- Is the intent of the code clear?
    - How does moveCharacter know which character to move without taking in any character or npcIDs?
- Does it have a clear, focused responsibility?
    - I can't really see any of the commands helper classes, so I can't tell for complete certain, but it looks like some commands helper is only responsible for one command (look and swap for example), while other commands helper is responsible for a group of commands (combat, item, communication)? I don't think it necessarily violates single responsibility principle persay, but wouldn't it be more consistent if you made all commands helper responsible for one command each, like you do with look and swap, or made them all responsible for a group of related commands?
- Is it designed to be hard to misuse? Could confusion cause errors?
    - In room controller, all the ids are of type ints, and rooms are also mapped so assumingly some int id each. However, because int is a very vague descriptor, someone might pass in some value when calling room functions that might be an int, but might not necessarily be an id. I think the code could benefit from using something like strong typing or at least aliases such that it's clear to everyone that they are passing in an ID value and not some other int.
- Specifically, does the interface hide design details to ease its use?
    - You use a class called JSONparser in room controller's constructor to parse for rooms. However, it feels like you still need to do some more logic in room controller itself before rooms can be generated. Wouldn't there be more abstraction if you could just pass the path to the JSON folder off to the json parser, and all the error checking and iterating through the files were done in json parsing and just an area or vector of areas was returned? "Parsing" seems to me to suggest that all reading through files and directories and error checking would be done there.

- The Commander's commandMap appears to be repeating a similar pattern of code for different command enums or helpers. Consider factoring out this pattern as a method or function that can be reused to provide a more readable abstraction. I was not sure what was the purpose of the commandMap itself, but the implementation of it is difficult to read and looks as if it is trying to be 'clever' in some way.

- Consider separating the responsibilities of command buffering and creating command objects from strings of the Commander class. These two responsibilities seem too different to be placed into a single class and it may be more manageable to have a buffer handling class to manage the queued commands of each user.

- A large portion of the presented code can benefit from added whitespace or newlines. By doing so, you can have more readable code and the option of using grouping to emphasize certain sets of statements are related to one another.

- For the CommandQueue struct, you may be better off using the networking::Connection struct as the key value for the map rather than uintptr_t. This will only be viable if you switch from using an unordered_map to a standard map, or defining a custom hashing function. Using Connection over uintptr_t would be simply using the abstractions provided to you rather than to have the class know about the implementation details defined in another class, which is a form of tight coupling.

- I noticed repeated strings for command comparison (e.g. "!LOGIN", "!SELECT", and "!NEW"). These strings should be defined as constants to avoid having to change each instance if the command word is redefined as another word. In addition, this would mitigate the risk of typing the command word incorrectly when adding new lines of code.

1.

I do not think RoomController should have the responsibility of getting the json file parsed. This binds everything in the json to the class. Resets in my opinion should not be part of this class. As you get to parsing that section, you will need to have a few helper methods to properly parse the reset objects. These functions in addition to the original json code should be put in its own class.

2.

Since addCharacterToRoom, addObjectToRoom, addDoorToRoom, only have int parameters, I assume the objects for these add functions do not live in RoomController itself. This seems odd because all you can get from this is if given an id, what room does it belong in. Given a room id, it does not seem like RoomController will be able to give me the existing NPC/other things in the room. Hopefully, you are making sure that the ids are unique, so you do not run into conflicts.

3.

Seeing your design for Authenticated commands, I think it would be a good idea to use that same design for unauthenticated commands. What mudServer's responsibility could be is to check the !command and call whatever appropriate action it needs. Having the class handle UseMgr does not make sense because it already deals with the initial command checking.

4

Why does UsrMgr have buildOutgoing? From the code before it, it looked like a class to contain the multiple connections in a class and to handle direct messaging. What I see is that for login, it looks like the messages are independent of the heartbeat system which makes sense. A user logging in should not be bound by the in-game world timing system. However UsrMgr is called directly as part of the core game loop.

Code Review #7 – Babka

---

class RoomController –– Responsibility, Dependency

`RoomController` should not know about JSON parser and source files. The class will only work if you provide a path to an existing JSON file.

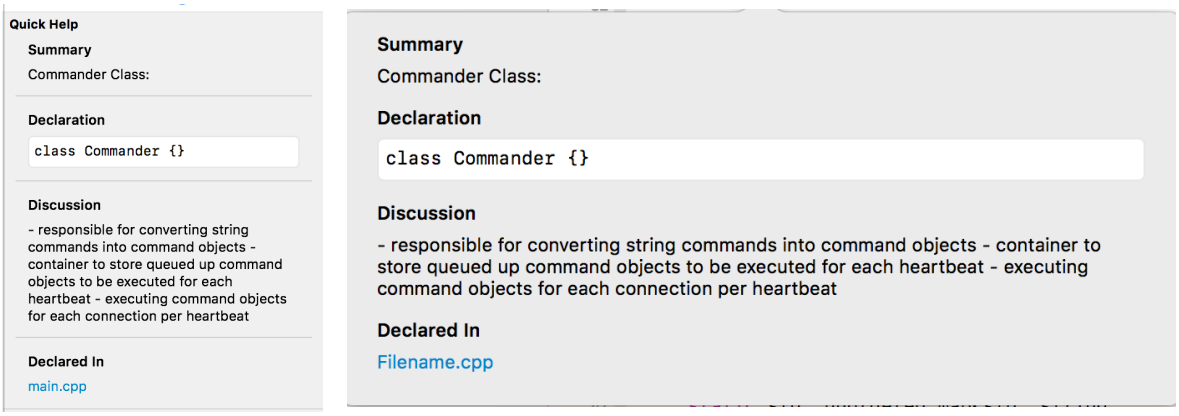
- Process the input files elsewhere to remove this unnecessary dependence. Pass only the required arguments to class constructor.
- All function arguments have type `int`. You allow arguments that might refer to `id` of another class. Make your functions more strongly-typed.
- Description of a room, characters, and doors most likely should not be a single string. Choose a different return type; for example, a struct that contains all three different strings.
- Room controller appears to be `AreaController`, and `WorldController` because it controls all of the existing rooms, not just a single room.
    - `Area` is responsible for:
      ```
      bool removeRoom(int id);
      bool removeDoorFromRoom(int doorID, int roomID);

      bool addCharacterToRoom(int npcID, int roomID);
      bool addObjectToRoom(int objectID, int roomID);
      bool addDoorToRoom(int doorID, int roomID);

      bool moveCharacter(int currentRoomID, int newRoomID);
      std::string getFullRoomDescription(int roomID);
      std::unordered_map<int, Room> rooms;
      ```
    - `World` is responsible for:
      ```
      std::unordered_map<std::string, Area> areas;
      ```

class Commander –– Implementation

- You do not need networking Message structure. Generalize functions to make your class work with any message that has a sender id and message text.
- There is no need for a constructor if the class has only static members.
- Fix comment formatting unless it is supported in your IDEs.



- Refactor code with repetitive variables and dependant types

```
    auto commKeywords = CommCommandsHelper::values();
    std::for_each(commKeywords.begin(), commKeywords.end(), [&](CommCommands c) {
        map.emplace(CommCommandsHelper::to_string(c), [&](networking::Connection id, std::string commandWord, std::string command) {return
std::make_unique<commands::Communicate>(id, commandWord, command);});
    });

    auto moveKeywords = MoveCommandsHelper::values();
    std::for_each(moveKeywords.begin(), moveKeywords.end(), [&](MoveCommands c) {
        map.emplace(MoveCommandsHelper::to_string(c), [&](networking::Connection id, std::string commandWord, std::string command) {return
std::make_unique<commands::Move>(id, commandWord, command);});
    });
```

And 6 more almost identical patterns:

```
auto X1Keywords = X1CommandsHelper::values();
std::for_each(X1Keywords.begin(), X1Keywords.end(), [&](X1Commands c)
    map.emplace(X1CommandsHelper::to_Type1(c), [&](networking::ConnectionId id,
                                        std::Type1 commandWords,
                                        std::Type1 command) {
        return std::make_unique<commands::X2>(id, commandWord, command);
    });
});
```

CMPT 373: Individual Code Review #7
Review Done on Group: Babka

1.

```
RoomController::RoomController(const std::string& pathToJSONFolder) {
    JSONParser json;
    if(is_directory(pathToJSONFolder)) {
        std::cout << "Debug info. Check area population" << std::endl;
        for (auto &entry : boost::make_iterator_range(directory_iterator(pathToJSONFolder), {})) {
            Area area = json.generateArea(entry.path().string());
            areas.insert({area.getName(), area});
            std::cout << entry << "\n";
        }
    }
}
```

Isolate the JSON mechanics, so that the room controller is only responsible for managing the rooms and not actually creating them. A suggestion I have for this, is to create a single class with helper functions that is responsible for the entire Area generation using the JSON. The current design suggests that the json is being parsed in many files respectively and is not a good design for possible changes. If the json file were to change drastically all instances of it would have to be updated and there is an added overhead of having to rad in the JSON file mutliple times.

2.

```
std::unique_ptr<Command> Commander::createNewCommand(const Message& msg) {

    std::string text = msg.text;
    boost::trim_if(text, boost::is_any_of(" "));
    boost::to_lower(text);
    auto commandWord = text.substr(0, text.find(' '));
    auto command = commandMap.find(commandWord);

    if(command != commandMap.end()) {
        return command->second(msg.connection, commandWord, text);
    }
    else {
        return commandMap[NotExistCommandsHelper::to_string(NotExistCommands::NOTEXIST)](msg.connection,
commandWord, text);
    }
}
```

There is a good use of enums through out the code and this a great sign that error checking is being done. Enums are a great design choices instead of booleans because you return a direct message as to what the functions/method has done, rather than having to make the API user guess what the meaning of true/false are.

3.

```
 std::unordered_map<std::string, std::function<std::unique_ptr<Command>(networking::Connection, std::string,
std::string)>> map;
    auto notExistKeywords = NotExistCommandsHelper::values();
    std::for_each(notExistKeywords.begin(), notExistKeywords.end(), [&](NotExistCommands c) {
```

```
        map.emplace(NotExistCommandsHelper::to_string(c), [&](networking::Connection id, std::string commandWord,
std::string command) {return std::make_unique<commands::CommandNotExist>(id, commandWord, command);});
    });
```

There is a good use of struct in place of significant data types that could be easily represented using numerical data dtypes like an int or long. The connection id is simply a large number to represent the unique connection of a user to the server and can be represented using an int, but it is better to use the networking::Connection data type as it prevents the possibilities of incorrect data passed into the function. If you had a function that took in a connection id and an int, and you represented the id with an int then the two can be easily swapped and cause many errors that would be hard to find.

4. This is not a design issue but the overall readability of the code is very bad. I have also seen some code from previous code review and it seems to be a consistent problem. In our readings we discussed the importance of communicating to other developers via code and how important it is to make sure they can understand it so errors can be track downed easily. This is any easy fix that will help a lot in the future when having to modify code for new features/requirments.

std::unordered_map<int, Room> rooms;

std::unordered_map<std::string, Area> areas;

- The room controller seems to own and manage different areas in the game. Areas and rooms are different in the context of this game, so it is confusing why is the room controller managing areas too.

- Structuring it this way where one room controller manages all of the rooms of all three areas, this might lead to players being able to go to rooms of another area if not handled properly. Having separate room controllers for each area gives decreases the likelihood of this happening.

```
if(is_directory(pathToJSONFolder)) {
    std::cout << "Debug info. Check area population" << std::endl;
    for (auto &entry : boost::make_iterator_range(directory_iterator(pathToJSONFolder), {})) {
      Area area = json.generateArea(entry.path().string());
      areas.insert({area.getName(), area});
      std::cout << entry << "\n";
```

- The responsibility of managing rooms and areas can be separated into two different classes, where room controller is responsible for adding/removing players/objects from rooms and area controller could be mapping which area an active player is currently in and reading and constructing each area from the json file.

```
void executeHeartbeat(std::unique_ptr<World>& world){
        …
        auto resultMessages = queue.front()->process(world);
        …
```

- executeHeartbeat() takes in unique_ptr to a World model, which is used when executing commands. Is this necessary? Based on the name, this ptr is a reference to the entire game model, which is the same as passing a sigleton of the entire game each time a command gets executed

- The server sleeps for one second each game loop, which it seems to be your way of handling heartbeats? This could be handled better instead of making the server wait for one second. Would the server receive any incomming messages while sleeping? Potentialy not receive some commands from players

Code Review for Babka

I am not sure about the [unordered map] of area being a member variable of the RoomController. I feel like a higher class like an AreaController should have a room controller instead.

Looking at the RoomController.cpp, it looks like you are parsing and generating the areas here as part of the constructor. That does not seem like it should be the responsibility of the RoomController as rooms are just part of the bigger area. This should probably be done in a some kind of a parser class instead.

Some lines of code are a little too long, which makes them hard to read and understand.

Related to the previous point, I find the code for Commander::commandMap() a little hard to read. It is a bit hard to follow. Better formatting should help make it easier to understand.

Code review 7
Vincent Lam
vtlam@sfu.ca
301228373

1.

```
bool moveCharacter(int currentRoomID, int newRoomID);\
```

I cannot say without looking at the full source code, and while its logical that a room controller have rooms methods in a sense, I personally find it an odd design choice, or rather a few odd choices. For instance, why return a boolean, and why only need to know the two room ids? Would a character be able to move itself, or would it need the game behavior above to do it?

A personal choice would likely involve this method being a "request", and have multiple overloaded parameters to choose from, that way many characters may have their own "move" but they may or may not have access to this method, in which they could have another one to use, like moveWorlds..etc.etc.

2.

```
std::unordered_map<std::string,
std::function<std::unique_ptr<Command>(networking::Connection, std::string, std::string)>>
Commander::commandMap = [](){

    std::unordered_map<std::string,
std::function<std::unique_ptr<Command>(networking::Connection, std::string, std::string)>>
map;
    auto notExistKeywords = NotExistCommandsHelper::values();
    std::for_each(notExistKeywords.begin(), notExistKeywords.end(), [&](NotExistCommands c) {
        map.emplace(NotExistCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::CommandNotExist>(id, commandWord, command);});
    });

    auto commKeywords = CommCommandsHelper::values();
    std::for_each(commKeywords.begin(), commKeywords.end(), [&](CommCommands c) {
```

```
        map.emplace(CommCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::Communicate>(id, commandWord, command);});
    });


    auto moveKeywords = MoveCommandsHelper::values();
    std::for_each(moveKeywords.begin(), moveKeywords.end(), [&](MoveCommands c) {
        map.emplace(MoveCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::Move>(id, commandWord, command);});
    });

    auto lookKeywords = LookCommandsHelper::values();
        std::for_each(lookKeywords.begin(), lookKeywords.end(), [&](LookCommands c) {
        map.emplace(LookCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::Look>(id, commandWord, command);});
    });

    auto itemKeywords = ItemCommandsHelper::values();
    std::for_each(itemKeywords.begin(), itemKeywords.end(), [&](ItemCommands c) {
        map.emplace(ItemCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::CommandItem>(id, commandWord, command);});
    });

    auto combatKeywords = CombatCommandsHelper::values();
    std::for_each(combatKeywords.begin(), combatKeywords.end(), [&](CombatCommands c) {
        map.emplace(CombatCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::CommandCombat>(id, commandWord, command);});
    });

    auto swapKeywords = SwapCommandsHelper::values();
    std::for_each(swapKeywords.begin(), swapKeywords.end(), [&](SwapCommands c) {
        map.emplace(SwapCommandsHelper::to_string(c), [&](networking::Connection id,
std::string commandWord, std::string command) {return
std::make_unique<commands::CommandSwap>(id, commandWord, command);});
    });
    return map;
}();
```

Alot of typedef would help here, also perhaps helper functions would really simplify what this does and how.

3.

```
void
processAuthenticatedMessages(const Message& message) {
   // is LOGOUT? else it's a command
   if(boost::contains(message.text, "!LOGOUT")) {
      UsrMgr.logout(message.connection);
      UsrMgr.printAllUsers();
   }else if(UsrMgr.ifHasActiveAvatar(message.connection)){
      if(boost::contains(message.text, "!SWITCH")){
         //TODO: Implement the switch of characters while in game
         UsrMgr.setHasActiveAvatar(message.connection, false);
      }else {
         cmdQueue.addCommand(message,
std::move(Commander::createNewCommand(message)));
      }
   }else{
      if(boost::contains(message.text, "!SELECT")){
         std::cout << "calling !SELECT" << std::endl;
         UsrMgr.setHasActiveAvatar(message.connection, true);
         UsrMgr.sendMessage(message.connection, "You've selected a character");
      }else if(boost::contains(message.text, "!NEW")){
         std::cout << "calling !NEW" << std::endl;
         //UsrMgr.setHasActiveAvatar(message.connection, true);
         UsrMgr.sendMessage(message.connection, "You've created a new character! Now call
!SELECT [avatar_name] to choose a character.");
      }else{
         UsrMgr.sendMessage(message.connection, "To select an existing avatar: !SELECT
[avatar_name]\nTo create a new avatar: !NEW [avatar_name]\nYour avatars: Swordmaster101,
thievingBoss, swedishfish");
      }
   }
}
```

Despite a commander class, there is alot of boolean logic for understanding a certain command here. Noticing how your class handles function pointers as commands, why not add these in as well with a public method in Commander?

4.

```
//parses command string, creates a command object and stores it in bufferedCommands
std::unique_ptr<Command> Commander::createNewCommand(const Message& msg) {

    std::string text = msg.text;
    boost::trim_if(text, boost::is_any_of(" "));
    boost::to_lower(text);
    auto commandWord = text.substr(0, text.find(' '));
    auto command = commandMap.find(commandWord);

    if(command != commandMap.end()) {
        return command->second(msg.connection, commandWord, text);
    }
    else {
        return
commandMap[NotExistCommandsHelper::to_string(NotExistCommands::NOTEXIST)](msg.con
nection, commandWord, text);
    }
}
```

Another small note here is to perhaps clean the last line here into more readable and simplier code.