Message Specs / Design :

How messages are sent

Messages will be sent via an intermediary message handler who will determine who is the recipient of the message.

How messages are received and acted upon.

Each messageable object will have a receive message method which the message handler will invoke.

How messages are addressed

Messages are addressed using entity names (sword, chest, other items.)

What content is included in a message

A message will contain:

- Sender
- Receiver
- Message

This will all be stored as strings inside a message object.

How objects register to receive messages

Any object which implements the Messageable_Entity class will automatically be registered with the message handler.

Whether a message contains information about who sent it

A message will contain the sender in order to faciitate replies if required.

Will use the exact same messaging objects and system, the addressing will be slightly adjusted in order to support announcements and blackboards.

The recipient will decide if they are receiving messages or not, if they are, they will receive the message, if they are not, it will go on the blackboard until they are ready to receive it. The sender will not know any different. The MessageHandler will tell the recipient that there is another message waiting, the recipient will keep track of how many and check when they are ready.

Blackboards will keep messages indefinitely, if the destination is invalid, the message will simply be dropped. The sender will not find out.

If the sender wants to make an announcement everyone (except the sender) will receive the message. To make an announcement, the sender will use themselves as the recipient.

Spike: 11

Title: Messaging

Author: Parth Madhani,101901858

Goals / deliverables:

- Code
- To implement an easily expandable messaging system.
- To avoid refactoring old code as much as possible to support the new classes.
- Messaging Specification

Technologies, Tools, and Resources used:

- Visual Studio IDE
- C++ Tutorials

Tasks undertaken:

- Decide on a method to implement the messaging system, including required functionality determined by the specification.
- Implement the required classes in order to support this.
- Research methods in reducing coupling between the objects and the messaging system.
- Implement these methods.

What we found out:

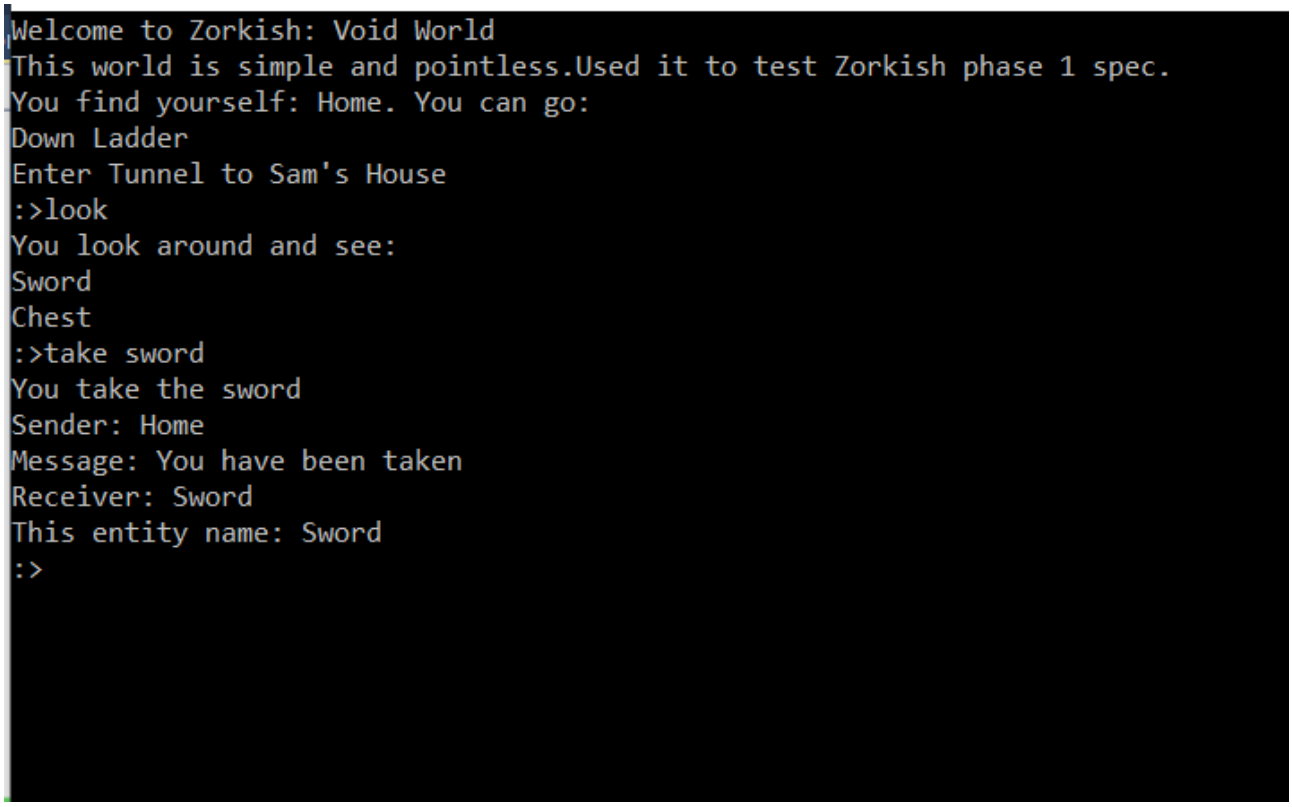
We found out how to implement a simple messaging system which can be easily expanded to add additional functionality such as incorporating an announcement or blackboard system through the use of a Message Handler to support sending messages.

The main goal we had in developing this was to have some way to reduce the coupling as much as possible so that a messageable object could just be one, and start sending and receiving messages as easily as possible. In the end, some methods need to be implemented by the child classes who decide to be a messageable object, but this is very minimal.

The MessageableEntity class handles every aspect to do with messaging, a child class need only provide a way to retrieve it's name (so it can be identified) and a way to process a message. The default behaviour simply prints the contents of the message when it is received.

One challenge that we faced during this implementation was dealing with the circular dependency between the MessageHandler and the Messageable_Entity class. This is because the message handler needs to have a list of the registered entities, and the entities need to be able to register themselves with the handler. This issue was solved by using a forward declaration of the Messageable_Entity class. Another issue I faced in achieving the coupling goal was, how do we allow each entity to know about the handler without passing a reference to each object through the constructor. If we didn't do this, then each time an entity is created we need to tell it who the handler is. Then we have to decide who is responsible for the handler. To solve this, we made a global MessageHandler declared in the MessageHandler. This way, anyone who has a need to use the message handler can simply include the header and have access to the handler directly. This approach even though it uses a global variable, is still controlled and is much more elegant than refactoring all of the old code.

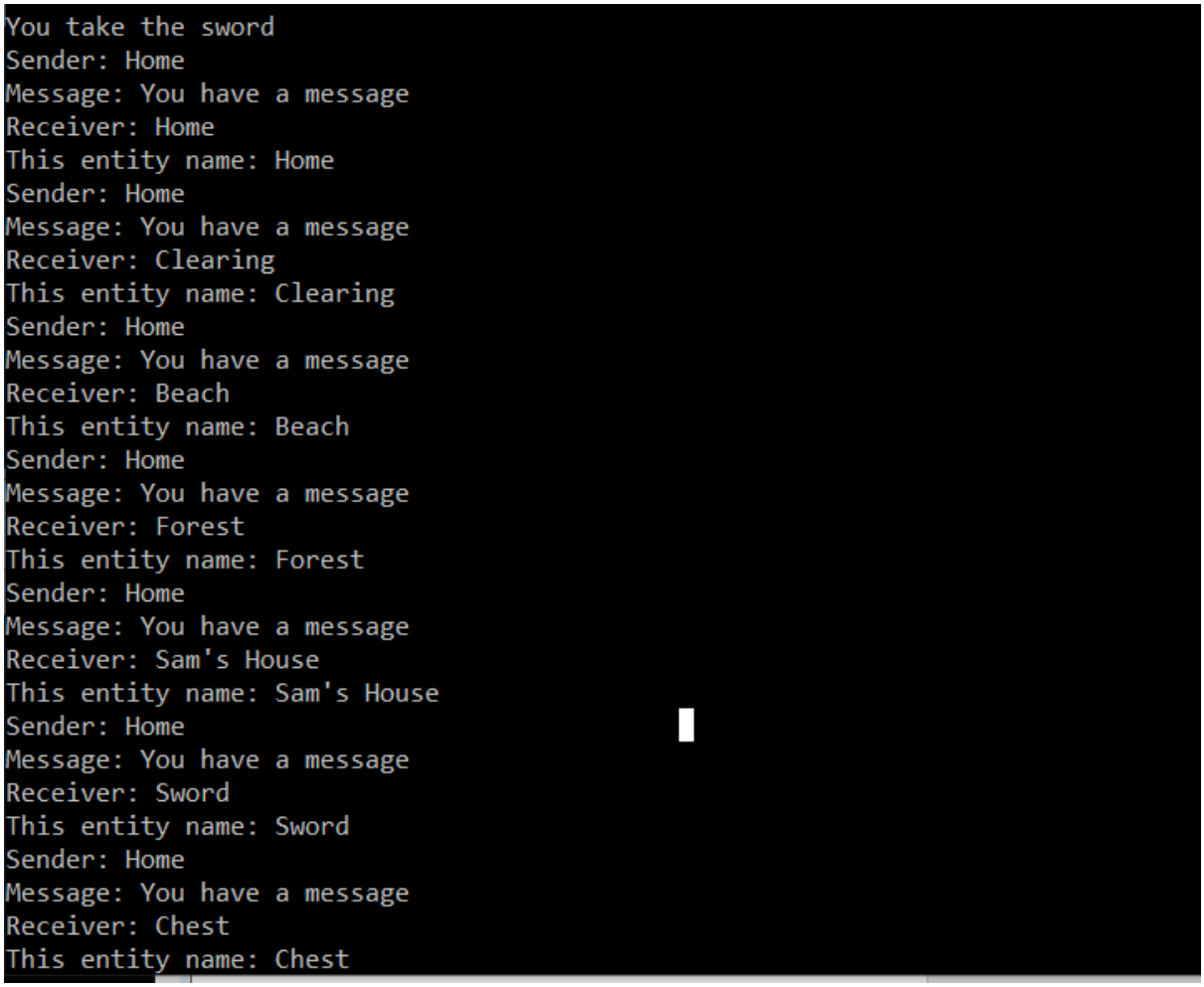
Testing Message Handler by using it for take command to get messages everytime an item is taken. Below is the output showing the messgae contents:



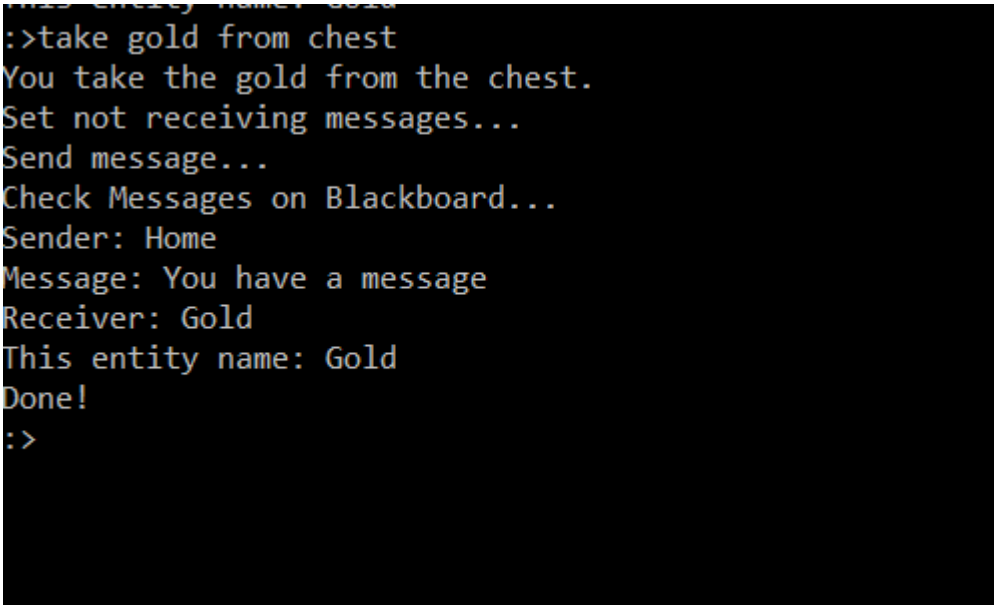
Coming to Blackboard/Announcements :

I found out how simple it is to expand the messaging system to support announcements and blackboards. We decided on a very simple structure in order to give rise to the additional functionality. When an announcement happens, it is received by everyone, even the sender. It is then up to the recipient to determine if the message is for them, if it is, what should they do with it. In order to achieve this, we added a simple case where the recipient IS the sender.

In order to support the blackboard aspect, each Messageable object now has some additional functions. This allows them to decide if they want to receive messages and also a function to check the blackboard. When there is a message on the blackboard waiting for an object, the object gets a counter incremented so it knows how many messages it has pending.

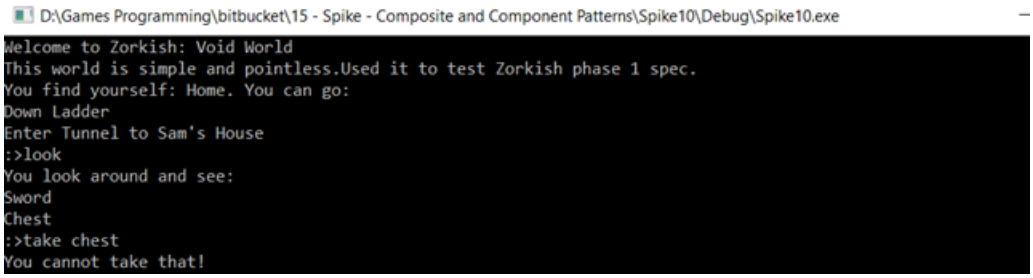


Announcement

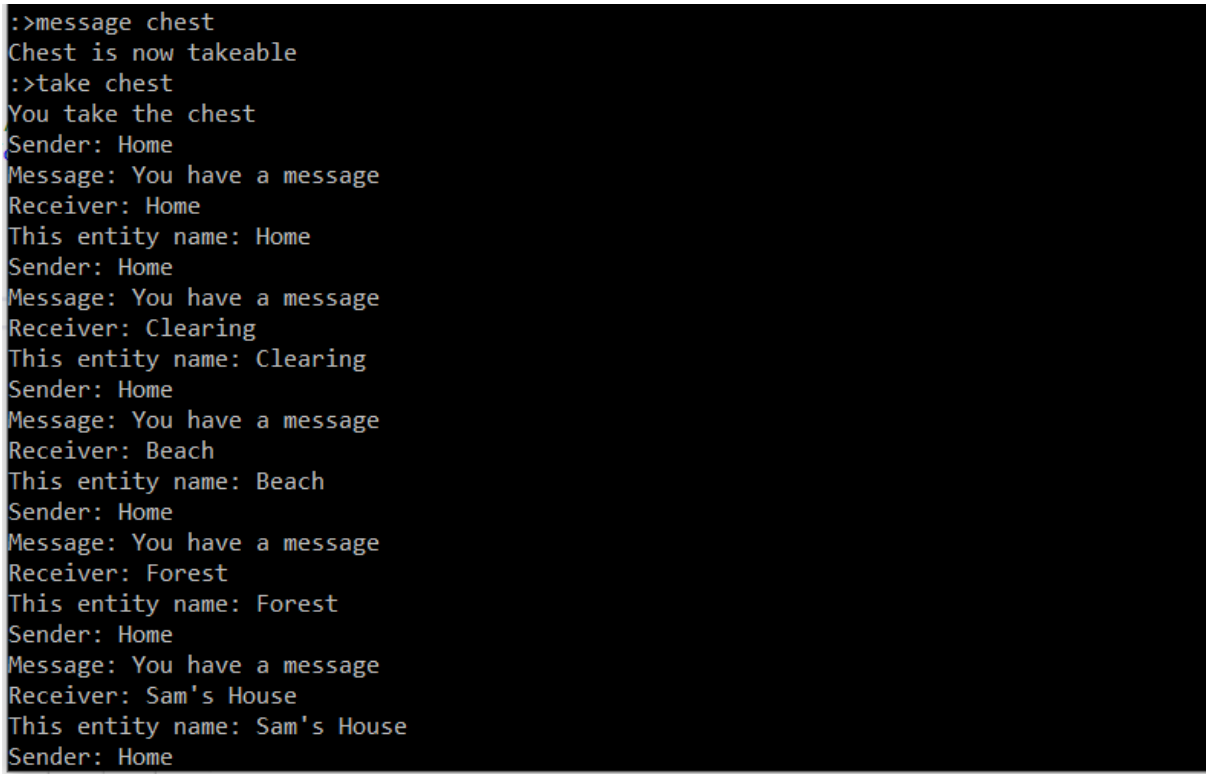


BlackBoard

Another thing I did was that when user uses ‘message’ command for an item such as chest which before was not takeable , by using message command it will change the component of that item to takeable and then user can take the item. In my case chest is not takeable by player but when player uses the message command it will make the chest takeable and allow the user to take the chest then.



Chest not takeable



Player messages chest and then player can take chest