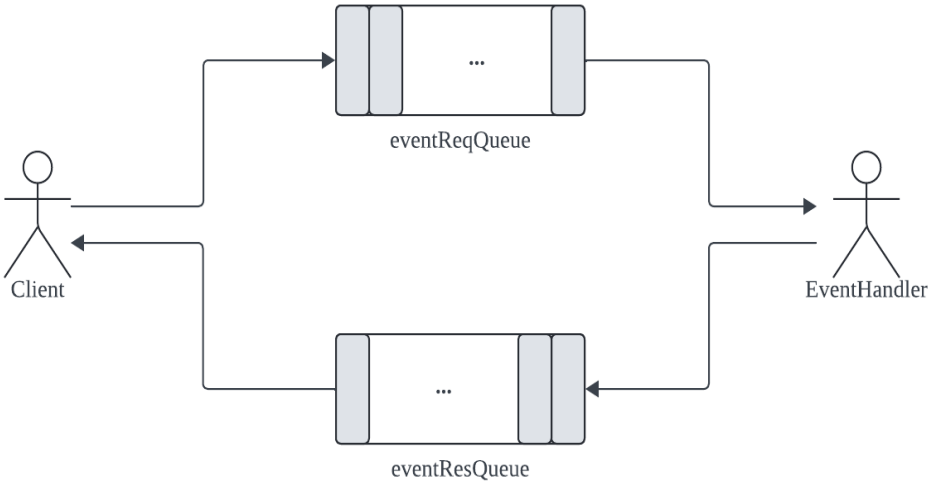
```
Project
├─ChatServer
├─Database
│   ├─include
│   ├─libpqxx
│   └─src
├─dist
├─GameClient
│   └─Constants
├─GameServer
│   ├─Constants
│   ├─DAO
│   ├─Model
│   └─Service
├─lib
│   └─include
│       └─libpqxx
└─ServerCore
    ├─include
    └─src
```

4. 전체 구조



전체 구조는 전통적인 서버 - 클라이언트 구조를 따르며, 각 클라이언트는 메인 서버와 채팅 서버에 연결됩니다.

기본 통신 프로토콜은 EventCode와 EventQueue를 활용하여 비동기 이벤트 기반으로 설계되었습니다.



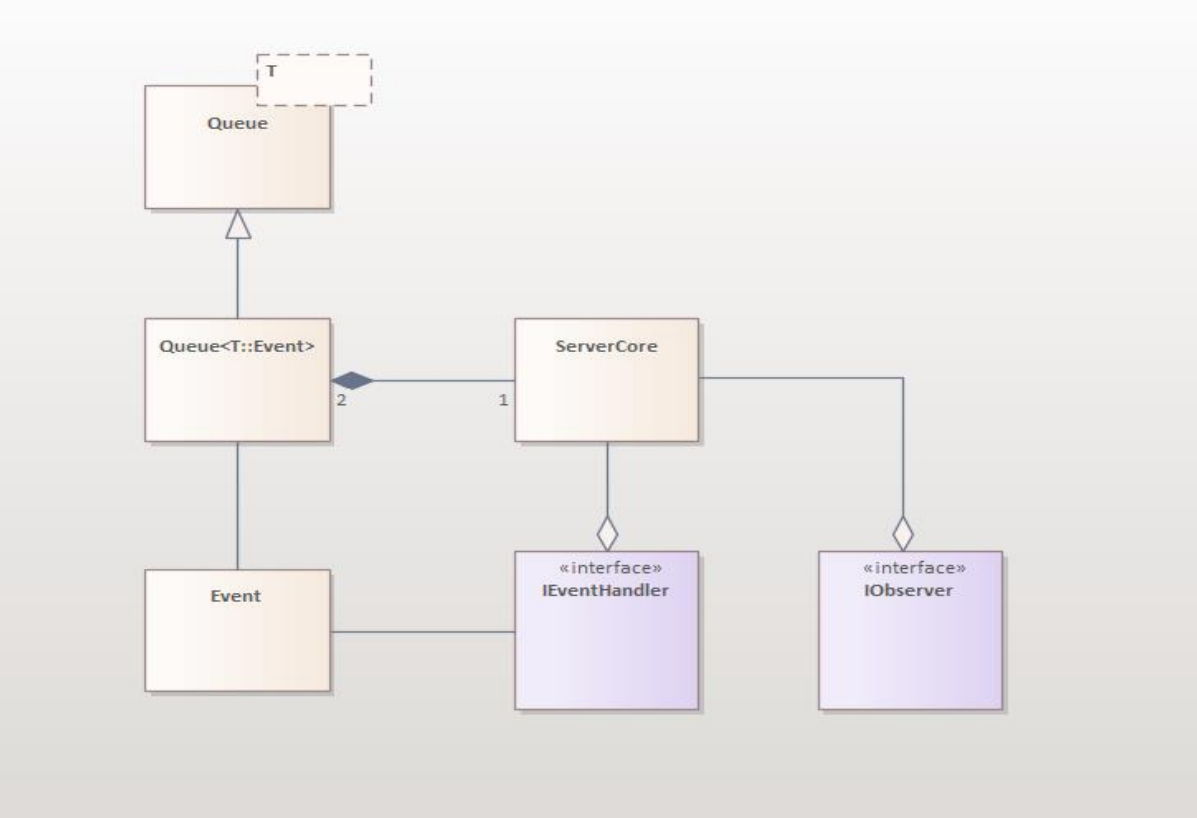
이벤트 처리 흐름은 다음과 같습니다.

1) **이벤트 수신**: 클라이언트에서 이벤트가 수신되면, 해당 이벤트는 eventReqQueue에 추가되어 처리를 대기합니다.
2) **이벤트 처리**: EventHandler가 eventReqQueue에서 이벤트를 꺼내 처리하고, 처리 결과를 eventResQueue에 넣어 순차적으로 클라이언트에 전송합니다.

위 구조를 통해 **이벤트 기반 비동기 처리**를 구현하였으며, EventHandler를 인터페이스로 설계하여 코드의 재사용성과 이식성을 높였습니다. DB는 메인 서버와 연동하여 클라이언트의 계정 정보, 인벤토리를 저장합니다.

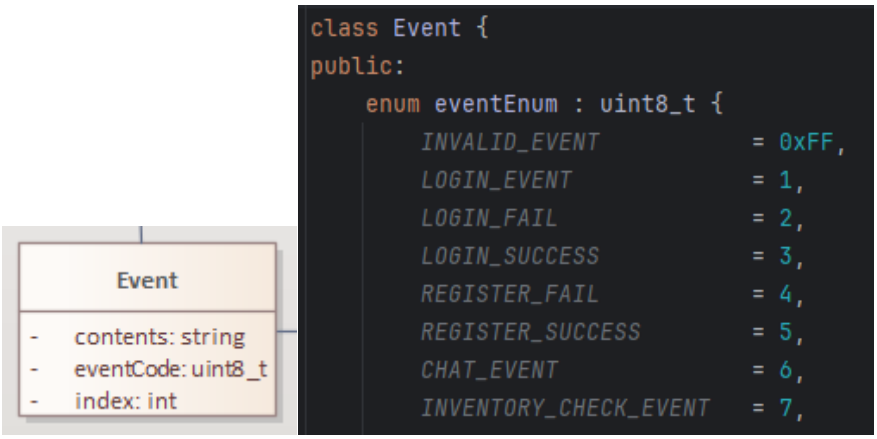
5. 클래스 다이어그램 및 세부 구현

- ServerCore



ServerCore는 클라이언트, 서버의 코어 라이브러리 입니다. Send, Recv, EventHandling 의 3개의 스레드를 유지하며, WSAWaitForMultipleEvents 함수를 사용해 비동기로 이벤트를 기다립니다.

Event Class는 다음과 같습니다.




```
class Event {
public:
    enum eventEnum : uint8_t {
        INVALID_EVENT           = 0xFF,
        LOGIN_EVENT             = 1,
        LOGIN_FAIL              = 2,
        LOGIN_SUCCESS           = 3,
        REGISTER_FAIL           = 4,
        REGISTER_SUCCESS        = 5,
        CHAT_EVENT              = 6,
        INVENTORY_CHECK_EVENT   = 7,
```

이는 index(연결된 소켓의 위치), eventCode, contents로 이루어지며, 전역에서 사용하는 eventEnum을 가집니다.

IObserver는 연결된 소켓 정보가 변함을 감지하기 위한 인터페이스입니다. 코드는 다음과 같습니다.

```
class IObserver {
public:
    virtual void update(const int clientsCount, const std::unique_ptr<Socket[]>& connectedSockets) = 0;
    virtual ~IObserver() {}
};
```

ServerCore가 publish를 호출 할 시, 연결된 observer가 update되는 구조입니다.
이벤트를 send, recv할 땐 다음과 같은 구조를 따릅니다.

- 헤더: eventcode (1 byte) 와 length (4 bytes)로 구성된 5 bytes
- 콘텐츠: 메시지 내용

ServerCore의 send 파트입니다.

```
bool ServerCore::send(std::unique_ptr<Event> event) {
    int         index       = event->getIndex();
    uint8_t     eventCode   = event->getEventCode();
    std::string contents    = event->getContents();

    uint32_t length = htonl( hostlong: static_cast<uint32_t>(contents.length()));

    std::string msg;
    msg.push_back( c: static_cast<char>(eventCode));
    msg.append( s: reinterpret_cast<const char*>(&length),  n: sizeof(length)).append( str: contents);

    const char* buf = msg.c_str();
    int totalLength = static_cast<int>(msg.length());
    int bytesSent = 0;

    while (bytesSent < totalLength) {
        int tempSent = ::send( s: connectedSockets[index].getSc(),  buf: buf + bytesSent,  len: totalLength - bytesSent,  flags: 0);
        if (tempSent <= 0) {
            std::cerr << "send error, errno: " << WSAGetLastError() << std::endl;
            return false;
        }
        bytesSent += tempSent;
    }

    return true;
}
```

send 스레드는 eventResQueue에 있는 이벤트를 하나씩 꺼내 처리합니다. 헤더와 콘텐츠를 포함합니다.

ServerCore의 recvThread 파트입니다.

```
[[noreturn]] unsigned int WINAPI ServerCore::runRecvWorkerThread(void* params) {
    ServerCore* serverCore = static_cast<ServerCore*>(params);

    WSANETWORKEVENTS ev;
    WSAEVENT handleArray[64];

    while (true) {
        handleArray[0] = serverCore->sc.getEv();
        for (int i=0; i<serverCore->clientsCount; i++)
            handleArray[i+1] = serverCore->connectedSockets[i].getEv();

        int index = WSAWaitForMultipleEvents( cEvents: serverCore->clientsCount+1, lphEvents: handleArray, fWaitAll: false, dwTime
        if ((index != WSA_WAIT_FAILED) and (index != WSA_WAIT_TIMEOUT)) {
            if (index == 0) {
                WSAEnumNetworkEvents( s: serverCore->sc.getSc(), hEventObject: serverCore->sc.getEv(), lpNetworkEvents: &ev);
                if (ev.lNetworkEvents == FD_ACCEPT) serverCore->accept();
            }
            else {
                index = index - 1;
                Socket& sc = serverCore->connectedSockets[index];
                WSAEnumNetworkEvents( s: sc.getSc(), hEventObject: sc.getEv(), lpNetworkEvents: &ev);
                if (ev.lNetworkEvents == FD_READ) serverCore->recv(index);
                else if (ev.lNetworkEvents == FD_CLOSE) serverCore->close(index);
            }
        }
    }
}
```

서버의 recv 스레드로, 네트워크 이벤트를 감지합니다. 새 클라이언트 연결, 데이터 수신, 연결 종료와 같은 이벤트를 처리합니다. WSAWaitForMultipleEvents를 사용하여 비동기로 여러 소켓 이벤트를 관리합니다.

ServerCore의 recv 파트입니다.

```
bool ServerCore::recv(const int index) {
    char header[5];
    memset( Dst: header,  Val: 0,  Size: sizeof(header));

    int bytesReceived = 0;
    while (bytesReceived < sizeof(header)) {
        int tempReceived = ::recv( s: connectedSockets[index].getSc(),  buf: header + bytesReceived,  len: sizeof(header) - bytesRe
        if (tempReceived <= 0) {
            std::cerr << "recv header error, errno: " << WSAGetLastError() << std::endl;
            return false;
        }
        bytesReceived += tempReceived;
    }

    uint8_t eventCode = static_cast<uint8_t>(header[0]);

    uint32_t length;
    memcpy( Dst: &length,  Src: &header[1],  Size: sizeof(length));
    length = ntohl( netlong: length);

    if (length > BUF_SIZE) {
        std::cerr << "recv error, message length exceeds BUF_SIZE" << std::endl;
        return false;
    }

    char buf[BUF_SIZE];
    bytesReceived = 0;
    while (bytesReceived < static_cast<int>(length)) {
        int tempReceived = ::recv( s: connectedSockets[index].getSc(),  buf: buf + bytesReceived,  len: static_cast<int>(length) -
        if (tempReceived <= 0) {
            std::cerr << "recv contents error, errno: " << WSAGetLastError() << std::endl;
            return false;
        }
        bytesReceived += tempReceived;
    }

    std::string msg( s: buf,  n: bytesReceived);
    auto pEvent : unique_ptr<Event> = std::make_unique<Event>(index, eventCode,  contents: msg);
    eventReqQueue.push( item: std::move(pEvent));

    return true;
}
```

먼저 헤더인 5 bytes를 읽고, eventCode와 읽어야 할 길이(length)를 확인합니다. 이후 나머지 콘텐츠를 읽고 이벤트를 적절히 가공하여, eventReqQueue에 push합니다. 이후 이벤트는 EventHandler의 처리를 기다립니다.

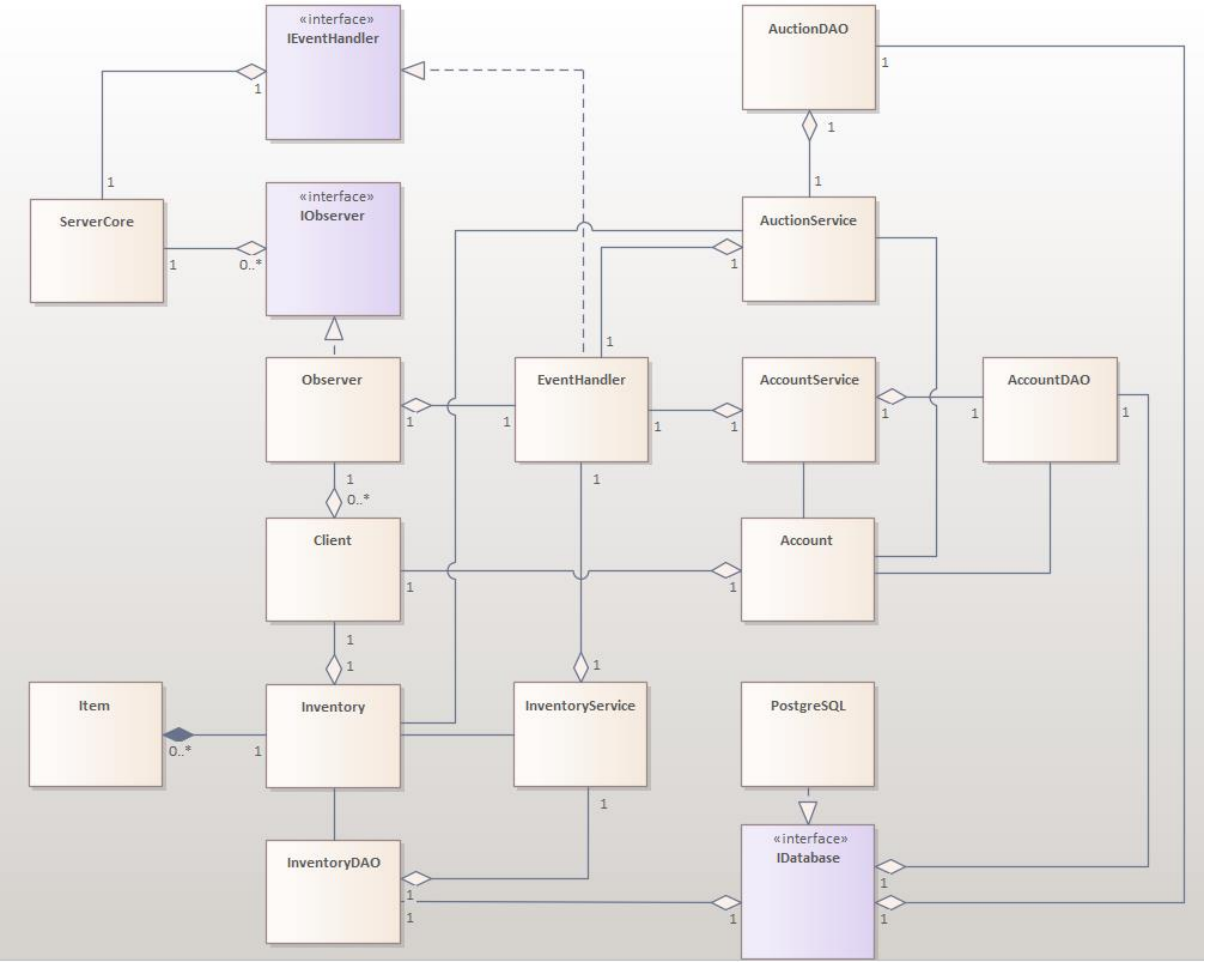
eventResQueue와 eventReqQueue에서 사용하는 Queue Class입니다.

```
template <class T>
class Queue {
public:
    bool push(std::unique_ptr<T> item) {
        std::unique_lock<std::mutex> ul( & mutex);

        qu.push(std::move(item));
        cv.notify_one();
        return true;
    }
    bool pop(std::unique_ptr<T>& item) {
        std::unique_lock<std::mutex> ul( & mutex);

        cv.wait(ul, [this]() { return !qu.empty(); });
        item = std::move(qu.front());
        qu.pop();
        return true;
    }
    bool empty() {
        return qu.empty();
    }
private:
    std::queue<std::unique_ptr<T>>  qu;
    std::mutex                      mutex;
    std::condition_variable         cv;
};
```

기존 queue는 thread-safe를 보장하지 않기에 wrapper class를 작성하였습니다.

push와 pop시에 lock을 걸어 thread-safe를 보장하였으며, condition_variable을 활용해 polling 대신 sleep - wake up으로 자원을 좀 더 절약할 수 있게 하였습니다.

- GameServer



EventHandler가 각각의 서비스 계층을 사용해 이벤트를 처리합니다. 회원가입 및 로그인, 인벤토리 관리, 유저 계정 관리 등 모든 서비스 로직이 EventHandler를 통해 처리됩니다.

GameServer의 Observer의 update 파트입니다.

```
void Observer::update(const int clientsCount, const std::unique_ptr<Socket[]>& connectedSockets) {
    std::lock_guard<std::mutex> lock( & mutex);
    for (int i=0; i<clientsCount; i++) {
        int socket_id = connectedSockets[i].getSocketId();
        indexToSocketId.emplace(i, socket_id);
        if (socketIdToClient.find( x: socket_id) == socketIdToClient.end()) socketIdToClient[socket_id] = std::make_unique<C
        else socketIdToClient[socket_id]->setIndex(i);
    }
}
```

Observer는 ServerCore의 연결된 소켓 정보가 변함을 감지하는 클래스입니다. Index 기반의 소켓 배열을 확장하기 위해 추가적으로 unordered_map을 사용해 기록하였습니다

GameServer의 EventHandler의 handling 파트의 일부 입니다.

```
vectorEventPtr ret;
switch (eventCode) {
    case Event::LOGIN_EVENT:
        ret = handleLogin(index, contents);
        break;
    case Event::INVENTORY_CHECK_EVENT:
        ret = handleInventoryCheck(index);
        break;
    case Event::GET_TEST_ITEM_EVENT:
        ret = handleGetTestItem(index, contents);
        break;
    case Event::BREAK_ITEM_EVENT:
        ret = handleBreakItem(index, contents);
        break;
    case Event::USER_INFO_EVENT:
        ret = handleUserInfo(index);
        break;
    case Event::SELL_ITEM_EVENT:
        ret = handleSellItem(index, contents);
        break;
    case Event::OPEN_AUCTION_EVENT:
        ret = handleOpenAuction(index);
        break;
    case Event::BID_EVENT:
        ret = handleBid(index, contents);
        break;
    case Event::BUYNOW_EVENT:
        ret = handleBuyNow(index, contents);
        break;
    default: {
        std::cerr << "Invalid Event" << std::endl;
        ret.emplace_back(std::make_unique<Event>(index, eventCode: Event::INVALID_EVENT, contents: ""));
        break;
    }
}
return ret;
```

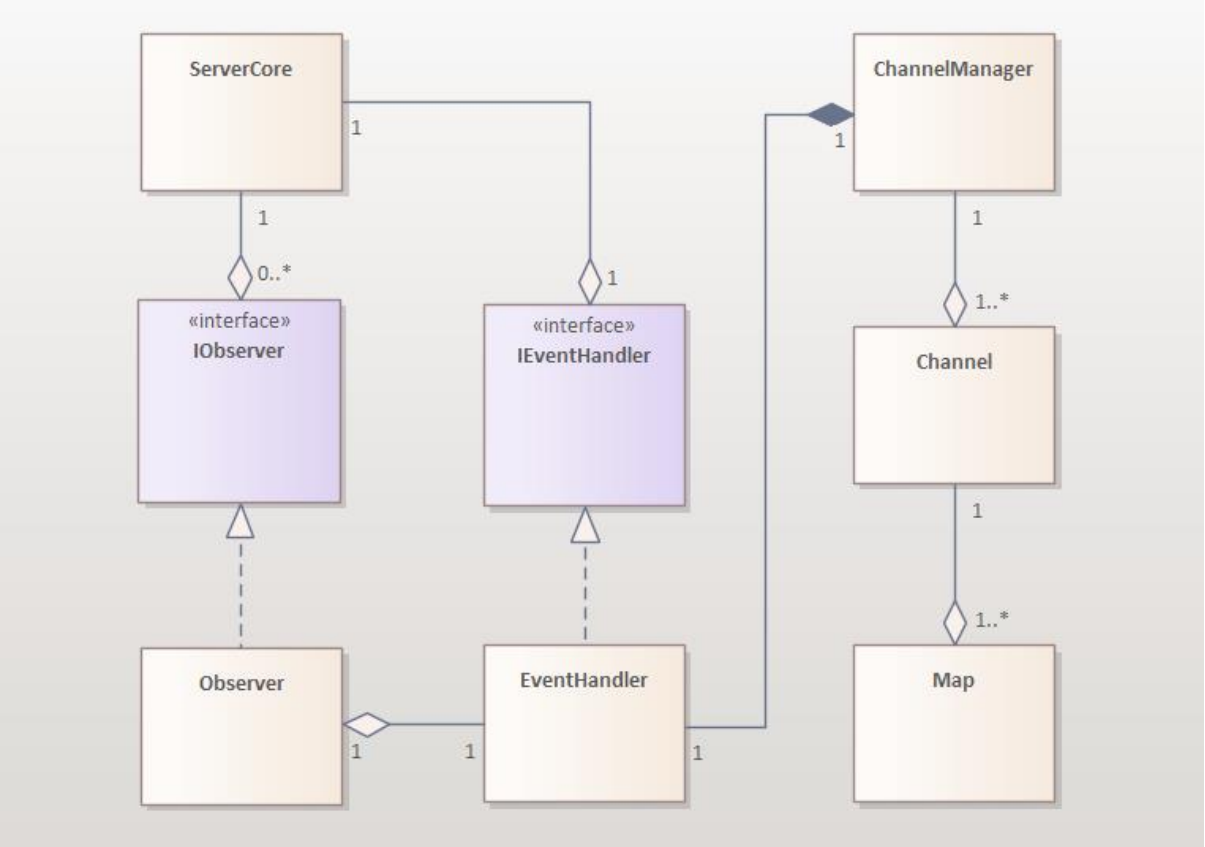
GameServer에 수신된 이벤트를 처리하는 핵심 로직입니다. eventCode에 따라 적절한 핸들러 함수를 호출하여 로그인, 인벤토리 확인, 경매장 기능 등의 이벤트를 처리하고, 결과 이벤트를 eventResQueue에 Push 합니다.

경매장에서 중요한 파트는 경매 만료된 물품에 대한 처리입니다. 여기서는 thread를 하나 더 추가하여 polling하는 방식으로 해결하였습니다. 아래는 해당 코드입니다.

```
// TODO: 경매장 서버 구현 시 삭제 예정
[[noreturn]] unsigned int WINAPI EventHandler::runAuctionWorkerThread(void* params) {
    EventHandler* eventHandler = static_cast<EventHandler*>(params);
    while (true) {
        auto outdatedItemResult :unique_ptr(...) = eventHandler->auctionService->outdatedItemCheck();
        for (const auto& outdatedItem :OutdatedItemResult const & : *outdatedItemResult) {
            int item_id = outdatedItem.item_id;
            int quantity = outdatedItem.quantity;
            auto sellerClient :shared_ptr(Client) = eventHandler->observer->accountIdToClient[outdatedItem.seller_id];
            if (!outdatedItem.success) {
                sellerClient->getInventory()->insertItem(item_id, quantity);
                eventHandler->eventReqQueue.push( item: std::make_unique<Event>( index: sellerClient->getIndex(),  eventCode: Event::EXPIR
            }
            else {
                auto bidderClient :shared_ptr(Client) = eventHandler->observer->accountIdToClient[outdatedItem.current_bidder_id];
                int current_price = outdatedItem.current_price;
                sellerClient->getAccount()->addBalance( b: current_price);
                bidderClient->getInventory()->insertItem(item_id, quantity);
                eventHandler->eventReqQueue.push( item: std::make_unique<Event>( index: sellerClient->getIndex(),  eventCode: Event::ITEM_
                eventHandler->eventReqQueue.push( item: std::make_unique<Event>( index: bidderClient->getIndex(),  eventCode: Event::WIN_A
            }
        }
        std::this_thread::sleep_for( rtime: std::chrono::seconds( rep: 5));
    }
}
```

해당 코드에선 5초마다 경매 만료된 물품을 검색하여 처리합니다. 경매가 만료되었을 경우, 판매자에게 아이템을 반환하거나, 판매자와 구매자에게 알림 이벤트를 주는 등의 서비스 로직을 포함합니다.

- ChatServer



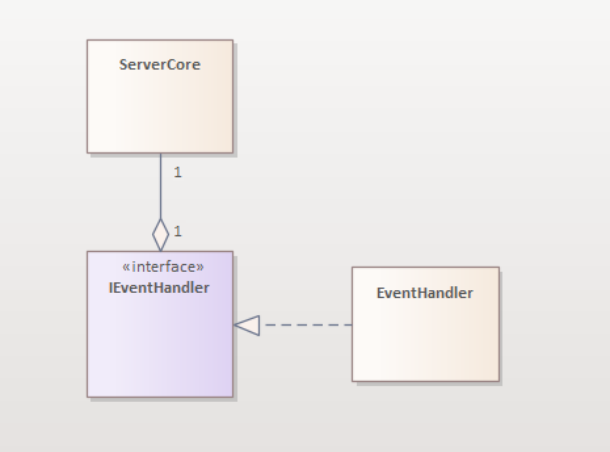
채팅 서버는 클라이언트 간의 실시간 채팅을 지원합니다. 실제 게임에서와 마찬가지로 채널, 맵에 따라 메시지를 받을 수 있는 대상이 정해져야 하기에, Pub/Sub 구조를 활용하였습니다. 클라이언트는 특정 채널이나 맵을 구독하고, 해당 채널이나 맵으로 메시지를 발행하거나 받을 수 있습니다.

ChatServer의 ChannelManager의 코드입니다.

```
class ChannelManager {
public:
    ChannelManager() {}
    bool join(SOCKET client, const std::string& channel_name, const std::string& map_name) {
        return channels[channel_name].join(client, map_name);
    }
    bool leave(SOCKET client, const std::string& channel_name, const std::string& map_name) {
        return channels[channel_name].leave(client, map_name);
    }
    std::vector<SOCKET> BroadCast() {
        std::vector<SOCKET> ret;
        for (auto channel :pair(...) : channels) {
            for (auto client :unsigned long long : channel.second.channelBroadCast())
                ret.emplace_back( & client);
        }
        return ret;
    }
    std::vector<SOCKET> channelBroadCast(const std::string& channel_name) {
        return channels[channel_name].channelBroadCast();
    }
    std::vector<SOCKET> mapBroadCast(const std::string& channel_name, const std::string& map_name) {
        return channels[channel_name].mapBroadCast(map_name);
    }
private:
    std::map<std::string, Channel> channels;
};
```

채널 및 맵의 관리를 담당합니다. 클라이언트의 Pub/Sub를 처리하며 채팅 모드에 따른 메시지 전송 대상자를 결정합니다.

- GameClient



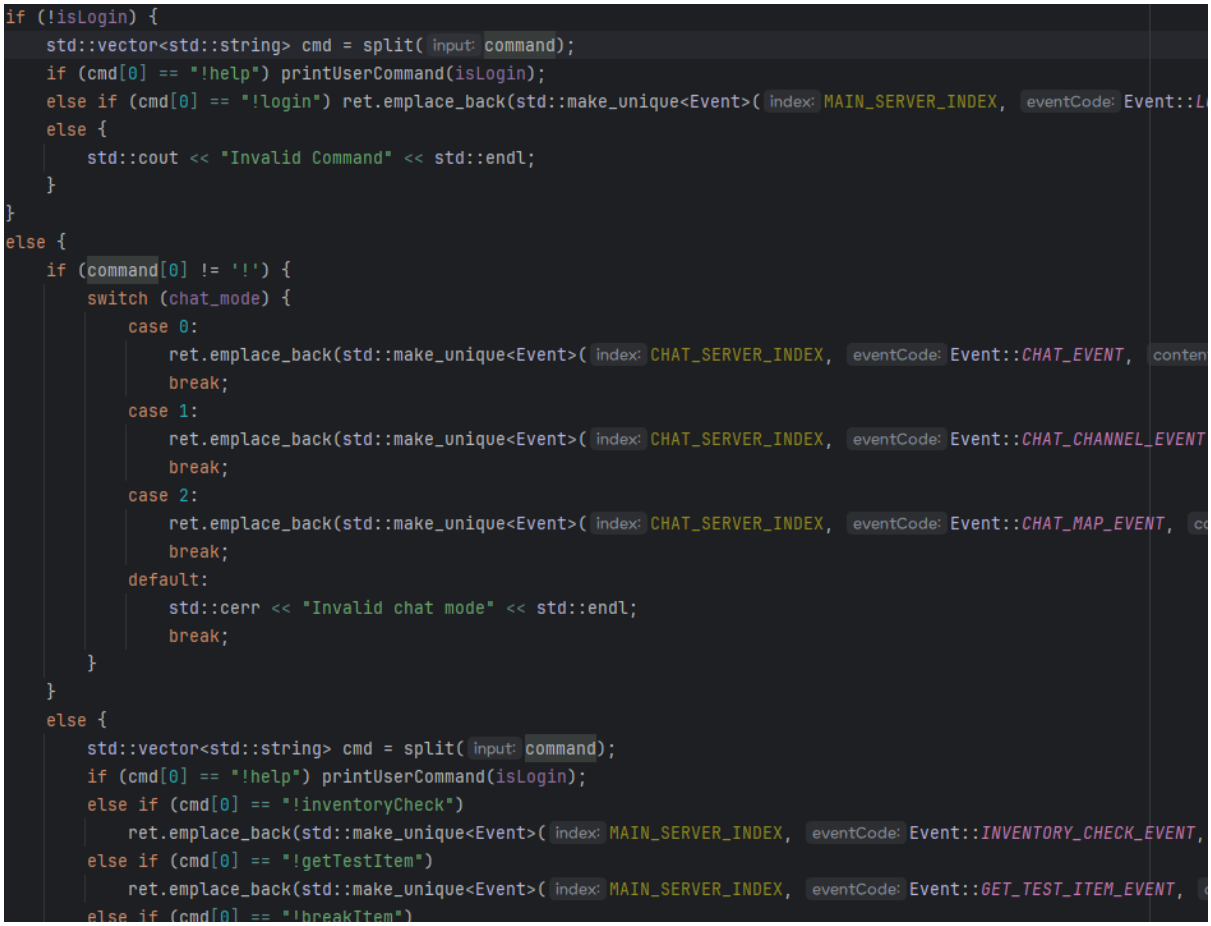
앞서 설명한 서버 구조와 유사한 구조를 따릅니다. EventHandler가 클라이언트의 모든 이벤트를 처리하고, 서버에 이벤트를 전송합니다.

GameClient의 EventHandler의 handling 파트의 일부입니다.

```
auto ret = std::vector<std::unique_ptr<Event>>();
switch (eventCode) {
    case Event::LOGIN_SUCCESS:
        std::cout << "Successfully logged in." << std::endl;
        isLogin = true;
        user_id = contents;
        channel_name = "channel_01";
        map_name = "map_01";
        ret.emplace_back(std::make_unique<Event>( index: CHAT_SERVER_INDEX,  eventCode: Event::SUB_EVENT,  contents: user_id + ","
        break;
    case Event::LOGIN_FAIL:
        std::cout << "Incorrect username or password." << std::endl;
        isLogin = false;
        break;
    case Event::REGISTER_SUCCESS:
        std::cout << "Successfully registered." << std::endl;
        isLogin = true;
        user_id = contents;
        channel_name = "channel_01";
        map_name = "map_01";
        ret.emplace_back(std::make_unique<Event>( index: CHAT_SERVER_INDEX,  eventCode: Event::SUB_EVENT,  contents: user_id + ","
        break;
    case Event::REGISTER_FAIL:
        std::cout << "Registration failed." << std::endl;
        isLogin = false;
        break;
    case Event::CHAT_EVENT:
        std::cout << contents << std::endl;
        break;
```

서버로부터 수신한 이벤트를 처리하고, 필요한 경우에만 후속 이벤트를 생성하여 eventResQueue에 Push합니다. eventCode에 따라 로그인 상태, 채널 및 맵 등을 관리합니다.

GameClient의 EventHandler의 userInputHandling 파트의 일부입니다.



```
if (!isLogin) {
    std::vector<std::string> cmd = split( input: command);
    if (cmd[0] == "!help") printUserCommand(isLogin);
    else if (cmd[0] == "!login") ret.emplace_back(std::make_unique<Event>( index: MAIN_SERVER_INDEX,  eventCode: Event::L
    else {
        std::cout << "Invalid Command" << std::endl;
    }
}
else {
    if (command[0] != '!') {
        switch (chat_mode) {
            case 0:
                ret.emplace_back(std::make_unique<Event>( index: CHAT_SERVER_INDEX,  eventCode: Event::CHAT_EVENT,  conten
                break;
            case 1:
                ret.emplace_back(std::make_unique<Event>( index: CHAT_SERVER_INDEX,  eventCode: Event::CHAT_CHANNEL_EVENT
                break;
            case 2:
                ret.emplace_back(std::make_unique<Event>( index: CHAT_SERVER_INDEX,  eventCode: Event::CHAT_MAP_EVENT,  co
                break;
            default:
                std::cerr << "Invalid chat mode" << std::endl;
                break;
        }
    }
    else {
        std::vector<std::string> cmd = split( input: command);
        if (cmd[0] == "!help") printUserCommand(isLogin);
        else if (cmd[0] == "!inventoryCheck")
            ret.emplace_back(std::make_unique<Event>( index: MAIN_SERVER_INDEX,  eventCode: Event::INVENTORY_CHECK_EVENT,
        else if (cmd[0] == "!getTestItem")
            ret.emplace_back(std::make_unique<Event>( index: MAIN_SERVER_INDEX,  eventCode: Event::GET_TEST_ITEM_EVENT,
        else if (cmd[0] == "!breakItem")
```

클라이언트에서 입력된 명령어를 적절히 파싱하여 이벤트를 생성하고, 이를 서버에 전송합니다.

6. 유저 기능 및 실행 화면

1. 회원가입 및 로그인 기능

시나리오 1: 회원가입 / 로그인

1. 유저는 게임 클라이언트에서 !login [id] [pw] 명령어를 입력하여 회원가입 / 로그인을 요청합니다.
    - 입력 예시: !login kosh7707 12345
2. 서버는 accounts table을 확인하여, 아이디가 존재할 시 로그인을, 존재하지 않을 시 회원가입을 진행합니다.
3. 회원가입 / 로그인 성공 시 클라이언트는 로그인 상태로 전환하며 기본 채널(channel_01)과 기본 맵(map_01)에 자동 참여합니다.

-로그인 예시-

```
connected to chat server
successfully connected to 127.0.0.1:7707
!login kosh11 11
index: 1, eventCode: 3, contents: kosh11
Successfully logged in.
index: 0, eventCode: 15, contents: kosh11
kosh11 is joined.
```

-회원가입 예시-

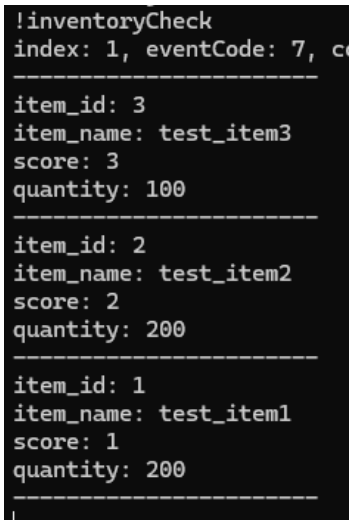
```
connected to chat server
successfully connected to 127.0.0.1:7707
!login kosh33 33
index: 1, eventCode: 5, contents: kosh33
Successfully registered.
index: 0, eventCode: 15, contents: kosh33
kosh33 is joined.
```

2. 인벤토리 기능

시나리오 1: 인벤토리 확인

1. 유저는 !inventoryCheck 명령어를 입력하여 현재 인벤토리에 있는 아이템을 확인합니다.
2. 서버는 유저의 인벤토리 정보를 클라이언트에 전송합니다.
3. 클라이언트는 각 아이템의 ID, 이름, 점수, 수량 등을 출력합니다.

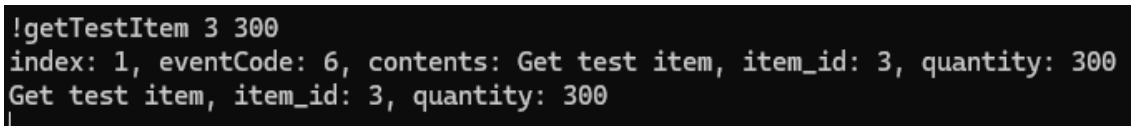
-인벤토리 확인 예시-



시나리오 2: 테스트 아이템 받기

1. 유저는 !getTestItem [item_id] [quantity] 명령어를 입력하여 아이템을 인벤토리에 추가합니다.
    - 입력 예시: !getTestItem 1 15
2. 서버는 요청을 처리하여 해당 아이템을 인벤토리에 추가합니다.

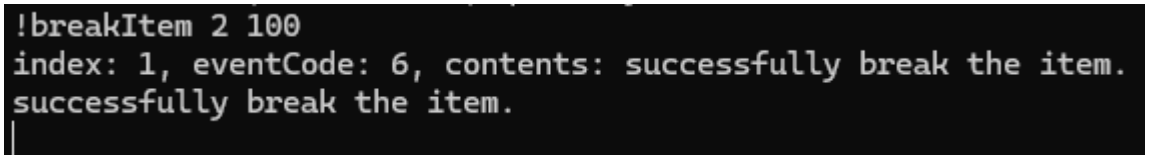
-테스트 아이템 받기 예시-



시나리오 3: 아이템 분해

1. 유저는 !breakItem [item_id] [quantity] 명령어를 입력하여 아이템을 분해합니다.
    - 입력 예시: !breakItem 1 15
2. 서버는 요청을 처리하여 해당 아이템을 인벤토리에서 제거하며, 해당 아이템의 score만큼 accounts의 score에 가산합니다.

-아이템 분해 예시-



시나리오 4: 유저 정보 확인

1. 유저는 !userInfo 명령어를 입력하여 자신의 유저 정보를 확인합니다.
2. 서버는 유저의 ID, 점수, 잔액 등의 정보를 클라이언트에 전송합니다.

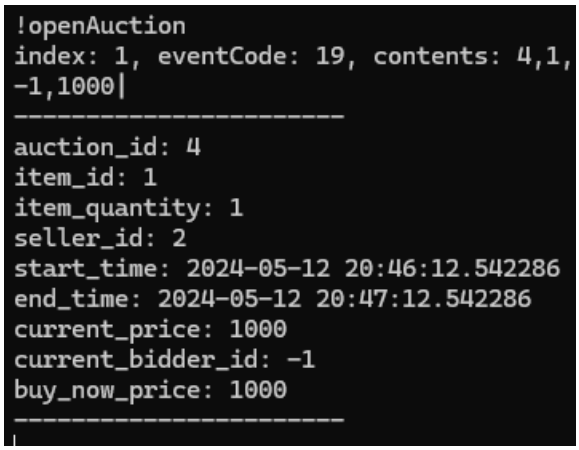
-유저 정보 확인 예시-

3. 경매장

시나리오 1: 경매장 열기

1. 유저는 !openAuction 명령어를 입력하여 경매장에서 현재 진행 중인 경매 목록을 확인합니다.
2. 서버는 경매 목록을 클라이언트에 전송합니다.
3. 클라이언트는 각 경매의 ID, 아이템, 수량, 판매자, 시작 시간, 종료 시간, 현재 가격 등을 출력합니다.

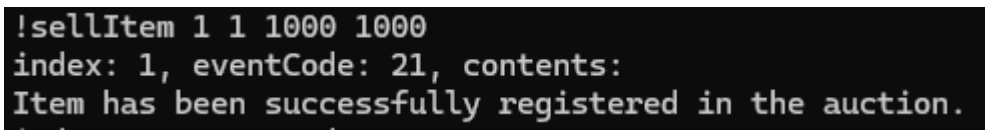
-경매장 열기 예시-



시나리오 2: 아이템 판매

1. 유저는 !sellItem [item_id] [quantity] [buyNowPrice] [startingBidPrice] 명령어를 입력하여 아이템을 경매에 등록합니다.
    - 입력 예시: !sellItem 1 5 500 200
2. 서버는 요청을 처리하고 결과를 클라이언트에 전송합니다.

-아이템 판매 예시-



시나리오 3: 즉시 구매

1. 유저는 !buyNow [auction_id] 명령어를 입력하여 해당 경매의 아이템을 즉시 구매합니다.
    - 입력 예시: !buyNow 1
2. 서버는 요청을 처리하고 결과를 클라이언트에 전송합니다.
3. 해당 과정에서 이전 입찰자가 있었을 경우, 이전 입찰자에게 유찰 메시지를 보냅니다.

-즉시 구매 예시-

```
auction_id: 5
item_id: 1
item_quantity: 10
seller_id: 2
start_time: 2024-05-12 20:48:52.340566
end_time: 2024-05-12 20:49:52.340566
current_price: 1000
current_bidder_id: -1
buy_now_price: 1000
----------------------
!buyNow 5
index: 1, eventCode: 27, contents:
Item has been successfully bought in the auction.
```

시나리오 4: 경매 입찰

1. 유저는 !bid [auction_id] [price] 명령어를 입력하여 경매에 입찰합니다.
    - 입력 예시: !bid 1 200
2. 서버는 요청을 처리하고 결과를 클라이언트에 전송합니다.
3. 해당 과정에서 이전 입찰자가 있었을 경우, 이전 입찰자에게 유찰 메시지를 보냅니다.

-경매 입찰 예시-

```
----------------------
auction_id: 6
item_id: 1
item_quantity: 100
seller_id: 2
start_time: 2024-05-12 20:50:41.035272
end_time: 2024-05-12 20:51:41.035272
current_price: 20
current_bidder_id: -1
buy_now_price: 3000
----------------------
```

auction_id: 6을 입찰할 경우

```
!bid 6 300
index: 1, eventCode: 24, contents:
Bid has been successfully placed in the a
!openAuction
index: 1, eventCode: 19, contents: 6,1,10
,1,3000|
----------------------
auction_id: 6
item_id: 1
item_quantity: 100
seller_id: 2
start_time: 2024-05-12 20:50:41.035272
end_time: 2024-05-12 20:51:54.203105
current_price: 300
current_bidder_id: 1
buy_now_price: 3000
----------------------
```

4. 채팅

시나리오 1: 채널 구독 및 메시지 전송

1. 유저는 !moveChannel [channel_name] 명령어를 입력하여 원하는 채널에 참여합니다.
    - 입력 예시: !moveChannel channel_02
2. 채널 변경 후, 유저는 !changeChatMode 1 명령어를 사용하여 채팅 모드를 채널 모드로 설정합니다.
3. 유저는 단순한 메시지 입력으로 현재 채널에 메시지를 전송합니다.
    - 입력 예시: Hello, everyone!

시나리오 2: 맵 구독 및 메시지 전송

1. 유저는 !moveMap [map_name] 명령어를 입력하여 특정 맵에 참여합니다.
    - 입력 예시: !moveMap map_02
2. 맵 변경 후, 유저는 !changeChatMode 2 명령어를 사용하여 채팅 모드를 맵 모드로 설정합니다.
3. 유저는 단순한 메시지 입력으로 현재 맵에 메시지를 전송합니다.
    - 입력 예시: This map is beautiful!

시나리오 3: 채팅 모드 변경

1. 유저는 !changeChatMode 0/1/2 명령어를 사용하여 채팅 모드를 변경합니다.
    - 0: 전체(All)
    - 1: 채널(Channel)
    - 2: 맵(Map)

5. 전체 명령어 요약

```
------------------------
회원가입 및 로그인
0. !login [id] [pw]
------------------------
인벤토리
01. !inventoryCheck
02. !getTestItem [item_id] [quantity]
03. !breakItem [item_id] [quantity]
04. !userInfo
------------------------
경매장
08. !openAuction
09. !sellItem [item_id] [quantity] [buyNowPrice] [startingBidPrice]
10. !buyNow [auction_id]
11. !bid [auction_id] [price]
------------------------
채팅
05. !moveChannel [channel_name]
06. !moveMap [map_name]
07. !changeChatMode 0/1/2
        --> 0: to All
        --> 1: to Channel
        --> 2: to Map
------------------------
00. !exit
------------------------
```