**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:

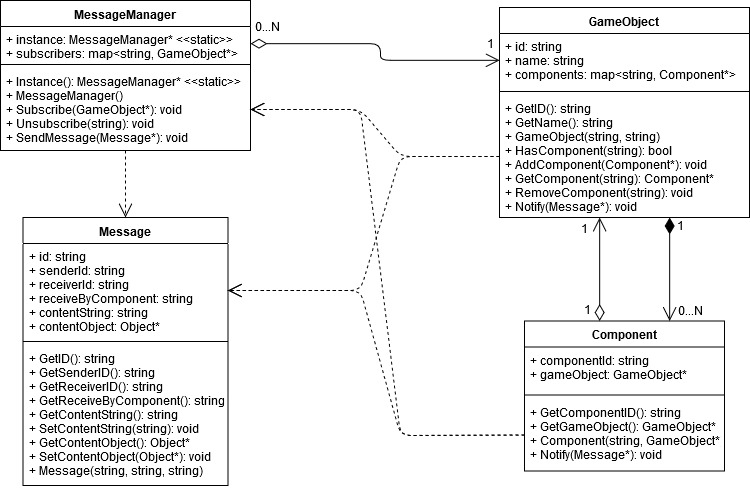
* I copied the “Zorkish Adventure” project and the task 16 extension report into the task folder, stripping out the spike report’s original content and replacing it with goals and resources pertaining to the task at hand.

Figure : the 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 std::any.
* 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.

# What we found out:

* The observer design pattern is well suited to the message dispatcher implementation of a messaging system.
* 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. Having a base Message class and a general Message class for handling already implemented messages, and then a Command Message class tailored to the Command system’s needs would allow unnecessary parameters to be cleaned up, as Command Messages would only need to encapsulate the information and objects required for the receiving command.
* 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 elected to use std::any, as it’s documentation indicated it had in-built safeguards for casting std::any values to other types.

# Task 18 – Message Systems – Design Diagram

Figure 1: The 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.