Extension: Task 19
Title: Messaging Extended
Author: Sam Huffer, 101633177

Goals / deliverables:

- Extend the previous spike to include one or more of the following:
  - Broadcast messages (specified by the sender)
  - Filtering of messages before delivery / pickup (by the blackboard / dispatch system, not the sender),
  - Scheduling of messages for the future
- You need to produce:
  - Updated design documents as applicable, clearly showing what you have had to add to support your additional features
  - Updated working code demonstration within Zorkish
- Notes:
  - You may like to include the ability for senders to cancel messages to support the above features.
  - Message filtering could be based on game entity values/types, or locations.
  - A message system is often a key part of any combat system. This might be a good target for you.

Technologies, Tools, and Resources used:

- Visual Studio 2019
- Microsoft Word
- Draw.io

Tasks undertaken:

- I copied the "Zorkish Adventure" project and the task 18 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 through the task instructions and considered what I could build to demonstrate the suggested features. I put together a UML class diagram for the required classes that would need to be added or changed, and planned what order I would tackle required changes in.
- I removed MessageManager's existing one-size-fits-all Subscribe(), Unsubscribe() and subscribers members, and replaced them with members suited for storing and handling

```
void MessageManager::Subscribe(Player* player){ ... }

void MessageManager::Subscribe(World* world){ ... }

void MessageManager::Subscribe(Command* command)
{
    if (subscribedCommands.count(command->GetName()))
    {
        std::cout << "Warning: a command with name \"" + command->GetName() + "\" is already subscribed to MessageManager.
    }
    else
    {
        subscribedCommands[command->GetName()] = command;
    }
}

void MessageManager::Subscribe(Location* location)
{
    if (subscribedLocations.count(location->GetID()))
    {
        std::cout << "Warning: a location with ID \"" + location->GetID() + "\" is already subscribed to MessageManager.\n
    }
    else
    {
        subscribedLocations[location->GetID()] = location;
    }
}

void MessageManager::Subscribe(std::string location, Path* path){ ... }

void MessageManager::Subscribe(std::string container, Item* item){ ... }

void MessageManager::Unsubscribe(std::string type){ ... }

void MessageManager::Unsubscribe(std::string type, std::string subscriberId){ ... }

void MessageManager::Unsubscribe(std::string type, std::string subscriberId, std::string containerId)
{
    if (type == "path")
    {
        if (subscribedPathsInLocation.count(containerId) && subscribedPathsInLocation[containerId].count(subscriberId))
        {
            subscribedPathsInLocation[containerId].erase(subscriberId);
        }
    }
    else if (type == "item")
    {
        if (subscribedItemsInContainer.count(containerId) && subscribedItemsInContainer[containerId].count(subscriberId))
        {
            subscribedItemsInContainer[containerId].erase(subscriberId);
        }
    }
    else
    {
        std::cout << "Warning: can't unsubscribe subscriber with ID \"" + subscriberId + "\". Invalid subscriber type \""
    }
}
```

Figure 1: MessageManager's Subscribe() and Unsubscribe() methods, with some open to convey how those ones specifically work, and give a general sense of how different GameObject types are handled.

```
Message* MessageManager::SendMessage(Message* message)
{
    switch (receiverTypes[message->GetReceiverType()])
    {
        case ReceiverPlayer:
            if (subscribedPlayer != nullptr)
            {
                return subscribedPlayer->Notify(message);
            }

            return nullptr;
        /* ... */
        /* ... */
        case ReceiverLocation:
            if (subscribedLocations.count(message->GetReceiverID()))
            {
                return subscribedLocations[message->GetReceiverID()]->Notify(message);
            }

            return nullptr;
        default:
            return nullptr;
    }
}

Message* MessageManager::SendMessage(Message* message, std::string container)
{
    switch (receiverTypes[message->GetReceiverType()])
    {
        case ReceiverPath:
            if (subscribedPathsInLocation.count(container) && subscribedPathsInLocation[container].count(message->GetReceiverID()))
            {
                return subscribedPathsInLocation[container][message->GetReceiverID()]->Notify(message);
            }

            return nullptr;
        case ReceiverGameObject:
        case ReceiverItem:
        case ReceiverButton:
        case ReceiverContainer:
        case ReceiverDescription:
        case ReceiverFlammable:
        case ReceiverLandmine:
        case ReceiverLock:
        case ReceiverMovable:
            if (subscribedItemsInContainer.count(container) && subscribedItemsInContainer[container].count(message->GetReceiverID()))
            {
                return subscribedItemsInContainer[container][message->GetReceiverID()]->Notify(message);
            }

            return nullptr;
        default:
            return nullptr;
    }
}
```

Figure 2: MessageManager.SendMessage()'s overloads and how they handled their respective GameObject types.

Players, Worlds, Commands, Locations, Paths and Items separately (fig. 1). I then updated SendMessage() to distinguish between types of Message recipients through an enum-using switch statement, and pass the message to objects in the appropriate list (fig. 2).

- I went through World.World() and found all now erroneous calls to MessageManager.Subscribe(), and updated them on a type-by-type basis to use the correct, overloaded Subscribe() method for that type. I removed the

```
#include "Notifiable.h"

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

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

Notifiable::Notifiable()
{
}

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

Message* Notifiable::Notify(Message* message)
{
    return nullptr;
}
```

Figure 3: Notifiable.cpp. Notifiable only has its constructor and the virtual method Notify() as members.

```
void Container::AddItem(Item* item)
{
    items.push_back(item);
    MessageManager::Instance()->Subscribe(gameObject->GetID(), item);
}

void Container::RemoveItem(std::vector<std::string> input)
{
    if (!items.empty())
    {
        std::string string = StringManager::Instance()->VectorToString(input, ' ');

        for (int i = 0; i < (int)items.size(); i++)
        {
            if (StringManager::Instance()->ToLowercase(items[i]->GetName()) == string)
            {
                MessageManager::Instance()->Unsubscribe("item", items[i]->GetID(), gameObject->GetID());
                items.erase(items.begin() + i);
                break;
            }
        }
    }
}
```

Figure 4: Container's updated AddItem() and RemoveItem() methods.

Subscribe() call for items, and added Subscribe() and Unsubscribe() calls to Container.AddItem() and RemoveItem() respectively to ensure Items are subscribed under the correct container (fig. 3).

- I went through each class that had a Notify() method and made it or its parent class inherit from Notifiable, with Notify() being a virtual method of Notifiable that just returns a nullptr unless overridden (fig. 4).
- I added to GameObject a field containerId to store the ID of the Container Item or Location holding a GameObject, and added to Container.AddItem(), Container.RemoveItem() and Location.AddPath() calls to public properties to set the GameObject's containerId to be the GameObject.id of the containing GameObject. While doing so, I also updated Location.AddPath() to subscribe the added path to the MessageManager under the Location's ID, and removed the call for that in World.World().
- I updated Message's constructor to request the ID and type of the Message's sender and recipient Notifiables, storing them in appropriate fields and accessible with appropriate public properties. Then, I updated all calls to Message's constructor to fit the new parameters (fig. 5), before combining MessageManager's overloaded SendMessage() methods, with the cases that would have used the passed string parameter now calling message->GetReceiverParentID() instead (fig. 6). Next, I updated all calls to SendMessage() to not pass any string parameters.

```
Message* msg = new Message(
    gameObject->GetID(), "container",
    gameObject->GetParentID(), gameObject->GetParentType(),
    gameObject->GetID(), "lock",
    gameObject->GetParentID(), gameObject->GetParentType(),
    (void*) new std::vector<std::string>({ "unlock", messageContent[1] })
);
```

Figure 5: a new message in Container.cpp using the new parameter setup.

- I made MessageManager inherit from Notifiable and implemented for it a Notify() method that could handle queries for if an Item is accessible to the player from their current location, which is specified in the query Message so that MessageManager only checks with the Player and current Location, effectively filtering out all other Locations or Containers the Item could be in. All Containers and Locations need be subscribed to MessageManager for this to work, as MessageManager needs a

```
case ReceiverPath:
    if (subscribedPathsInLocation.count(message->GetReceiverParentID()) && subscribedPathsInLocation[message->GetReceiverParentID()].count(message->GetReceiverID()))
    {
        return subscribedPathsInLocation[message->GetReceiverParentID()][message->GetReceiverID()]->Notify(message);
    }

    return nullptr;
```

Figure 6: MessageManager.SendMessage()'s updated case ReceiverPath, using the Message.GetReceiverParentID() public property.

```
Message* MessageManager::Notify(Message* message)
{
    if (message->GetSenderType() == "command")
    {
        std::vector<std::string> messageContent = *(std::vector<std::string>*) message->GetContent();

        if (messageContent[0] == "access item from location")
        {
            std::vector<std::string> itemNameVector = StringManager::Instance()->StringToVector(messageContent[1], ' ');
            std::string playerLocation = messageContent[2];
            std::string itemID = "null";
            std::string itemName = "null";
            std::string parentID = "null";
            std::string parentType = "null";

            if (((Container*)subscribedPlayer->GetComponent("container"))->HasItem(itemNameVector))
            {
                itemID = ((Container*)subscribedPlayer->GetComponent("container"))->GetItem(itemNameVector)->GetID();
                itemName = ((Container*)subscribedPlayer->GetComponent("container"))->GetItem(itemNameVector)->GetName();
                parentID = subscribedPlayer->GetID();
                parentType = "player";
            }
            else if (subscribedLocations.count(playerLocation) && ((Container*)subscribedLocations[playerLocation]->GetComponent("container"))->HasItem(itemNameVector))
            {
                itemID = ((Container*)subscribedLocations[playerLocation]->GetComponent("container"))->GetItem(itemNameVector)->GetID();
                itemName = ((Container*)subscribedLocations[playerLocation]->GetComponent("container"))->GetItem(itemNameVector)->GetName();
                parentID = playerLocation;
                parentType = "location";
            }

            return new Message(
                "MessageManager", "MessageManager",
                "null", "null",
                message->GetSenderID(), message->GetSenderType(),
                message->GetSenderParentID(), message->GetSenderParentType(),
                (void*) new std::vector<std::string>({ itemID, itemName, parentID, parentType })
            );
        }
    }

    return nullptr;
}
```

Figure 7: MessageManager.Notify()

```
Message* GameObject::Notify(Message* message)
{
    std::string receiverType = message->GetReceiverType();

    if (receiverType == "gameObject")
    {
        std::vector<std::string> messageContent = *(std::vector<std::string>*) message->GetContent();

        if (messageContent[0] == "has component")
        {
            return new Message(
                id, type,
                parentId, parentType,
                message->GetSenderID(), message->GetSenderType(),
                message->GetSenderParentID(), message->GetSenderParentType(),
                (void*) new std::string(HasComponent(messageContent[1]) ? "Yes" : "No")
            );
        }
    }
    else if (HasComponent(receiverType)) { ... }

    return nullptr;
}
```

Figure 8: GameObject.Notify() handling queries about the Components it has.

```
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") { ... }
        else if (messageContent[0] == "is locked")
        {
            return new Message(
                gameObject->GetID(), gameObject->GetType(),
                gameObject->GetParentID(), gameObject->GetParentType(),
                message->GetSenderID(), message->GetSenderType(),
                message->GetSenderParentID(), message->GetSenderParentType(),
                (void*) new std::string(isLocked ? "locked" : "unlocked")
            );
        }
    }

    return nullptr;
}
```

Figure 9: Lock.Notify() handling queries about its status.

reference to them to access their public members, or to call their Notify() method if I were to change the query handling to use Messages.

- I updated GameObject.Notify to be able to take queries for if it has a particular Components (fig. 8), and the Container and Lock Components' Notify methods to be able to take queries regarding their current state (fig. 9, fig. 10).

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

        if (messageContent[0] == "open") { ... }
        else if (messageContent[0] == "is open")
        {
            std::string result;

            if (gameObject->HasComponent("lock"))
            {
                Message* msgIsLocked = new Message(
                    gameObject->GetID(), "container",
                    gameObject->GetParentID(), gameObject->GetParentType(),
                    gameObject->GetID(), "lock",
                    gameObject->GetParentID(), gameObject->GetParentType(),
                    (void*) new std::vector<std::string>({ "is locked" })
                );
                Message* resultIsLocked = MessageManager::Instance()->SendMessage(msgIsLocked);
                std::string messageContent = *(std::string*)resultIsLocked->GetContent();

                result =
                    (messageContent == "locked") ? "locked"
                    : isOpen ? "open"
                    : "closed";
            }
            else
            {
                result = isOpen ? "open" : "closed";
            }

            return new Message(
                gameObject->GetID(), gameObject->GetType(),
                gameObject->GetParentID(), gameObject->GetParentType(),
                message->GetSenderID(), message->GetSenderType(),
                message->GetSenderParentID(), message->GetSenderParentType(),
                (void*) new std::string(result)
            );
        }
    }

    return nullptr;
}
```

Figure 10: Container.Notify() handling queries about its status.

- I updated CommandOpen.Process() to operate entirely using messages rather than directly accessing Items (fig. 11, fig. 12), given that it only requests information and changes states within Items' Components, rather than moving Items about like Commands Take, Put or Drop.
- I created skeletons for the Button, Landmine and Flammable Component classes and began fleshing out the Flammable Component so that when it got a Message to ignite, it would destroy its game object (fig. 13). To enable it to reply with a "burst into flames" message and then be destroyed, I added to MessageManager methods for queueing Messages and for triggering the sending of queued Messages with a particular tag (fig. 14). I then added to MessageManager.SendMessage() a check for if a Message is overriding filtering, necessitated

```
//Validate container exists via messages
Message* msgAccessContainerItemFromLocation = new Message(
    "OPEN", "command",
    "null", "null",
    "messageManager", "messageManager",
    "null", "null",
    (void*) new std::vector<std::string>({ "access item from location", StringManager::Instance()->VectorToString(containerNameVector, ' '), world->GetCurrentLocation()->GetID() })
);
Message* resultAccessContainerItemFromLocation = MessageManager::Instance()->SendMessage(msgAccessContainerItemFromLocation);
std::vector<std::string> outputVector = *(std::vector<std::string>*)resultAccessContainerItemFromLocation->GetContent();

if (outputVector[0] == "null")
{
    return "You cannot find \"" + StringManager::Instance()->VectorToString(containerNameVector, ' ') + "\" at your current location or in your inventory.";
}

containerID = outputVector[0];
containerName = outputVector[1];
containerParentID = outputVector[2];
containerParentType = outputVector[3];

//Validate container is actually a container via messages
Message* msgHasContainerComponent = new Message(
    "OPEN", "command",
    "null", "null",
    containerID, "gameObject",
    containerParentID, containerParentType,
    (void*) new std::vector<std::string>({ "has component", "container" })
);
Message* resultHasContainerComponent = MessageManager::Instance()->SendMessage(msgHasContainerComponent);

if (*(std::string*) resultHasContainerComponent->GetContent() == "No")
{
    return containerName + " is not a container; you cannot open it.";
}
```

Figure 11: CommandOpen.Process() querying if the Player can access an Item at their current Location, and if it's a Container, via messages. If Containers require a key to open, the key's availability is queried in the same manner.

```
//Open / unlock container via messages
Message* msgOpenContainerItem = new Message(
    "OPEN", "command",
    "null", "null",
    containerID, "container",
    containerParentID, containerParentType,
    (void*)& messageContent);
Message* resultOpenContainerItem = MessageManager::Instance()->SendMessage(msgOpenContainerItem);
std::string output = *(std::string*)resultOpenContainerItem->GetContent();

//Handle results
if (output == "already unlocked") { ... }
else if (output == "locked") { ... }
else if (output == "can't unlock") { ... }
else if (output == "unlocked")
{
    //Verify via messages that the container was opened / unlocked
    Message* msgContainerOpen = new Message(
        "OPEN", "command",
        "null", "null",
        containerID, "container",
        containerParentID, containerParentType,
        (void*) new std::vector<std::string>({ "is open" }));
    Message* resultContainerOpen = MessageManager::Instance()->SendMessage(msgContainerOpen);
    std::string outputContainerOpen = *(std::string*)resultContainerOpen->GetContent();

    if (outputContainerOpen == "open")
    {
        return "You unlocked and opened " + containerName + ". " + CommandManager::Instance()->Process(StringManager::Instance()->StringToVector("look in " + StringManager::Instance()->ToLowercase(containerName), ' '), world, player);
    }
    else if (outputContainerOpen == "locked")
    {
        return "You try to unlock " + containerName + " with " + keyName + ", but the lock stays locked. It seems stuck. It shouldn't be. This is not right at all.";
    }
    else
    {
        return "You unlock " + containerName + " with " + keyName + ", but " + containerName + " doesn't open. It seems stuck. It shouldn't be. This is not right at all.";
    }
}
```

Figure 12: CommandOpen.Process() opening Containers, processing the results, and verifying if Containers have been opened, via messages.

```
Message* Flammable::Notify(Message* message)
{
    if (!ignited)
    {
        std::string messageContent = *(std::string*)message->GetContent();

        if (messageContent == "ignite")
        {
            ignited = true;

            Message* msgRemove = new Message(
                gameObject->GetID(), "flammable",
                gameObject->GetParentID(), gameObject->GetParentType(),
                gameObject->GetParentID(), gameObject->GetParentType(),
                "override", "any",
                (void*) new std::vector<void*>({ (void*)new std::string("remove"), (void*)new int(0) , (void*)
            );

            MessageManager::Instance()->QueueMessage(msgRemove);

            return new Message(
                gameObject->GetID(), "flammable",
                gameObject->GetParentID(), gameObject->GetParentType(),
                message->GetSenderID(), message->GetSenderType(),
                message->GetSenderParentID(), message->GetSenderParentType(),
                (void*) new std::string(gameObject->GetName() + " bursts into flames and is destroyed.")
            );
        }
    }

    return nullptr;
}
```

Figure 13: Flammable.Notify(), which queues a Message for Flammable.gameObject's container to remove Flammable.gameObject.

by Items wanting to remove themselves from their containing Container Component for which the Item wouldn't know if it was attached to an Item that was also inside a Container or which Container that was (fig. 15).

- I implemented Button and Landmine's Notify() methods, adding handling for trigger Messages that would broadcast to each triggerable GameObject at the target Location a trigger message (fig, 17). Then I added to MessageManager.SendMessage()'s case for handling non-overridden Items a check for if a Message's receiverID is "all", looping through all Items subscribed under a particular Location's ID and sending them the sent Message, compiling any non-nullptr responses into a vector of Message's and returning that to the messenger that sent the broadcast Message.
- I updated the world text file specification to include Buttons, Landmines, Flammable

```
void MessageManager::QueueMessage(Message* message)
{
    queuedMessages.push_back(message);
}

void MessageManager::SendQueuedMessages(std::string tag)
{
    std::vector<Message*> stillQueuedMessages = std::vector<Message*>();
    std::vector<void*> messageContent;
    std::string messageTag;
    int* messageDelayCount;

    for (Message* message : queuedMessages)
    {
        messageContent = *(std::vector<void*>*)message->GetContent();
        messageTag = *(std::string*)messageContent[0];

        if (messageTag == tag)
        {
            messageDelayCount = (int*)messageContent[1];

            if (*messageDelayCount == 0)
            {
                SendMessage(message);
            }
            else
            {
                *messageDelayCount -= 1;
                stillQueuedMessages.push_back(message);
            }
        }
    }

    queuedMessages = stillQueuedMessages;
}
```

Figure 14: MessageManager.QueueMessage() and MessageManager.SendQueuedMessages().

```
Message* MessageManager::SendMessage(Message* message)
{
    if (message->GetReceiverParentID() == "override" && message->GetReceiverParentType() == "any")
    {
        switch (receiverTypes[message->GetReceiverType()])
        {
            /* ... */
            case ReceiverLocation:
                if (subscribedLocations.count(message->GetReceiverID())) { ... }

                return nullptr;
            case ReceiverPath:
                for (std::pair<std::string, std::map<std::string, Path*>> pair : subscribedPathsInLocation)
                {
                    if (subscribedPathsInLocation[pair.first].count(message->GetReceiverID()))
                    {
                        return subscribedPathsInLocation[pair.first][message->GetReceiverID()]->Notify(message);
                    }
                }

                return nullptr;
```

Figure 15: MessageManager.SendMessage()'s new checks for overriding filtering. Locations are handled normally, but Paths and Items cycle through their containing Locations / Containers for the one that contains them, rather than checking if a specified one has them.

```
#Dungeon 6: use button to detonate landmines and burn down trees
L:dungeon_6:Dungeon 6: a cold, dark dungeon with a 6 painted on the wall.
C:unlock_commands:dungeon_6:use
I:dungeon_6:red_button:Red Button:A big, red button mounted on the wall.:none
C:button:red_button:landmine:dungeon_6
I:dungeon_6:landmine_a:Landmine A:A landmine affixed to the ground.:landmine
I:dungeon_6:bush:Bush:A short, squat bush.:flammable
I:dungeon_6:tree:Tree:A nice, shady tree.:flammable
```

Figure 16: a section added to "Dungeon World.txt" to allow for demonstrating Button, Landmine and Flammable Components (and thereby demonstrating their addition to a world), and a new use Command.

```
Message* Landmine::Notify(Message* message)
{
    if (!detonated)
    {
        std::string messageContent = *(std::string*)message->GetContent();
        std::vector<std::string> flammableReplies = std::vector<std::string>();
        flammableReplies.push_back(std::string("A landmine EXPLODES!"));

        if (messageContent == "trigger")
        {
            detonated = true;

            Message* msgRemove = new Message(
                gameObject->GetID(), "landmine",
                gameObject->GetParentID(), gameObject->GetParentType(),
                gameObject->GetParentID(), gameObject->GetParentType(),
                "override", "any",
                (void*) new std::vector<void*>({ (void*)new std::string("remove"), (void*)new ir
            );
            MessageManager::Instance()->QueueMessage(msgRemove);

            Message* msgIgnite = new Message(
                gameObject->GetID(), "landmine",
                gameObject->GetParentID(), gameObject->GetParentType(),
                "all", "flammable",
                gameObject->GetParentID(), gameObject->GetParentType(),
                (void*) new std::string("ignite")
            );
            Message* replyIgnite = MessageManager::Instance()->SendMessage(msgIgnite);
            std::vector<Message*>* replies = (std::vector<Message*>*)replyIgnite->GetContent();

            for (Message* reply : *replies)
            {
                if (reply != nullptr)
                {
                    flammableReplies.push_back(*(std::string*)reply->GetContent());
                }
            }

            return new Message(
                gameObject->GetID(), "landmine",
                gameObject->GetParentID(), gameObject->GetParentType(),
                message->GetSenderID(), message->GetSenderType(),
                message->GetSenderParentID(), message->GetSenderParentType(),
                (void*) new std::vector<std::string>(flammableReplies)
            );
        }
    }

    return nullptr;
}
```

Figure 17: Landmine.Notify(), which, when triggered, broadcasts to all Flammable GameObjects in its GameObject's parent that they should all "ignite".

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

    std::string usableName = StringManager::Instance()->VectorToString(input, ' ');
    Item* usableItem = nullptr;

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

    if (usableItem == nullptr)
    {
        return "You cannot find \"" + usableName + "\" at your current location or in your inventory.";
    }
    else if (!usableItem->HasComponent("button"))
    {
        return "You cannot use " + usableItem->GetName() + ".";
    }
    else
    {
        std::string result = "";
        Message* msgUse = new Message(
            "USE", "command",
            "null", "null",
            usableItem->GetID(), "button",
            usableItem->GetParentID(), usableItem->GetParentType(),
            (void*) new std::string("use")
        );
        Message* replyUse = MessageManager::Instance()->SendMessage(msgUse);
        std::vector<std::string> messageContent = *(std::vector<std::string>*)replyUse->GetContent();
        MessageManager::Instance()->SendQueuedMessages("remove");

        for (int i = 0; i < (int)messageContent.size(); i++)
        {
            if (i != 0)
            {
                result += "\n";
            }

            result += messageContent[i];
        }

        return result;
    }
}
```

Figure 18: CommandUse.Process()

GameObjects, and a use Command, and added a Location to the Dungeon World to demonstrate their use (fig. 16).

- I updated ComponentFactory to include enumerated values for Flammable, Landmine and Button Components, to create Flammable and Landmine Components upon request, and to return via ComponentTypeExists() if a passed string matches a type of Component.

```
if (splitLine.size() != 5)
{
    std::cout << "Error, \"" << filename << "\", line " << lineCount << ": Wrong number of values for loading button component. V
    std::cout << generalComponentFormat;
    std::cout << buttonComponentFormat;
    loadedSuccessfully = false;
}
else if (!ComponentFactory::Instance()->ComponentTypeExists(splitLine[3]))
{
    std::cout << "Error, \"" << filename << "\", line " << lineCount << ": \"" + splitLine[3] + "\" is not a valid component type
    std::cout << generalComponentFormat;
    std::cout << containerComponentFormat;
    loadedSuccessfully = false;
}
else if (!locations.count(splitLine[4]))
{
    std::cout << "Error, \"" << filename << "\", line " << lineCount << ": triggers_at_location_id must be the id of a location t
    std::cout << generalComponentFormat;
    std::cout << containerComponentFormat;
    loadedSuccessfully = false;
}
else
{
    gameObjects[splitLine[2]]->AddComponent((Component*) new Button(gameObjects[splitLine[2]], splitLine[3], splitLine[4]));
}
```

Figure 19: World.World()'s handling of Button Components.

- I updated World.World() to handle when a world text file specifies a Button, Landmine or Flammable component is to be custom created; the former requires some Component-specific handling (fig. 19), while the latter two are handled more or less the same as Movable Components. I also had it use

ComponentFactory.ComponentTypeExists() to verify that component types specified in world text files for adding a Component or linking a button to a type of Component are valid Component types.

- I added CommandUse to the CommandManager's lists of Commands and implemented the functionality of its Process() method, configuring it to check if a specified Button exists at the current Location and is a Button, triggering it if it does exist, returning all output generated by it, and cleaning up any GameObjects to be destroyed (fig. 18). While doing so, I double checked that the messaging of the Flammable, Landmine and Button Components linked together properly with MessageManager and CommandUse, fixing bugs and reconfiguring the methods until they worked properly together.
- Once I finished the code, I updated the UML class diagram to match the final version of the code (see design diagrams).

What we found out:

- Filtering is good for restricting messages to being send-able only to game entities at a particular Location and/or of a particular type.
- Filtered broadcasting is good for triggering behaviours of all game entities of a particular type at a particular Location.
- Using messages to request information from Items and GameObjects that could be obtained via public members of those classes helps decouple them from the requester, but the code for creating the Message, sending it, and interpreting the response can be somewhat longer depending on the case, and less efficient than using public members.
- The way I've structured the hierarchy of GameObjects, GameObjects have the ID and type of any parent GameObject, but not those of the parent GameObject that their parent GameObject is childed to. Ideally, this would be dealt with by having all GameObjects be able to access a reference to their parent GameObject, but I'm wary of including such references in C++; in C# this would be fine, but I didn't want to deal with weird circular referencing crap in C++, so I kept the string ID and type fields and just let Flammable Components bypass needing the ID and type of any parent GameObjects of their own GameObject's parent GameObject
- Compiling reply Messages from a broadcast and then casting them properly to process can be a bit finicky and took a bit to get right. I found it easiest to just compile the non-nullptr reply Messages as they were into a vector of Messages and let the original sender process those messages, rather than try to compile their contents into a vector of void*'s. That just ended up being a mess of nested vectors and casting where I would forget what type things were supposed to be.
- With how things send Messages in this program, objects to be removed as a result of a Command should be cleaned up at the end by the Command, not while Messages might still be sent. Having the thing that triggered their destruction trigger the sending of their queued clean-up Messages left open the door to objects being removed while a SendMessage() call earlier on the stack was still iterating through a list they were in, and causing an error once the stack returned to that call to SendMessage().

Task 19 – Messaging Extended – Design Diagram