Spike: Task 18
Title: Message Systems
Author: Sam Huffer, 101633177

Goals / deliverables:

- A messaging system to facilitate communication between game objects in Zorkish Adventures, including:
  - A design for the message system, expressed as a class/module/sequence diagram(s), including a clear description of your message details, submitted as a PDF doc with all figures and diagrams created.
  - A working demonstration within Zorkish Adventures that shows how a message can be used to change a game entity.
- The implementation for this spike should NOT include:
  - A message system for the command processer
  - Broadcasting, filtering, or schedule/delay behaviour

Technologies, Tools, and Resources used:

- Visual Studio 2019
- Microsoft Word
- Draw.io

Tasks undertaken:

- I copied the "Zorkish Adventure" project and the task 16 extension 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.

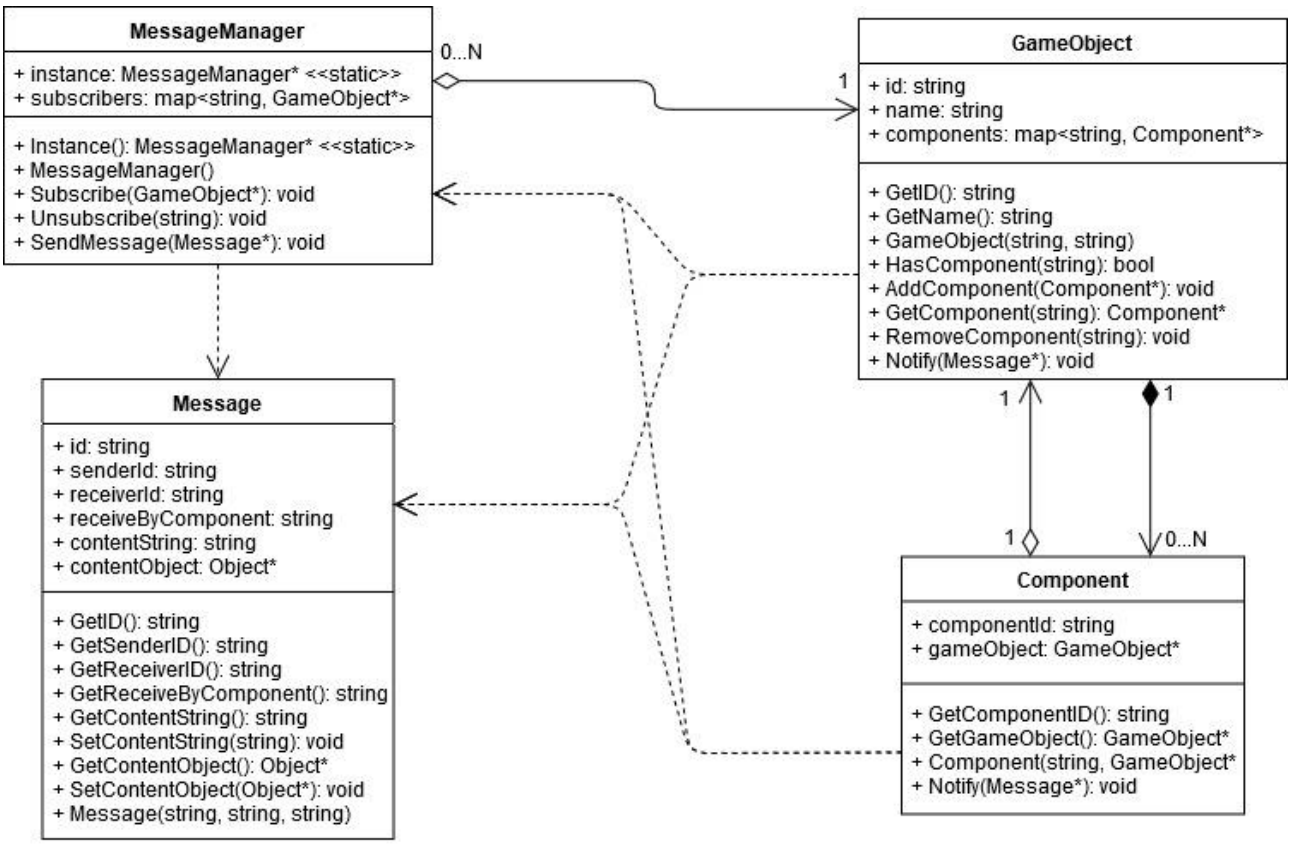

Figure 1: the initial UML class diagram for my dispatcher-style messaging system.

- I had a look at my lecture notes on design patterns for messaging systems, and found the observer pattern. I took that and put together a UML class diagram for a dispatcher-style messaging system (fig. 1).
- I implemented the MessageManager (fig. 2) and Message classes, Message pretty much being a collection of fields with public properties for getting them. The contentString and contentObject fields I condensed down to just Message.content, which I made of type void*.
- I went through World.World(), and any time a GameObject would be added to the map gameObjects, I also had it subscribe to the MessageManager. I also added to MessageManager an UnsubscribeAll() method which clears the list of subscribers, and modified Game.SetNextStage() to call it so that objects from a no-longer extant World object would not be stored in MessageManager.subscribers when a new World is constructed.
- When I made that change, I noticed that I had forgotten to reset which Commands are unlocked when returning from StageGameplay to StageMainMenu, so I added to CommandManager a LockCommands() method that adds all Commands except help, look and quit to CommandManager.unavailableCommands, and removed any Command in unavailableCommands from availableCommands.
- I added to GameObject and Component the virtual method Notify() to allow them to receive messages. I updated it to return Message*'s to the sending object, which wasn't specified in the original UML. By default, I had it return a nullptr, as Messages to a base GameObject or Component are meaningless. GameObject's also checks if received messages should instead be passed to a component rather than the game object itself.
- I reworked CommandOpen to open containers using Messages rather than directly accessing Container's public properties, and modified the Container and Lock Components to open and unlock using Messages, returning to CommandOpen the results of trying to open or unlock the Container.

```
Message* Container::Notify(Message* message)
{
    if (message->GetSenderID() == "OPEN")
    {
        std::vector<std::string> messageContent = *(std::vector<std::string>*)message->GetContent();

        if (messageContent[0] == "open")
        {
            if (gameObject->HasComponent("lock"))
            {
                if (!((Lock*)gameObject->GetComponent("lock"))->GetIsLocked())
                {
                    isOpen = true;
                    return new Message(gameObject->GetID(), "container", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("already unlocked"));
                }
                else if (messageContent.size() == 1)
                {
                    return new Message(gameObject->GetID(), "container", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("locked"));
                }
                else
                {
                    Message* msg = new Message(gameObject->GetID(), "container", gameObject->GetID(), "lock", (void*) new std::vector<std::string>({ "unlock", messageContent[1] } ));
                    Message* result = MessageManager::Instance()->SendMessage(msg);
                    std::string messageContent = *(std::string*)result->GetContent();

                    if (messageContent == "unlocked")
                    {
                        isOpen = true;
                    }

                    return new Message(gameObject->GetID(), "container", message->GetSenderID(), message->GetSenderType(), result->GetContent());
                }
            }
            else
            {
                if (messageContent.size() > 1)
                {
                    return new Message(gameObject->GetID(), "container", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("no lock"));
                }
                else if (isOpen)
                {
                    return new Message(gameObject->GetID(), "container", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("already open"));
                }
                else
                {
                    isOpen = true;
                    return new Message(gameObject->GetID(), "container", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("opened"));
                }
            }
        }
    }

    return nullptr;
}
```

Figure 2: Container.Notify(), which now does the checks to see if a container is already open, locked, gets opened, etc. without CommandOpen having to call its methods.

```
Message* Lock::Notify(Message* message)
{
    if (message->GetSenderID() == gameObject->GetID())
    {
        std::vector<std::string> messageContent = *(std::vector<std::string>*)message->GetContent();

        if (messageContent.size() == 2 && messageContent[0] == "unlock")
        {
            if (UnlockableWith(messageContent[1]))
            {
                Unlock(messageContent[1]);
                return new Message(gameObject->GetID(), "lock", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("unlocked"));
            }
            else
            {
                return new Message(gameObject->GetID(), "lock", message->GetSenderID(), message->GetSenderType(), (void*) new std::string("can't unlock"));
            }
        }
    }

    return nullptr;
}
```

Figure 3: Lock.Notify(), which does the checks to see if a lock can be unlocked by the key with encapsulated id.

```
std::string CommandOpen::Process(std::vector<std::string> input, World* world, Player* player)
{
    input.erase(input.begin());

    std::vector<std::string> messageContent = std::vector<std::string>({ "open" });
    Item* containerItem = nullptr;
    Item* keyItem = nullptr;
    std::vector<std::string> containerNameVector = std::vector<std::string>();
    std::vector<std::string> keyNameVector = std::vector<std::string>();
    std::string containerName;
    std::string keyName;

    if (!StringManager::Instance()->VectorContainsString(input, "with")) { ... }
    else { ... }

    //Validate container exists and is a container
    if (((Container*)world->GetCurrentLocation()->GetComponent("container"))->HasItem(containerNameVector)) { ..
    else if (((Container*)player->GetComponent("container"))->HasItem(containerNameVector)) { ... }

    if (containerItem == nullptr)
    {
        return "You cannot find \"" + StringManager::Instance()->VectorToString(containerNameVector, ' ') + "\"
    }
    else
    {
        containerName = containerItem->GetName();

        if (!containerItem->HasComponent("container"))
        {
            return containerName + " is not a container; you cannot open it.";
        }
    }

    //Send Message
    Message* msg = new Message("OPEN", "command", containerItem->GetID(), "container", (void*)& messageContent);
    Message* result = MessageManager::Instance()->SendMessage(msg);
    std::string output = *(std::string*)result->GetContent();

    //Handle results
    if (output == "already unlocked")
    {
        return containerName + " is already unlocked. Just open it normally.";
    }
    else if (output == "locked") { ... }
    else if (output == "can't unlock") { ... }
    else if (output == "unlocked") { ... }
    else if (output == "no lock") { ... }
    else if (output == "already open") { ... }
    else if (output == "opened") { ... }
    else { ... }
}
```

Figure 4: The updated CommandOpen.Process() method. It instantiates required variables, checks if it needs to split the input string, assigning the appropriate parts to vectors of strings as seen in previous tasks, checking if the key exists if specified, appending its ID to messageContent if it does. It then checks the Container exits and is a Container, before sending the Message to the Container Item, specifying it needs to be passed to its Container Component. MessageManager.SendMessage() passes the message to the specified GameObject, whose Container Component's Notify() method processes the message, and returns a result, which is then checked to return appropriate output.

What we found out:

- The observer design pattern is well suited to the message dispatcher implementation of a messaging system.
- When trying to implement Message's object* members (contentObject and GetContentObject()), I found that the type object didn't exist for C++, so I had to find another way to store an object of any type in the Message. I came across both the types void*, which can be a pointer to an object or value of any type, and std::any, which can store single values of any type. I initially elected to use std::any, as it's documentation indicated it had in-built safeguards for casting std::any values to other types. However, I ran into issues trying to cast std::any back to a string, so I swapped std::any references out for void* references instead. That removed the errors while allowing valid casts from particular pointer types to and from void*.
- To make full use of messaging to pass information and trigger behaviour rather than relying on classes' inbuilt methods, it's useful if Messages can be directly replied to, to pass output or results back to the object that initiated the messaging.
- Messages should have the id and type not just for the recipient of the Message, but also the sender, so that reply Messages can easily be sent to the original sender as necessary to query information or trigger particular behaviours.
- Particularly given the example used listed in the spike instruction document, Messages seem like a good way to prompt game entities to change states from one thing to another and have them perform the checks on the input such that the original sender of the message, probably a Command for Zorkish, does not have to directly access the game entity's members to get information from it. Commands HiScore and Quit don't operate on GameObjects, so would have no use for Messages unless Stage changing was reworked to use Messages, which would be overkill for this spike. Moving Items between Containers seems doable with the way I've implemented Messages, but unnecessary for this spike and possibly more cumbersome to implement than Commands directly operating on the Player, current Location, and their Items. Commands where you're basically just requesting information, such as CommandLook, would be quite feasible for using Messages with minimal cumbersomeness, but again seem more than is necessary to demonstrate working Messages. In both of these cases, as somewhat with opening Containers, the Commands (at this stage at least) would still need to know about and be able to access Items and their Components directly to perform some checks. That said, if MessageManager is configured to do the same checks and ensure players can't access Items or Locations outside their current Location (except for moving), then such checks could be handled by MessageManager, reducing Item/Component-Command coupling.
- I did not implement this, but messages do seem like they would be a good way for commands to communicate. Certainly it would allow for cleaning up of parameters for Commands' CanProcess() and Process() methods, since each requires the input string vector, World pointer and Player*, but they don't all need to use them.

Task 18 – Message Systems – Design Diagram

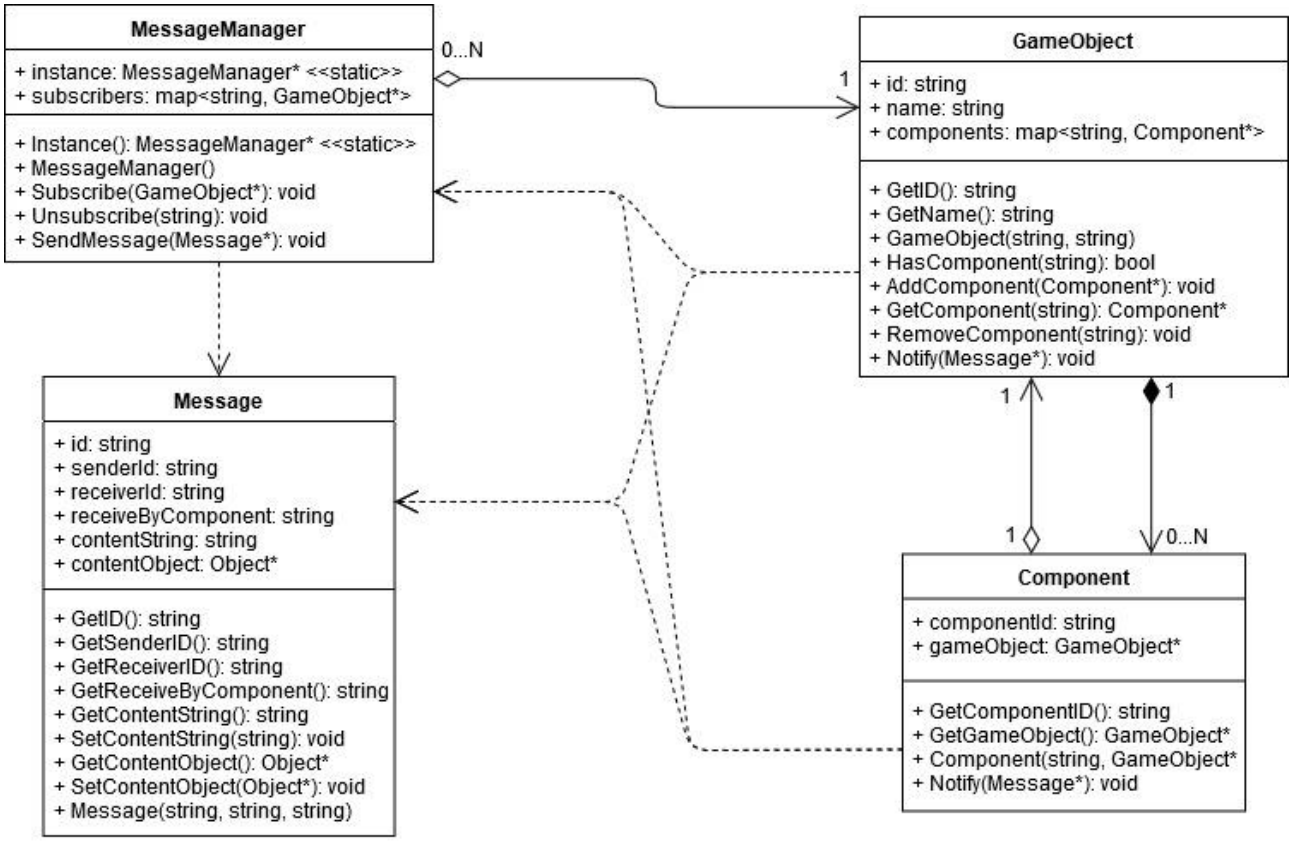

Figure 1: The initial UML class diagram for my dispatcher-style messaging system. Included down here again because the task instructions ask for a document with all diagrams/figures apparently separate from the spike report, but Doubtfire only accepts one document for this task, so it's all just going to be appended here at the end even though all these diagrams are up in the spike report as evidence screenshots anyway because why not.