**Extension:** Task 19

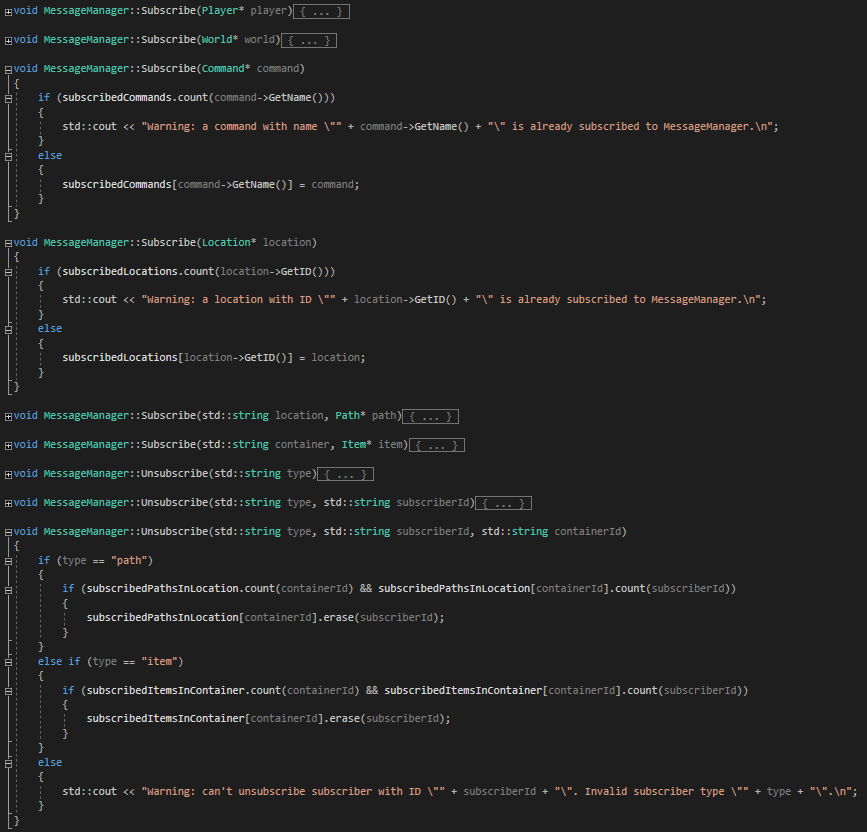
**Title:** Messaging Extended

**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:

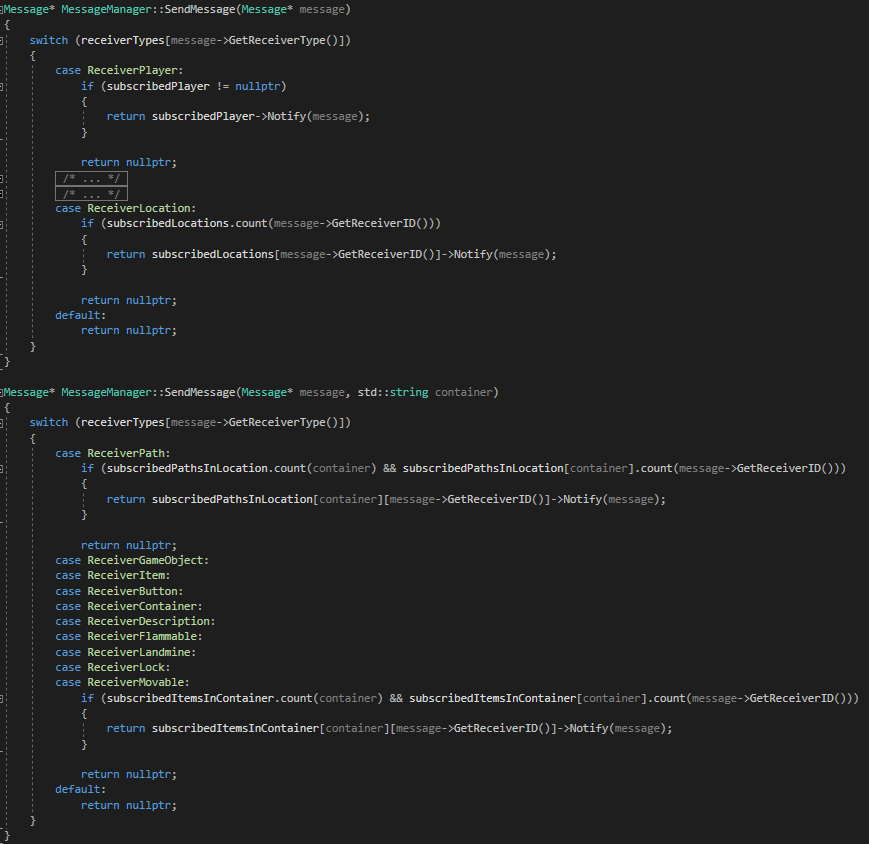
* I copied the “Zorkish Adventure” project and the task 18 spike report into the task folder, stripping out the spike report’s original content and replacing it with goals and resources pertaining to the task at hand.
* I had a look through the task instructions and considered what I could build to demonstrate 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 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).

Figure : 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.

Figure : MessageManager.SendMessage()’s overloads and how they handle their respective GameObject types.

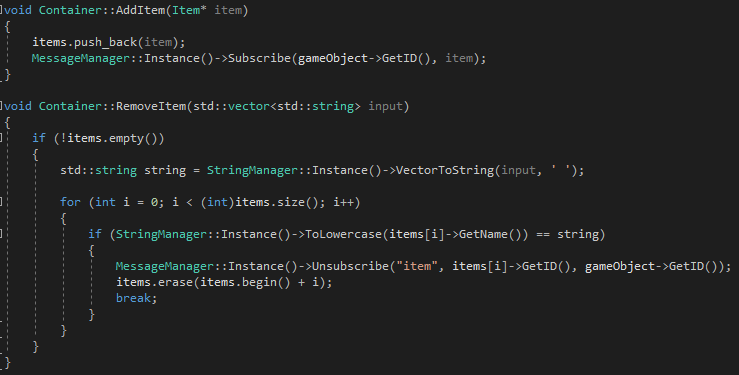
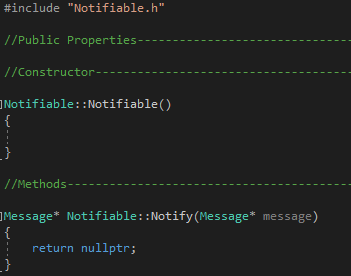
* I went through World.World() and found all now erroneous calls to MessageManager.Subscribe(), and updated them on a type-by-type basis to use the correct, overloaded Subscribe() method for that type. I removed the Subscribe() call for items, and added Subscribe() and Unsubscribe() calls to Container.AddItem() and RemoveItem() respectively to ensure Items are subscribed under the correct container (fig. 3).

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

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

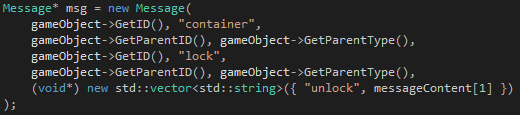
* I went through each class that had a Notify() method and made it or its parent class inherit from Notifiable, with Notify() being a virtual method of Notifiable that just returns a nullptr unless overridden (fig. 4).
* I added to GameObject a field containerId to store the ID of the Container Item 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.

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

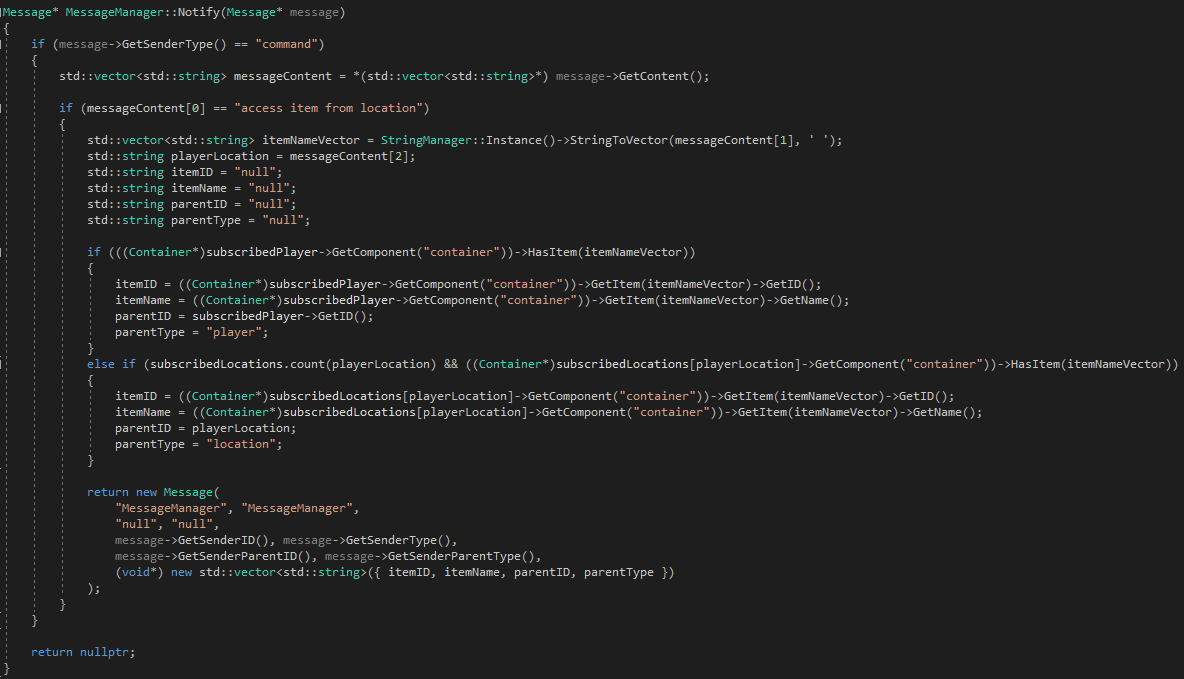
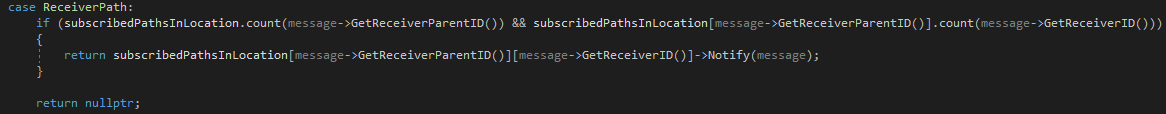
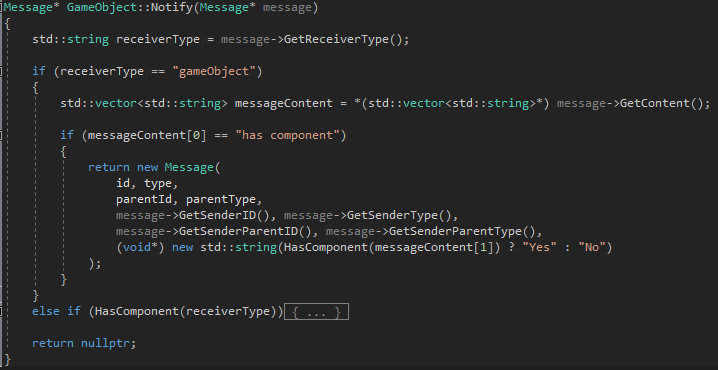
* I made MessageManager inherit from Notifiable and implemented for it a Notify() method that could handle queries for if an Item is accessible to the player from their current location, which is specified in the query Message so that MessageManager only checks with the Player and current Location, effectively 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  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.

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

Figure 7: MessageManager.Notify()

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

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

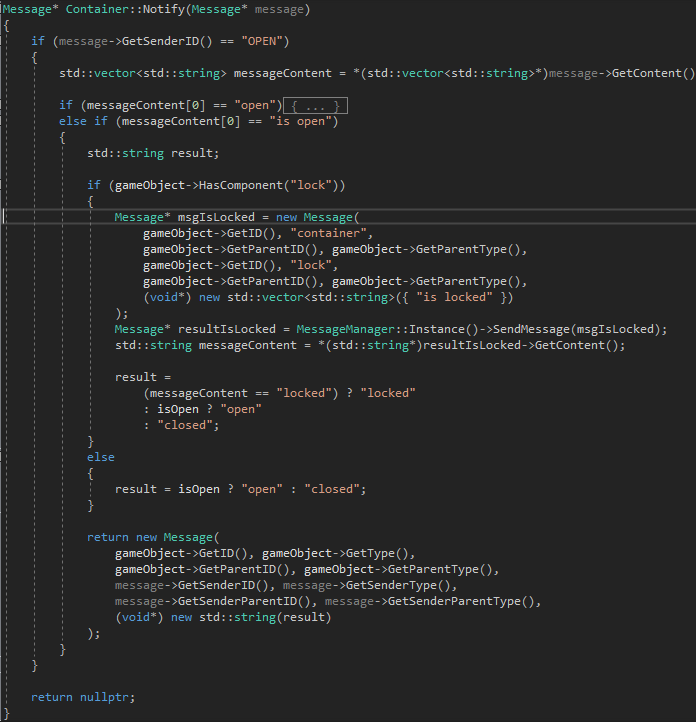
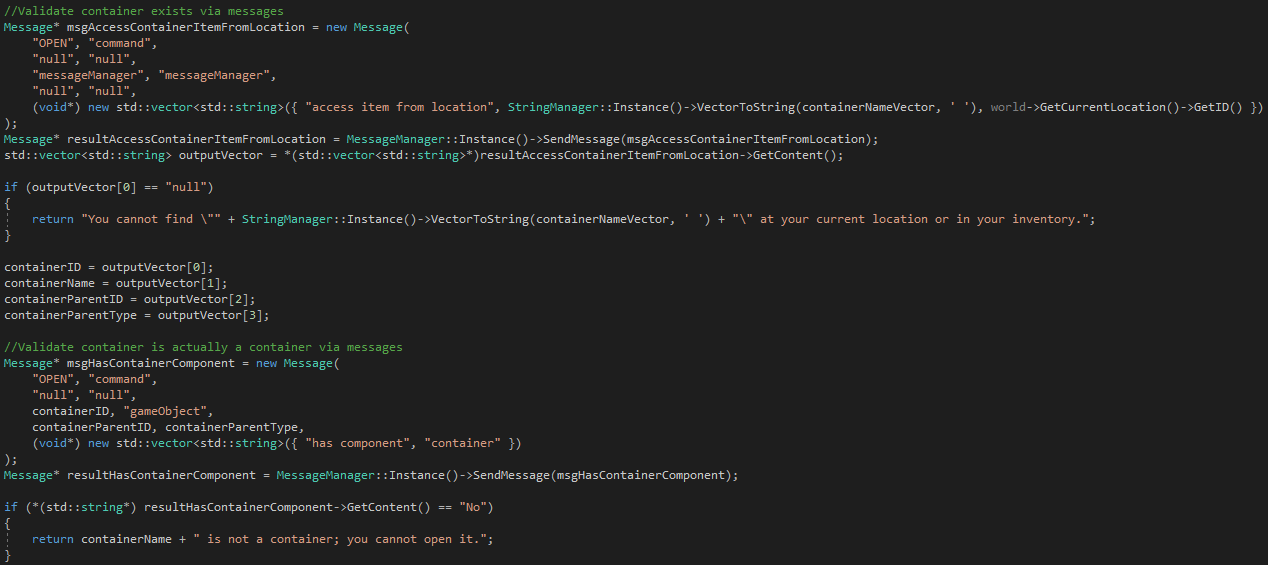
* I updated GameObject.Notify to be able to take queries for if it has a particular Components (fig. 8), and the Container and Lock Components’ Notify methods to be able to take queries regarding their current state (fig. 9, fig. 10).
* I updated CommandOpen.Process() to operate entirely using messages 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.
* TODO: add Button, Landmine and Flammable Component classes
  + Skeletons
  + Text file specification
  + World.World() handling
  + Internal behaviour

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

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.

# What we found out:

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

* 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.

# Task 19 – Messaging Extended – Design Diagram