Individual Code Review 6

1. The Commander class has a notion of a world. This makes the world tightly coupled with the Commander class. The Commander should only have the responsibility of managing the conversions to strings to command objects.

2. The Commander class also has a `respondToClient` method that takes in a Connection object, a UserManager and a deque of messages. This couples the messaging system to the Commander class. A better alternative is to focus the Commander class to just convert strings to Command objects. A separate class should be created and expose an API that accepts connections and messages. This separates the responsibility of the Commander class and messaging

3. In the commander class there is a commandMap that has the following type:

```
std::unordered_map<std::string,
std::function<std::unique_ptr<Command>(networking::Connection, std::string,
std::string)>> commandMap
```

   A better alternative is to use type aliases for reability. For example:

```
using CommandKeyWord = std::string;
using CommandParameters = std::string;
using CommandFunctionPtr =
std::function<std::unique_ptr<Command>(networking::Connection, CommandKeyWord,
CommandParameters)>;
using CommandMap = std::unordered_map<std::string, CommandFunctionPtr>;
```

   Giving names to the parameters of the Command object such as CommandKeyWord, CommandParameters provides more contexts to other members of the team.

4. Commander also knows about heartbeats and execution of commands. Those should be separated out of commander and should be in their own entity. For example, adding a CommandExecutor class solely for executing commands from a queue. The owner of the Commander class will build the commands, then the inputs of the user to be converted into a Command, then sent to CommandExecutor for processing. CommandExecutor can even use a class to communicate outputs to the user.

Code Review 6

```
std::unordered_map<itemID, int> invContainer;
```

First thing I noticed was that you store items using their ID as the key. This is something that I struggled with in terms of settling on its implementation details when designing the Inventory class as well.

For this to work, you would most likely require another map containing the object, or some data structure to return the corresponding object when passing in an ID.

I don't have access to the entire project, so maybe you guys did figure out an awesome way to circumnavigate that problem, and that's great, but at a glance there's something about this implementation that stands out to me.

```
public:
    enum itemType {
        CONSUMABLE, EQUIPMENT, MISC
    };

    baseItem(int type, std::string name, std::string desc, bool quest, bool dropp

    // Getters
    const itemType getType() const { return itemClass; };

    const std::string getName() const { return itemName; };

    const std::string getDesc() const { return itemDesc; };

private:
    itemType itemClass;

    std::string itemName;
    std::string itemDesc;

    bool isQuestItem;
    bool isDroppable;
};
```

Second thing, the implementation of **baseItem** class. The json files that he provided us have a list of keywords to determine the object. I see that the **getName** method only returns a single string. Because I don't have the .cpp file for **baseItem**, I don't know how it's implemented, and this raises a concern. Items also have a short description, long description, and very long description, so it's good to also keep that in mind.

Maybe it returns the first keyword in the list, or maybe you guys aren't using the sample data that Sumner provided. However, this is something to keep an eye out and make sure that this wasn't a mistake.

```
void Commander::setUpFunctionMap() {
    auto notExistKeywords = NotExistCommandsHelper::values();
    std::for_each(notExistKeywords.begin(), notExistKeywords.end(), [this](NotExistCommands
        commandMap.emplace(NotExistCommandsHelper::to_string(c), [&](networking::Connection
    });

    auto commKeywords = CommCommandsHelper::values();
    std::for_each(commKeywords.begin(), commKeywords.end(), [this](CommCommands c) {
        commandMap.emplace(CommCommandsHelper::to_string(c), [&](networking::Connection id,
    });


    auto moveKeywords = MoveCommandsHelper::values();
    std::for_each(moveKeywords.begin(), moveKeywords.end(), [this](MoveCommands c) {
        commandMap.emplace(MoveCommandsHelper::to_string(c), [&](networking::Connection id,
    });

    auto lookKeywords = LookCommandsHelper::values();
        std::for_each(lookKeywords.begin(), lookKeywords.end(), [this](LookCommands c) {
        commandMap.emplace(LookCommandsHelper::to_string(c), [&](networking::Connection id,
    });

    auto itemKeywords = ItemCommandsHelper::values();
    std::for_each(itemKeywords.begin(), itemKeywords.end(), [this](ItemCommands c) {
        commandMap.emplace(ItemCommandsHelper::to_string(c), [&](networking::Connection id,
    });

    auto combatKeywords = CombatCommandsHelper::values();
    std::for_each(combatKeywords.begin(), combatKeywords.end(), [this](CombatCommands c) {
        commandMap.emplace(CombatCommandsHelper::to_string(c), [&](networking::Connection i
    });

    auto swapKeywords = SwapCommandsHelper::values();
    std::for_each(swapKeywords.begin(), swapKeywords.end(), [this](SwapCommands c) {
        commandMap.emplace(SwapCommandsHelper::to_string(c), [&](networking::Connection id,
    });
}
```

This function can also be modularized into independent methods. **setUpFunctionMap** can rather call methods such as **setupMoveCommandHelper, setupLookCommandHelper**, and more, instead of having this super method.

```
//parses command string, creates a command object and stores it in bufferedCommands
void Commander::createNewCommand(const networking::Connection connectionId, const std::string

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

    if(command != commandMap.end()) {
        addCommandToBuffer(command->second(connectionId, commandWord, enteredCommand));
    }
    else {
        addCommandToBuffer(commandMap[NotExistCommandsHelper::to_string(NotExistCommands::NOTE
    }
    return commandMap.count(commandWord) == 0 ?
}
```

Is there a reason why the first argument isn't a const reference like the second? Why is this void function returning an incomplete ternary operator? This **createNewCommand** method also seems to be breaking the single responsibility principle.

If I am designing a **createNewCommand** method, I expect the preparation of the command I am creating to be done somewhere else, rather than within the method **create**. The entire first half of the create method can be put into its own method and be called either a line before the create is called, or within the create method, but by its method name, not the raw implementation detail.

Here's how I would do it:

```
std::string parseCommand(const std::string &command) {
    std::string commandToProcess = command;

    boost::trim_if(commandToProcess, boost::is_any_of(" "));
    boost::to_lower(commandToProcess);

    return enteredCommand.substr(0, enteredCommand.find(' '));
}

//parses command string, creates a command object and stores it in bufferedCommands
void Commander::createNewCommand(const networking::Connection &connectionId, const std::string &enteredCommand) {
    auto command = commandMap.find(enteredCommand);

    (command != commandMap.end()) ?
        addCommandToBuffer(command->second(connectionId, commandWord, enteredCommand)) :
        addCommandToBuffer(commandMap[NotExistCommandsHelper::to_string(NotExistCommands::NOTEXIST)](connectionId
}
```

and here's main:

```
int main(...) {
    auto id = ...;
    auto command = ...;

    createNewCommand(id, parseCommand(command));
}
```

A minor gripe that I also have is with the spacing, or lack there of:

```
// Will be moved to main server loop
void Commander::addCommandToBuffer(std::unique_ptr<Command> command) {
    auto connectionId = command->getCallerConnectionId();
    auto avatarCommandDeque = bufferedCommands.find(connectionId);
    if(avatarCommandDeque != bufferedCommands.end()){
        avatarCommandDeque->second.push_back(std::move(command));
    }else{
        std::deque<std::unique_ptr<Command>> newCommandDeque;
        newCommandDeque.push_back(std::move(command));
        bufferedCommands.insert({connectionId, std::move(newCommandDeque)});
    }


}
```

To something like:

```
// Will be moved to main server loop
void Commander::addCommandToBuffer(std::unique_ptr<Command> command) {
    auto connectionId = command->getCallerConnectionId();
    auto avatarCommandDeque = bufferedCommands.find(connectionId);

    if(avatarCommandDeque != bufferedCommands.end()){
        avatarCommandDeque->second.push_back(std::move(command));
    }
    else{
        std::deque<std::unique_ptr<Command>> newCommandDeque;

        newCommandDeque.push_back(std::move(command));
        bufferedCommands.insert({connectionId, std::move(newCommandDeque)});
    }


}
```

This allows better readability to whoever is maintaining the code, and groups related details together. At a glance, which one is easier to look at and understand?

Individual Code Review 6

#1. Is the intent of the code clear?

a) Functions passed as template argument:

```
template<class KEY, class T, class D>
struct FunctionMap {
    std::unordered_map<KEY, T)>> map;
    T& operator[](const std:string& index) {
        //
    }
};
```

Possible problems/ideas:
According to this Stack Overflow Link, the use of template will be tricky with inline functions. Also, other problems can arise with template argument deduction, and for more information you can look into Template Argument Deduction Link.

Intent of the code:
I understand the use of wrapper functions is to simplify calling of another function, but to me it is not evident what your intention is in terms of using it. What are you planning to use this code for? What are you going to do with it? Please provide these details in the comments.

#2. Are there alternative designs that could provide better trade offs?
a) Yes, about the Inventory class you provided, I realized you guys are using 'int' as the itemID. This can work, but may not be efficient for future iterations because the type of itemID can be changed to string, long, or any other type other than an int. To fix this, you can use a struct ID, and have a operator == written for it so you can do the ID comparisons.

Here is an example:

```
struct Id {
    long long id{};

    Id() {};

    Id(long long i) : id{i} {}

    bool operator==(const adventure::model::Id &other) const { return (id == other.id); }
};
```

b) In order to integrate the type of Id for itemID within your inventory map, you need to create another struct for the IdHasher. C++17 requires a hasher for any type other the primitive types to be used within the map.

Here is an example:

```
struct IdHasher {
    std::size_t operator()(const Id &id) const {
        using std::size_t;
        using std::hash;
        return (hash<long long>()(id.id));
    }
};
```

To use this within your map, you need to write your map as
std::unordered_map<Id, int, IdHasher> invContainer;

#3. Improvements?

a) The use of a unique_ptr:

I read this mini article online on how unique_ptr has no memory or performance overhead, and it offers the benefit of automatically managing the lifetime of its resource without any additional cost. However, in terms of performance, std::unique_ptr is the at the same range like new and delete with optimization.

More info: Memory and Performance Overhead of Smart Pointers

b) The use of a unordered_map:
std::unordered_map<*uintptr_t*, std::deque<std::unique_ptr<Command>>> bufferedCommands;

I understand that the map stores pairs of {avatarId, commandObjectQue}, and the use of map to store these information is efficient when you want to retrieve the information later. I also know the use of unordered_map is faster than a regular map if order does not matter. My question is if you are receiving and adding commands, shouldn't the order of commands matter? If yes, then a regular map would be more efficient since the order matters.

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

a) The design of the class, Commander, was clear and concise, and the function names speak for themselves. Also, In some cases where the code was complex or variable names were confusing, your comments really helped me to understand the purpose of the code.

b)Also, 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.

Code Review #6 – Babka

---

namespace item –– Types, Format, and Style

The items do not have too much to comment on, but few alternative solutions I would recoupment are:

- Rename classes to consistent conventional class name starting with a capital letter: `BaseItem` and similar.
- Rename data members to not *shadow* the class name. Class `Item` does not need to specify that it refers to ~~`item`~~`Name` or ~~`item`~~`Desc`.
- Add parameter names to your constructors. The constructor declaration for `usableItem` is not clear as to what it needs as the only `int` argument.
- Rename ~~`equipItem`~~ to `equipableItem`. Equip is a verb and it would be a good name for a function, but not a class.
- If all classes are in a single file, place more spaces between separate classes `baseItem`, `usableItem`, `equipItem`, and `miscItem` to make it easier to spot different components.

---

class Inventory –– Naming

- Member function names all begin with check and it is unclear what exactly the function checks. Consider changing the names to explain what the functions do, such as `isOwned(…)`, `capacity()`, `size()`, and `alterSize(…)`.
- Choose argument names consistently. Change `Quantity` to `quantity` in `removeItem` declaration.
- Function `<#listItems >` displays items as a single string. Maybe `displayItems` would be a better name.

class Commander –– Responsibility and Implementation

Code has improved significantly compared to the previous version.

Although world is rarely used in the implementation, you still use it in the constructor of your class. Separate `World` and commands as suggested during the in-class review.

Try to separate your commands and heartbeat as they are not logically connected. Heartbeat is the requirement of our project, but a class that is supposed to parse and execute commands need not know about it.

Creating a command with arguments `connectionId` and `enteredCommand` does not make sense. The function maps a user to a command the user entered instead of creating a new command.

Follow previous advices and rename `UsrMgr` to a more appropriate variable name using camel-case and change `bufferedCommands` declaration to make code clearer.

Parts of code were incorrectly pasted in and the code would not compile. Creating new command is a `void` function that attempts to return:
~~`return commandMap.count(commandWord) == 0 ?`~~

Be consistent in your code. If you choose to use `ID` for items, then use `ID` (not ~~`Id`~~) for commands and users.

Code review for Babka

1. I like that your group uses return a const reference to the methods in your Item classes such as getType(), getName(), and getDesc(). However, I think you may not need to make those methods return const reference because there is a high chance that you would need to use the data from those method. Therefore, return a copy may be an option.
2. The use of std::for_each() loop makes the code very compact and clean. I would imagine if you used the regular for loop, the setUpFunctionMap() would be very messy.
3. Passing by value. In the createNewCommand() method, I see that you pass the connectionID by value. Although connectionID may be small, it is still an object. Therefore, it would be better if you pass the connectionID by const reference.
4. You make a good use of inheritance for your item. The inheritance is straight forward and there is only one layer of inheritance.
5. In the addCommandoBuffer(), there is the use of std::move() to add the command to avatarCommandQueque(). I think it is a proper way to deal with ownership of the variable where pass the ownership of a variable to another function.

```
// Will be moved to main server loop
void Commander::addCommandToBuffer(std::unique_ptr<Command> command) {
    auto connectionId = command->getCallerConnectionId();
    auto avatarCommandDeque = bufferedCommands.find(connectionId);
    if(avatarCommandDeque != bufferedCommands.end()){
        avatarCommandDeque->second.push_back(std::move(command));
    }else{
        std::deque<std::unique_ptr<Command>> newCommandDeque;
        newCommandDeque.push_back(std::move(command));
        bufferedCommands.insert({connectionId, std::move(newCommandDeque)});
    }


}
```