Spike: Task 15
Title: Composite and Component Patterns
Author: Sam Huffer, 101633177

Goals / deliverables:

- Composite Pattern:
  - Adventure (world) files that include the specification of game entities, their properties, and any nested entities (composition) they may contain.
  - Players are able to observe and modify entities contents and location, with the Commands "look in", "take from", "put in" and "open with".
- Component Pattern:
  - Game objects that receive attributes (damage, health, flammability, etc.) from component objects rather than inheritance
  - Game objects that receive actions (can be picked up, can be attacked, etc.) form component objects rather than inheritance.

Technologies, Tools, and Resources used:

- Visual Studio 2019
- Microsoft Word

Tasks undertaken:

- I copied the "Zorkish Adventure" project and the task 14 spike report into the task folder, stripping out the spike report's original content and replacing it with goals and resources pertaining to the task at hand.
- I had a look at the task instructions, lecture notes, and examples to get an idea of how the component and composite patterns work, and what could be put together for each.
- Given the notes and examples, I have already completed the following deliverables:
  - A specification of a game world with game objects, their properties, items within containers, etc. (completed for Task 12: Game Graphs From Data)
  - CommandLook, CommandTake, and CommandPut (made for Task 10: Game Data Structures, and adapted to the command pattern properly for Task 14: Command Pattern)

  That leaves the following to be completed for this spike:
  - World specification
    - Remove container item specification
    - Add component specification so that components can be added to Game Objects
  - Classes:
    - Component (each has a reference to its GameObject); derived classes:
      - Container
      - Lock (for lockable containers)
      - Movable (to register if an item can be moved)
      - Description
    - GameObject (has components, they can be added and removed); derived classes:
      - Location (has a container and a description component)
      - Player (has a container component)
      - Path (has a description component)
      - Item (has a description component, may have a movable and/or a container component)
    - CommandOpen (to open containers, using a key if they're locked)
- I put together a UML class diagram of all the classes that would need to be added or have methods or fields swapped out (fig. 1).

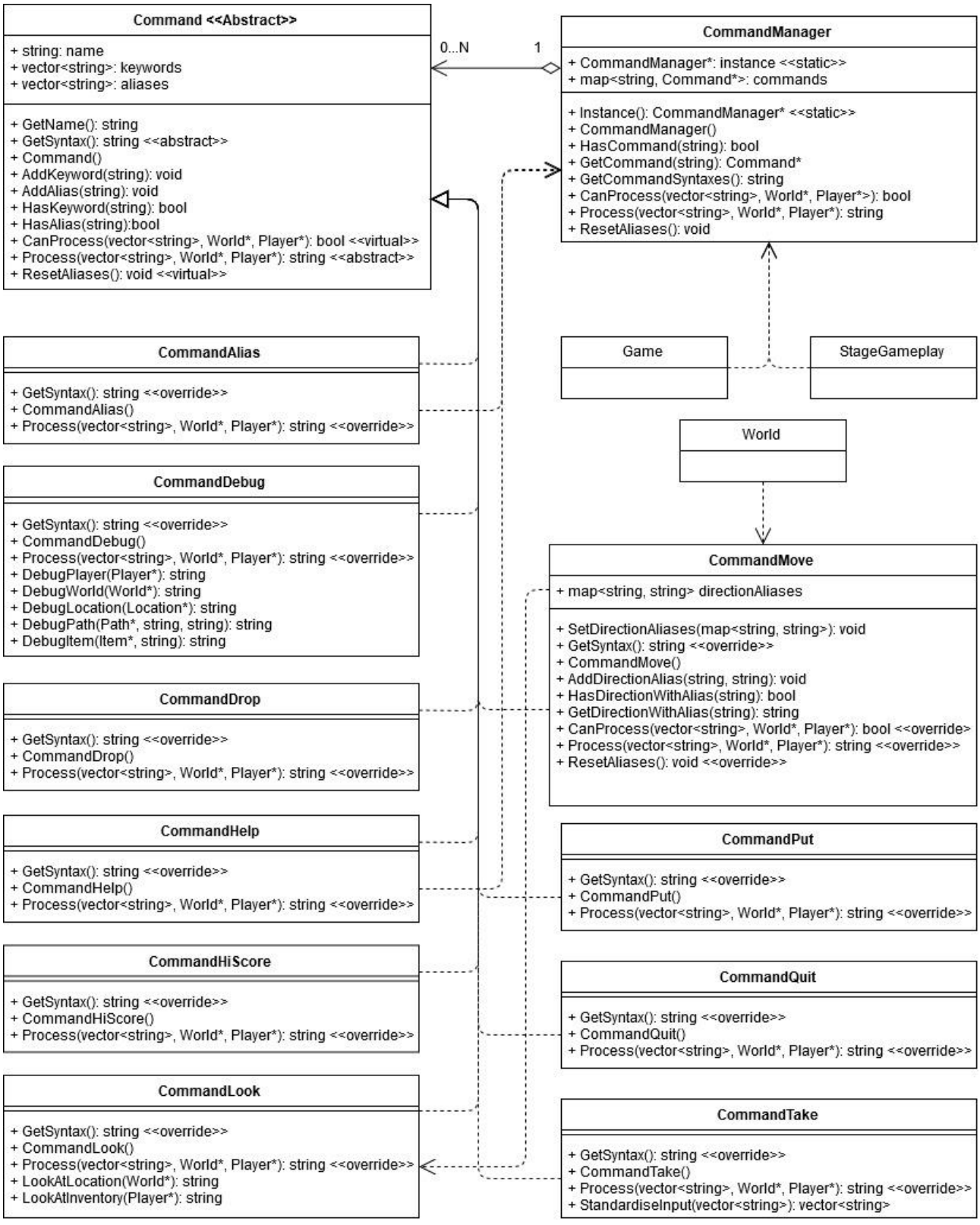

Figure 1: the initial UML diagram I put together while figuring out how to go about completing this task.

- I started implementing the component pattern with the Component base class, Container and Description Components, the GameObject base class, and Item, Player, Path and Location GameObjects, forward declaring in the Component and GameObject classes the other, and the Item class in the Container class.

```
Location::Location(std::string id, std::string name, std::string description) : GameObject(id, name)
{
    this->paths = std::map<std::string, Path*>();
    AddComponent((Component*) new Description((GameObject*)this, description));
    AddComponent((Component*) new Container((GameObject*)this, true, true));
}
```

Figure 2: Location's new constructor that works with the component pattern, as an example of the component pattern being integrated into existing classes.

```
//Public Properties----------

std::string Component::GetComponentID()
{
    return componentId;
}

GameObject* Component::GetGameObject()
{
    return gameObject;
}

//Constructor----------

Component::Component(std::string id, GameObject* gameObject)
{
    this->componentId = id;
    this->gameObject = gameObject;
}

//Methods----------
```

Figure 3: Component.cpp

```
std::string GameObject::GetID() { ... }

std::string GameObject::GetName() { ... }

//Constructor----------

GameObject::GameObject(std::string id, std::string name)
{
    this->id = id;
    this->name = name;
}

//Methods----------

bool GameObject::HasComponent(std::string component)
{
    for (std::pair<std::string, Component*> pair : components)
    {
        if (pair.first == component)
        {
            return true;
        }
    }

    return false;
}

void GameObject::AddComponent(Component* component)
{
    if (HasComponent(component->GetComponentID()))
    {
        std::cout << "GameObject " + id + " already has a " + component->GetComponentID();
    }
    else
    {
        components[component->GetComponentID()] = component;
    }
}

Component* GameObject::GetComponent(std::string component)
{
    for (std::pair<std::string, Component*> pair : components)
    {
        if (pair.first == component)
        {
            return pair.second;
        }
    }

    return nullptr;
}

void GameObject::RemoveComponent(std::string component)
{
    components.erase(component);
}
```

Figure 4: GameObject.cpp

- I went through all the Commands, finding all the errors created from the change to using Components, and fixed them all such that all the Commands used GameObjects and Components retrieved through GameObject.GetComponent() rather than members of their derived classes that no longer existed.
- I went through all the errors in World, correcting those that needed to line up with Components rather than non-existing members of classes, removing the checks for ContainerItem, as that has been removed from the project, and modifying other game object specifications and their formatting error checks to handle an updated text file formatting specification that included listing components types in-line, creating them with a ComponentFactory (e.g. fig. 5). I then went through the text file, modifying the formatting of game objects where necessary, and adding the details of what the formatting should be in a comment block at the start of "Test World.txt".
- I ran into a couple of errors where game objects were throwing bad memory allocation or access violation exceptions. Looking into those, I found that I wasn't assigning items' and locations' container components to the containers vector, but rather those game objects themselves and Visual Studio hadn't highlighted those invalid casts in-editor. I fixed those up and the program was back to running as normal.
- I implemented the Movable Component (just an empty marker Component) and added to CommandTake and CommandPut standardised move-item-from-to methods that moved Items from

```
else
{
    Item* item = new Item(splitLine[2], splitLine[3], splitLine[4]);
    containers[splitLine[1]]->AddItem(item);
    gameObjects[item->GetID()] = (GameObject*)item;

    if (splitLine[5] != "none")
    {
        std::vector<std::string> componentIds = StringManager::Instance()->StringToVector(splitLine[5], ',');

        for (std::string componentId : componentIds)
        {
            Component* component = ComponentFactory::Instance()->CreateComponent(componentId, (GameObject*)item);

            if (component == nullptr)
            {
                std::cout << "Error, \"" << filename << "\", line " << lineCount << ": Invalid component ID \"" +
                std::cout << "\tFormat: \"I:container_id:item_id:Item Name:item description:component_id_1,compone
                loadedSuccessfully = false;
            }
            else
```

Figure 5: Item's updated construction section in World.World() for once the formatting of the line its variables are being drawn from is validated.

```
std::string CommandTake::TakeFromContainer(std::vector<std::string> itemName, GameObject* containerFrom, Player* player)
{
    Item* item = ((Container*)containerFrom->GetComponent("container"))->GetItem(itemName);

    if (!item->HasComponent("movable"))
    {
        return "You cannot move " + item->GetName() + ".";
    }
    else
    {
        ((Container*)containerFrom->GetComponent("container"))->RemoveItem(itemName);
        ((Container*)player->GetComponent("container"))->AddItem(item);
        return "You added " + item->GetName() + " to your inventory.";
    }
}
```

Figure 5: CommandTake.TakeFromContainer(), which checks if an Item is Movable before putting it in a player's inventory.

one GameObject's Container Component to another's, with checks for if the Item lacked a Movable Component.

- I added to Container the isOpen and alwaysOpen fields, and appropriate public properties and constructor parameters with defaults. I then added to CommandLook, CommandTake and CommandPut checks against looking into, taking from, or putting into a Container that is closed.
- I implemented the Lock Component, having it be able to return if it's locked and unlock if presented with an acceptable key, and added to Container checks for if the lock is locked when the player tries to open it or look inside it.

```
else if (splitLine[1] == "lock")
{
    std::string lockFormat = "\tLock Format: \"C:lock:game_object_id:is_locked?:unlockable_with_item_id_1,unlockable_with_item_id_2, . . .\". If not unlockable, specify \"none\".\n\n";

    if (splitLine[3] != "Yes" && splitLine[3] != "No")
    {
        std::cout << "Error, \"" << filename << "\", line " << lineCount << ": is_locked? must be \"Yes\" or \"No\".\n";
        std::cout << generalFormat;
        std::cout << lockFormat;
        loadedSuccessfully = false;
    }
    else if (splitLine[4].size() == 0)
    {
        std::cout << "Error, \"" << filename << "\", line " << lineCount << ": You must list the items that the lock can be unlocked with, or \"none\" if not unlockable.\n";
        std::cout << generalFormat;
        std::cout << lockFormat;
        loadedSuccessfully = false;
    }
    else
    {
        std::vector<std::string> keys = StringManager::Instance()->StringToVector(splitLine[4], ',');
        bool validLock = true;

        for (std::string key : keys)
        {
            if (!gameObjects.count(key))
            {
                std::cout << "Error, \"" << filename << "\", line " << lineCount << ": No game object with ID \"" + key + "\" exists. You must create the item(s) that unlock a lock befo
                std::cout << generalFormat;
                std::cout << lockFormat;
                loadedSuccessfully = false;
                validLock = false;
            }
        }

        if (validLock)
        {
            gameObjects[splitLine[2]]->AddComponent((Component*) new Lock(gameObjects[splitLine[2]], splitLine[3] == "Yes", keys));
        }
    }
}
```

Figure 6: Example of a custom component's validation checks. Above this screenshot it checks that the component type and game object id are validly specified.

- I added to World.World() processing checks to allow the player to add custom Components of each type to game objects that they have created, and to pass any necessary parameters to that Component's constructor. I then added extra items and their custom components to the starting location to demonstrate the custom loading of components and to allow for demonstration of the Lock Component.

- I added to CommandOpen the capacity for it to check for key names and container names, and split input into the two, similar to how CommandTake and CommandPut process their respective inputs, and then if the player called “open container with key”, to check if the key will open the Container’s Lock, and open it if so.

```
else    //Player's input was "open containerName with keyName"
{
    Item* keyItem = nullptr;

    if (((Container*)world->GetCurrentLocation()->GetComponent("container"))->HasItem(keyName))
    {
        keyItem = ((Container*)world->GetCurrentLocation()->GetComponent("container"))->GetItem(keyName);
    }
    else if (((Container*)player->GetComponent("container"))->HasItem(keyName))
    {
        keyItem = ((Container*)player->GetComponent("container"))->GetItem(keyName);
    }

    if (keyItem == nullptr)
    {
        return "You cannot find \"" + StringManager::Instance()->VectorToString(keyName, ' ') + "\" at your current location or in your inventory.";
    }
    else if (!containerItem->HasComponent("lock"))
    {
        return "You don't need " + keyItem->GetName() + " to open " + containerItem->GetName() + "; it doesn't have a lock. Just open it normally.";
    }

    Lock* lock = (Lock*)containerItem->GetComponent("lock");

    if (!lock->GetIsLocked())
    {
        return containerItem->GetName() + " is already unlocked. Just open it normally.";
    }
    else if (!lock->UnlockableWith(keyItem->GetID()))
    {
        return "You try to unlock " + containerItem->GetName() + " with " + keyItem->GetName() + ", but it doesn't fit the lock.";
    }

    lock->Unlock(keyItem->GetID());

    if (lock->GetIsLocked())
    {
        return "You try to unlock " + containerItem->GetName() + " with " + keyItem->GetName() + ", but the lock stays locked.";
    }

    container->SetIsOpen(true);

    if (container->GetIsOpen())
    {
        return "You unlocked and opened " + containerItem->GetName() + ".";
    }
    else
    {
        return "You unlock " + containerItem->GetName() + " with " + keyItem->GetName() + ", but " + containerItem->GetName() + " doesn't open. It seems stuck.
    }
}
```

Figure 7: CommandOpen’s checks for opening an item with a key.

- While testing the unlocking of Containers’ Locks, I found that if one loaded a world, exited, and loaded up another world, the player retained their inventory from the first world and had it available to them in the second. I moved the instantiation of the Player object from StageGameplay’s constructor to StageGameplay.Setup() so that it would be refreshed during each playthrough.
- I added to each successful case of CommandOpen opening a Container a call to the CommandManager to run CommandLook and look inside the newly-opened container.

What we found out:

- Barring CommandOpen, all of the deliverables listed for the composite pattern I seemed to have already implemented for other tasks. ContainerItems containing Items that might also be ContainerItems seemed to perfectly fit the descriptions of the composite pattern linked to in the lecture notes.
- The lecture notes and linked examples seemed to suggest that the component pattern can be easily used with the component pattern, such as with the aforementioned case of Items that might be Containers containing more Items.
- Going through where the component pattern could be implemented for this task in Zorkish, I considered making Paths inherit from Component, but I decided against that as I figured it would be best if GameObjects only had one of any given class of Component, and Locations would often need multiple Paths, not just the one.

- Lock could be a component of Container, but it's weird for Container to only have a component field for Locks, so it makes more sense for Lock to be a component of an Item, and Container checks if its GameObject (i.e. Item) has a Lock component. Doing it this way requires Components to have a reference to their GameObjects, the same as how Unity has the public variable "[some component].gameObject".
- I tried using the factory pattern to create the Components that Items were specified as having beyond the default Description Component, and I found that unless I wanted to add to ComponentFactory several overloaded CreateComponent() methods to handle different parameter configurations, I could only have it create Components that had either no extra parameters beyond those necessary for all Components (e.g. Movable), or Components that had parameters that one could safely assign default values for their parameters (e.g. Container). Anything that would require case-by-case parameters would be more unwieldy to list as part of an Item's specifications anyway, as the way I have the text file specification set up, if a Component is listed as part of an Item's specifications, it's only specified by it's Component type, not with any parameters. As such, any components that required case-by-case parameters (e.g. Lock, which requires a list of keys that can unlock it) I had to list in their own specification line at any point after the Item (or GameObject generally) that they were to be added to.
- Apparently C++ doesn't like enums values and classes that have the same names.