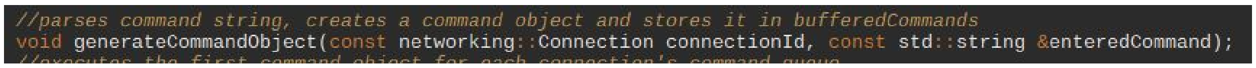
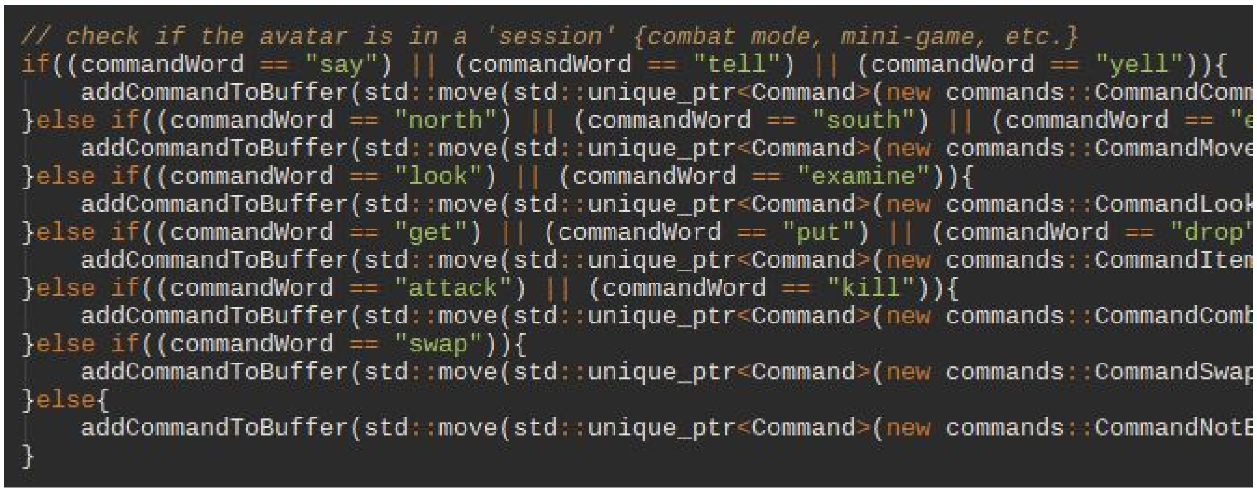
CMPT 373 Individual Code Review 5

This is a review of the code submitted by group babka.



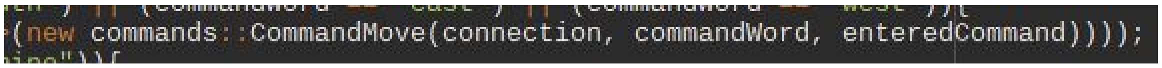
```
//parses command string, creates a command object and stores it in bufferedCommands
void generateCommandObject(const networking::Connection connectionId, const std::string &enteredCommand);
//executes the first command object for each connection's command queue
```

The generateCommandObject function could be improved. First, connectionId could be passed by const reference. The name of the function is also unclear based on the comment describing what the method does. Instead, “add” of “buffer” could be used instead of “generate.”



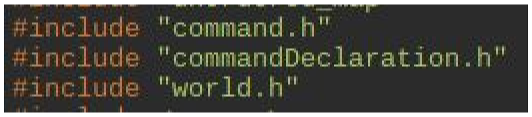
```
// check if the avatar is in a 'session' {combat mode, mini-game, etc.}
if((commandWord == "say") || (commandWord == "tell") || (commandWord == "yell")){
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandComm
}else if((commandWord == "north") || (commandWord == "south") || (commandWord == "e
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandMove
}else if((commandWord == "look") || (commandWord == "examine")){
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandLook
}else if((commandWord == "get") || (commandWord == "put") || (commandWord == "drop"
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandItem
}else if((commandWord == "attack") || (commandWord == "kill")){
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandComb
}else if((commandWord == "swap")){
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandSwap
}else{
    addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandNotE
}
```

These complicated conditional statements are hard to read. Using boolean variables representing the more complex expressions would improve readability.



```
(new commands::CommandMove(connection, commandWord, enteredCommand))));
```

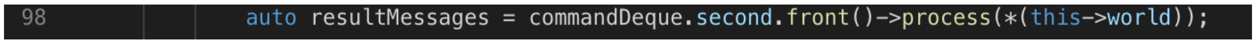
Passing both commandWord and enteredCommand is redundant when making new command objects. The command and then the string containing the remaining command parameters could be passed instead.



```
#include "command.h"
#include "commandDeclaration.h"
#include "world.h"
```

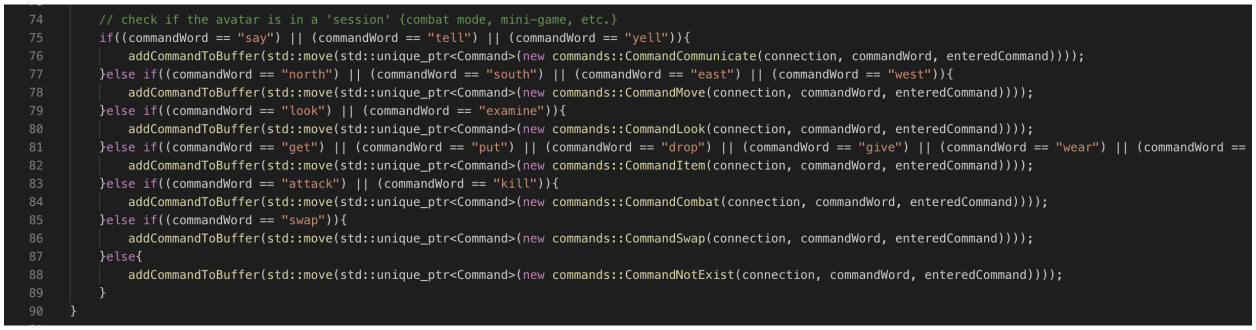
I noticed that classes like World are declared with capitals, but your model class files are not. This could be changed for consistency. It is common C++ convention to declare classes with the first letter of every word capitalized. So “world.h” could be changed to “World.h” to reflect that it is the header file for the “World” class and so on.

Jycasas - babka



```cpp
        auto resultMessages = commandDeque.second.front()->process(*(this->world));
```

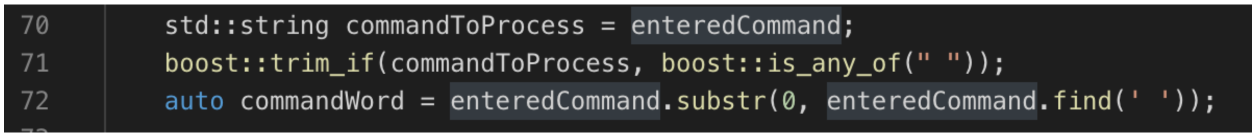
I like the overall concept of the function however, I would assume the world object is massive that contains all the object in the world. This could become an issue that might cause the game run slower.



```cpp
74      // check if the avatar is in a 'session' {combat mode, mini-game, etc.}
75      if((commandWord == "say") || (commandWord == "tell") || (commandWord == "yell")){
76          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandCommunicate(connection, commandWord, enteredCommand))));
77      }else if((commandWord == "north") || (commandWord == "south") || (commandWord == "east") || (commandWord == "west")){
78          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandMove(connection, commandWord, enteredCommand))));
79      }else if((commandWord == "look") || (commandWord == "examine")){
80          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandLook(connection, commandWord, enteredCommand))));
81      }else if((commandWord == "get") || (commandWord == "put") || (commandWord == "drop") || (commandWord == "give") || (commandWord == "wear") || (commandWord ==
82          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandItem(connection, commandWord, enteredCommand))));
83      }else if((commandWord == "attack") || (commandWord == "kill")){
84          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandCombat(connection, commandWord, enteredCommand))));
85      }else if((commandWord == "swap")){
86          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandSwap(connection, commandWord, enteredCommand))));
87      }else{
88          addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandNotExist(connection, commandWord, enteredCommand))));
89      }
90  }
```

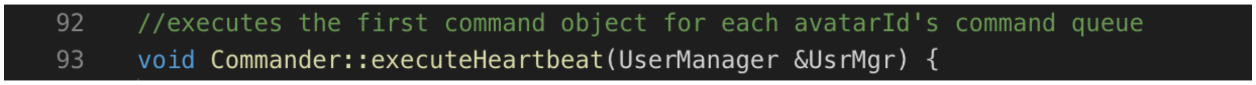
I think it would be better to simplify these conditional statements. Create a variable that holds the values of the expression or maybe pass a struct that depicts the type of command it is to make it more workable.

Also, on line 81, the conditional statement is too long which ruins the readability of the code.



```cpp
    std::string commandToProcess = enteredCommand;
    boost::trim_if(commandToProcess, boost::is_any_of(" "));
    auto commandWord = enteredCommand.substr(0, enteredCommand.find(' '));
```

Passing a struct here would be better to increase readability instead of passing a string. Creating this struct earlier on can help increase the readability.



```cpp
//executes the first command object for each avatarId's command queue
void Commander::executeHeartbeat(UserManager &UsrMgr) {
```

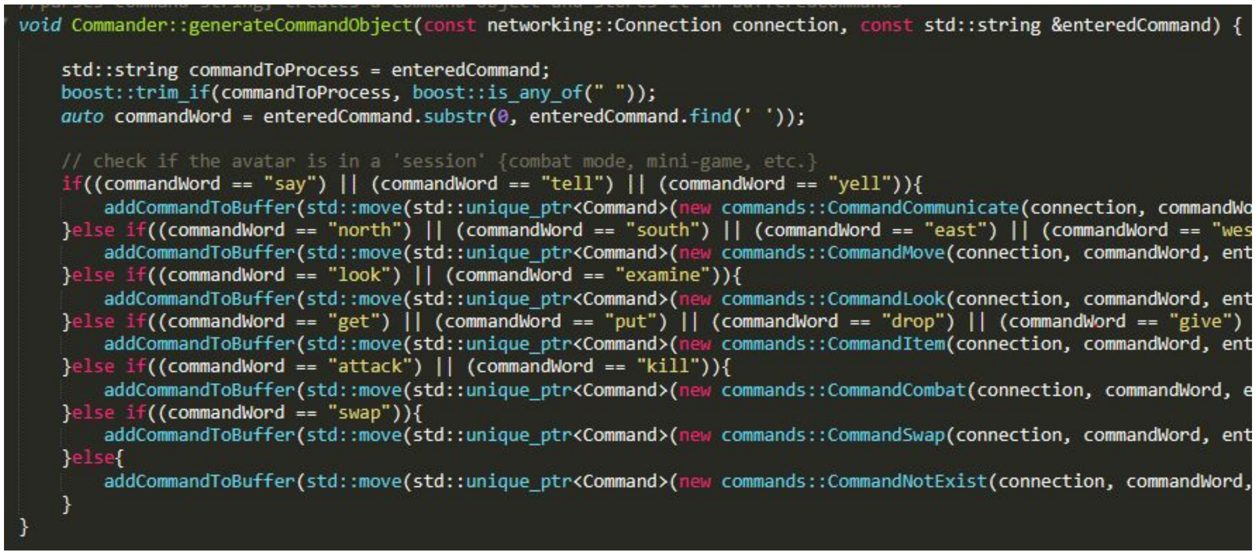
I think it would be better to just have a composition of the UserManger object. There could be other cases in the future that might use this.

Overall:
I like the general concept of the game. I understand the direction and can easily understand what the goal of the programmer. Just some minor fixes can improve the readability of the code.

Individual Code Review 5

Are there alternative designs that could provide better trade offs?

#1. Issue: if and else-ifs; use switch statement instead.



```
void Commander::generateCommandObject(const networking::Connection connection, const std::string &enteredCommand) {

    std::string commandToProcess = enteredCommand;
    boost::trim_if(commandToProcess, boost::is_any_of(" "));
    auto commandWord = enteredCommand.substr(0, enteredCommand.find(' '));

    // check if the avatar is in a 'session' {combat mode, mini-game, etc.}
    if((commandWord == "say") || (commandWord == "tell") || (commandWord == "yell")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandCommunicate(connection, commandWo
    }else if((commandWord == "north") || (commandWord == "south") || (commandWord == "east") || (commandWord == "wes
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandMove(connection, commandWord, ent
    }else if((commandWord == "look") || (commandWord == "examine")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandLook(connection, commandWord, ent
    }else if((commandWord == "get") || (commandWord == "put") || (commandWord == "drop") || (commandWord == "give")
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandItem(connection, commandWord, ent
    }else if((commandWord == "attack") || (commandWord == "kill")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandCombat(connection, commandWord, e
    }else if((commandWord == "swap")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandSwap(connection, commandWord, ent
    }else{
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandNotExist(connection, commandWord,
    }
}
```

The main issue is with readability and code clarity. There is a lot of if and else if being used here which can be simplified by using a switch or case statement instead. You can also create the default case and use it for error-handling purposes which is useful.

You can have a look at this reference: https://en.cppreference.com/w/cpp/language/switch

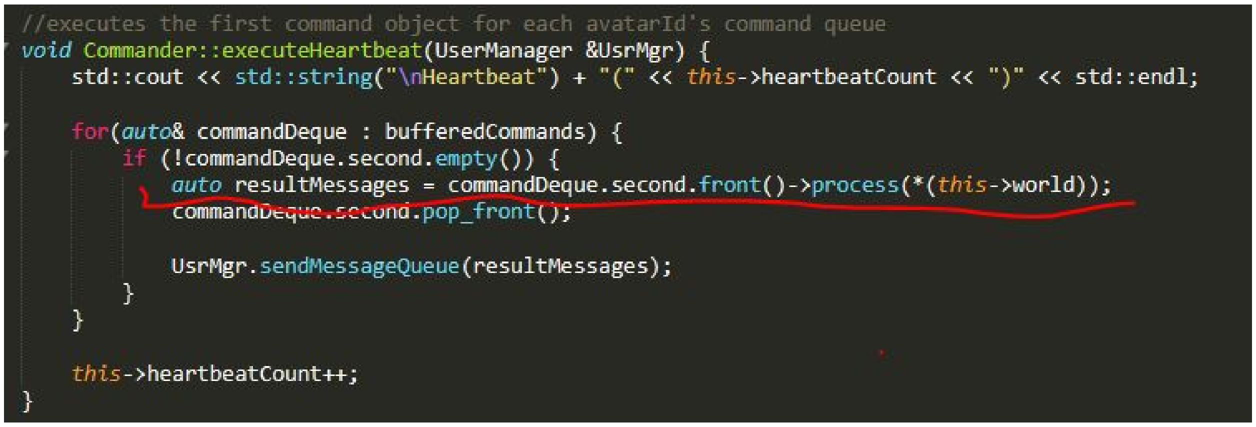
Are errors handled properly?

Issue: lowercase vs uppercase commands

Regarding the same code above, I'm not sure if this situation is being handled in any of your other classes, but I'm wondering what if the user enters a command like "Say" or "SAY", does your program realize that it is the same thing? It would be nice to add this safety measure by writing code that transfers all lowercase to uppercase or vice versa or have something for error-handling purposes.

#2. Are control structures deeply nested?

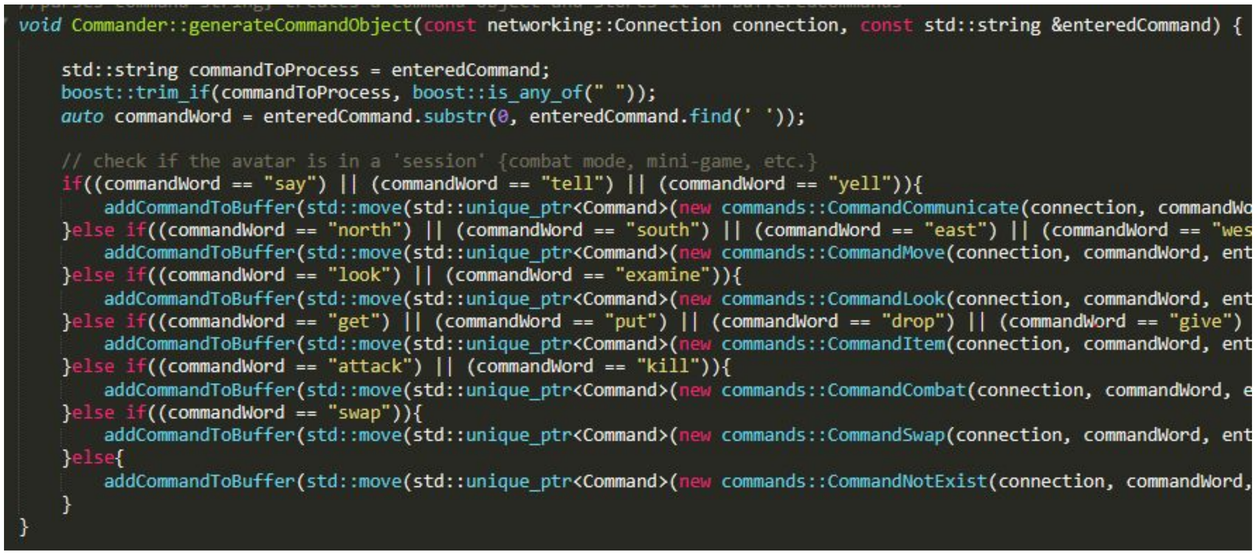
Issue: complex code



```cpp
//executes the first command object for each avatarId's command queue
void Commander::executeHeartbeat(UserManager &UsrMgr) {
    std::cout << std::string("\nHeartbeat") + "(" << this->heartbeatCount << ")" << std::endl;

    for(auto& commandDeque : bufferedCommands) {
        if (!commandDeque.second.empty()) {
            auto resultMessages = commandDeque.second.front()->process(*(this->world));
            commandDeque.second.pop_front();

            UsrMgr.sendMessageQueue(resultMessages);
        }
    }

    this->heartbeatCount++;
}
```

I really have a hard time understanding that underlined line of code. Personally, I would prefer to break down that line of code to multiple lines with the use of meaningful variables, so I know what each function and pointer returns. In that case, it is easier to know what is being sent as input and received as output.

#3. Are there alternative designs that could provide better trade offs?



```cpp
void Commander::generateCommandObject(const networking::Connection connection, const std::string &enteredCommand) {

    std::string commandToProcess = enteredCommand;
    boost::trim_if(commandToProcess, boost::is_any_of(" "));
    auto commandWord = enteredCommand.substr(0, enteredCommand.find(' '));

    // check if the avatar is in a 'session' {combat mode, mini-game, etc.}
    if((commandWord == "say") || (commandWord == "tell") || (commandWord == "yell")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandCommunicate(connection, commandWo
    }else if((commandWord == "north") || (commandWord == "south") || (commandWord == "east") || (commandWord == "wes
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandMove(connection, commandWord, ent
    }else if((commandWord == "look") || (commandWord == "examine")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandLook(connection, commandWord, ent
    }else if((commandWord == "get") || (commandWord == "put") || (commandWord == "drop") || (commandWord == "give")
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandItem(connection, commandWord, ent
    }else if((commandWord == "attack") || (commandWord == "kill")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandCombat(connection, commandWord, e
    }else if((commandWord == "swap")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandSwap(connection, commandWord, ent
    }else{
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandNotExist(connection, commandWord,
    }
}
```

Suggestion: Use BETTER_ENUM in C++17

Also another suggestion would be to use an enum class for the commands if it is possible. There is this feature in C++17 that allows you to create the enum and then give it its keywords.
For example, BETTER_ENUM(Combat, attack, kill), here Combat is the name of the enum class and attack and kill are the keywords or options that is within enum Combat.

Reference: http://aantron.github.io/better-enums/ApiReference.htm

Improvements:

#4. Is the intent of the code clear? Do comments clarify or just repeat what the code does?

The design of the class, Commander was clear, and the function names speak for themselves. Also, In some situations where the code was complex or variable names were confusing, your comments really helped me to understand the purpose of the code and function within the class. As a code reviewer, I thank you for adding comments.

Also… use of external libraries such as Boost

I would like to give a bonus point to the use of external libraries such as Boost to parse the command string in order to create a command object instead of implementing those methods on your own which probably would have been time consuming, error prone, and less efficient.

Code Review #5 – Babka

class Commander — Responsibility

```
/**
 * Commander Class:
 *
 * - responsible for converting string commands into command objects
 * - container to store queued up command objects to be executed for each heartbeat
 * - executing command objects for each connection per heartbeat
 *
 */
```

Reads *"Commander Class: violates single responsibility principle"*

Although all steps are related to in-game commands, you identify three distinguishable tasks:
1. *parse* textual representations of commands;
2. *store* parsed commands in a buffer; and
3. *execute* commands when heartbeat happens.

Alternative solution would be to implement additional classes, for example `CommandParser` and `CommandBuffer` (optional), and handle related responsibility there. They will *parse* strings and *store* commands for `Commander` class to *execute* when needed.

The name is misleading. Word *"commander"* is defined as *"a person in authority, especially over a body of troops"* or *"a military operation* or *human or group who manages effort of an organization."* Think of a class name that will relate to the functionality you provide.

class Commander — Types, Format, and Style

```
std::unordered_map<uintptr_t, std::deque<std::unique_ptr<Command>> > bufferedCommands;
//map stores {avatarId, commandObjectQueue} pairs
```

This type is difficult to understand. Types you use are too complex and require an additional comment to describe the meaning. Anyone else would not know the meaning of the first type parameter and what it maps to.

Spacing leads to an incorrect assumption that this line compares two values.

Line length is too long. Avoid any line of code that exceeds 80 columns.

Filenames conventionally have the same name as the declared class. Rename your header files, i.e. ~~`"command.h"`~~ to `"Command.h"`.

Inconsistent names and types are used throughout the class. If an argument refers to the same object, choose a name and use it for all declarations. Rename ~~`conn`~~ to `connectionId`.

Try one or more alternative solutions below to increase readability of your code:

1. hide the underlying map by creating a `CommandBuffer` class to deal with storage, as described above:
   ```
   CommandBuffer bufferedCommands;
   ```

2. minimize `std` scope resolution with `using` directives:
   ```
   using std::unordered_map;
   using std::deque;
   using std::unique_ptr;
   unordered_map<uintptr_t, deque<unique_ptr<Command>>> bufferedCommands;
   ```

3. use type aliases to describe the type of class data members:
   ```
   using AvatarId = uintptr_t;
   using CommandDeque = std::deque<std::unique_ptr<Command>>;
   using Buffer = std::unordered_map<AvatarId, CommandDeque>;
   Buffer bufferedCommands;
   ```

4. make use of the existing `Connection` and `ConnectionHash` structures.

---

class Commander — Implementation

1. Dependence of the only constructor on `World` is unnecessary. Data member `world` used only once. If you will not use it in other functions, remove it and instead pass it as an `executeHeartbeat` argument; or choose another way.

2. The function `generateCommandObject` repeatedly compares value of the first word in an long `if`, `else if`, `else`. Reduce the unreadable conditional to a 1-line function call by extracting a function that, given entered command, returns the correct command to add to the buffer.

3. It is unclear how you perform deallocation of commands. If you are not using `delete` properly, you may be leaking memory.

4. Implementation is very difficult to follow. Anyone who is not familiar with all of: `std::deque`, as well as your implementations of `Commander`, `Command`, and `World` classes will not understand what is assigned to the `resultMessages`. Do not chain data or function members; instead, use a variable for an intermediate step. *Especially* when you will use it for other computations.

   ~~`auto resultMessages = commandDeque.second.front()->process(*(this->world));`~~

5. Only the programmer who implemented this class can easily understand how this class works. The desired result is clear, but implementation is puzzling. You must review your header and implementation files and write self-documenting code describing its purpose. Assume the reader is a beginner and does not know anything about your project.

   C++ Core Guidelines

Individual Code Review 5

1. Unnecessary comments (lines 1-3 ,29, 55 – 57)

   Comments such as //Constructor are unnecessary as the constructor should be obvious to any programmer reading it. It is good to omit unnecessary comments as they can proliferate as you work on the project and they will make the code harder to read in the future.

2. Using new

   The teacher has mentioned that we should not be using new in c++. If you choose to use new, then you must manage that resource created and this leave room for errors in the code. Thus, it is better to avoid it since its not necessary anyways.

3. Class name doesn't make sense

   In my opinion the class name "commander" is not descriptive enough. When I read commander, I think of a military person. Thus, I think you should rename it to CommandHandler as I believe this reveals its true intent right away.

4. Hard-coded strings (lines 75-85)

   You are using a lot of hard coded strings in the function generateCommandObject(). You should refactor this into a string constant like: const std::string SAY_CMD = "say"; . To me this is more readable as we know this string is for a string command. Also, if in the future you want to change or delete this command, its easier as when this is changed, your IDE will highlight all the areas using this constant. Furthermore, you could refactor it to a map of enum strings and get the benefits of enum as well.

Code review - Bakka

1. Using if/else too much could create a problem with run time because it's technically a O(N) operation and if you want to add (a lot) more commands, this would be the bottleneck of the game.

2. I think putting each command in its own object is a good idea, but maybe there's a better way to do this like inheritance/polymorphism to make the code shorter because now you're passing (connection, commandWord, enteredCommand) for every type. I think with the virtual function you may only need to just write 1 line for any of the command.

3. To me, there's really a problem with the code format because it's just too long with a lot of chaining. But you can improve it easily.

For example, all the || conditions can be put on separate lines to line them up, easier to know what word you're talking about.

Secondly, the line auto resultMessages = commandDeque.second.front()->process(*(this->world)) maybe a bit hard to understand, My guess is it's calling the function from a pointer, passing this->world as a parameter. My group had the same thing and a comment from the prof was better not using the "clever" part of the language too much because it would be harder to extract the meaning. Instead it can be done using inheritance/polymorphism.

And lastly, make some middle steps instead of chaining too much for lines like this addCommandToBuffer(std::move(std::unique_ptr<Command>(new commands::CommandNotExist(connection, commandWord, enteredCommand))));

4. There could a security issue if you pass world object around to do the commands. It's not wrong but can create a bit of problems because now everything has access to change the world so you must be more careful with that. I would switch the call the other way round, which is the world controls itself and it commands to process.

1.

```
void Commander::generateCommandObject(const networking::Connection connection, const
std::string &enteredCommand) {

    std::string commandToProcess = enteredCommand;
    boost::trim_if(commandToProcess, boost::is_any_of(" "));
    auto commandWord = enteredCommand.substr(0, enteredCommand.find(' '));

    // check if the avatar is in a 'session' {combat mode, mini-game, etc.}
    if((commandWord == "say") || (commandWord == "tell") || (commandWord == "yell")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandCommunicate(connection, commandWord, enteredCommand))));
    }else if((commandWord == "north") || (commandWord == "south") || (commandWord == "east") ||
(commandWord == "west")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandMove(connection, commandWord, enteredCommand))));
    }else if((commandWord == "look") || (commandWord == "examine")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandLook(connection, commandWord, enteredCommand))));
    }else if((commandWord == "get") || (commandWord == "put") || (commandWord == "drop") ||
(commandWord == "give") || (commandWord == "wear") || (commandWord == "remove")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandItem(connection, commandWord, enteredCommand))));
    }else if((commandWord == "attack") || (commandWord == "kill")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandCombat(connection, commandWord, enteredCommand))));
    }else if((commandWord == "swap")){
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandSwap(connection, commandWord, enteredCommand))));
    }else{
        addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandNotExist(connection, commandWord, enteredCommand))));
    }
```

This looks extremely odd and unreadable. This suggests you could turn some of these boolean logics into helper functions, and use Enums to give more precision to the code. Furthermore, I would suggest that since many of these constructors have identitcal parameters, you may pass in a struct holding all of these instead. This allows for the possibilities of bugs occuring in this body to be at a minimum.

2.

```
//executes the first command object for each avatarId's command queue
void Commander::executeHeartbeat(UserManager &UsrMgr) {
    std::cout << std::string("\nHeartbeat") + "(" << this->heartbeatCount << ")" << std::endl;

    for(auto& commandDeque : bufferedCommands) {
        if (!commandDeque.second.empty()) {
            auto resultMessages = commandDeque.second.front()->process(*(this->world));
            commandDeque.second.pop_front();

            UsrMgr.sendMessageQueue(resultMessages);
        }
    }

    this->heartbeatCount++;
}
```

Not sure if this is the intended effect, but this essentially preforms all the possible commands in each buffer. The thing to note here is there is that it preforms iterating by each connection (each user).
So if user A connects first, and user B connects second. User B will always have second priority each heartbeat over every single action User A has.

3.

```
addCommandToBuffer(std::move(std::unique_ptr<Command>(new
commands::CommandSwap(connection, commandWord, enteredCommand))));
```

Another thing I noticed is that, theres seperate objects for each type of command, yet the commandWord, the defining characteristic is passed in, which seems rather redudant. Personally I'd have a base Command class with a method that can identify which of its derived classes should handle the /command. (We learned this in class where a superclass can get info about a derived class). And the derived class can perhaps specify itself further. Such as for /give you can have a BaseCommand("/give") turns into ItemCommand("/give") which turns into GiveCommand(..).

4.

```
void Commander::generateCommandObject(const networking::Connection connection, const
std::string &enteredCommand) {
```

Again, alot of these suggestions are purely preference, but it seems here the choice is the have Connection to be the pure defining factor for each account. As my team did the same, the main difficulty is now any thing your conneciton does, would require a method to identify the connection with every single command, event, and data handling for it. This can be huge nightmare where, lets say, an event in the game damages someone in the game, gives them a status or item, or more effects, since there are multiple calls to find the same Avatar for one event. What I suggest is to give the Connection Container class a bit more depth. For instance, in our design, we have a User class, which also holds a Connection object, and is hashed using it. Furthermore, we attached our "avatar", and "account info" to each User, so that we can immediately grab the data collection without looking up what the connection is again.