Spike: 11
Title: Messaging

Author: Ford Killeen, 9731822

Goals / deliverables:
Goals this spike aims to achieve:

- Extend upon Zorkish by adding a messaging system
- To put some thought into how you implement the messages and what a message may consist of

Deliverables required:

- Code showing off the messaging system
- Spike report
- Messaging specification

Technologies, Tools, and Resources used:
The following is required to complete this spike:

- Visual Studio 2015
- Zorkish game specification
- Online C++ references

Tasks undertaken:
The list below details the steps taken to complete this spike.

- Firstly I got a pen and paper and drew a quick plan of how I wanted my messaging system to work. It consisted of a `Message` class, `Listener` class and `Messenger` class.
- **`Message`**. The message class represents an actual message, each message will have a unique id, a tag representing what type of message it is and the contents of the message in a string.

```
class Message
{
private:
    int id;
    Tag tag;
    string content;
public:
    Message(Tag _tag, string _content);
    void Message::setId(int _id);
    int getId();
    Tag getTag();
    string getContents();
};
```

- **`Listener`**. The listener class is a pure virtual class that will be inherited by any class that is going to receive and handle messages.

```
class Listener
{
public:
    virtual void handleMessage(Message* _msg) = 0;
};
```

- **`Messenger`**. The messenger class is what sends out the actual messages, as it contains a list of all of the listeners to send new messages to. This class is also a singleton and as such is static so it can be accessed from anywhere, allowing anything to send messages at any time.

```
class Messenger
{
private:
    Messenger();
    int _currid;
    static Messenger *_instance;
    vector<Listener*> listeners;
public:
    static Messenger &instance();
    void addListener(Listener *_listener);
    void sendMessage(Message *_message);
    int getNextId();
};
```

- Then I just implemented the messaging system throughout the game, by making the `Entity` class a subclass of `Listener`, and adding messages that are sent on certain commands input by the user. These are then handled by the entity in the `handleMessage()` method.

```
void Entity::handleMessage(Message* _msg)
{
    switch (_msg->getTag())
    {
    case HEALTH:
        if (hasComponent(HealthComponent::IDENTIFIER)){ ... }
        break;
    case DAMAGE:
        if (hasComponent(DamageableComponent::IDENTIFIER) && hasComponent(HealthComponent::IDENTIFIER)){ ... }
        break;
    }
}
```

```
Starting : Test Adventure

 -> use sword
Entity health: 10

 -> use potion
Entity health: 15

 -> use sword
Entity health: 5

 -> use sword
Entity is dead
```

What we found out:

By completing this spike we found out how useful a messaging system is within a game environment. No longer do I need to have references and pointers to objects scattered around the place, or pass in more data than required in update calls, now I can just kick off a message from anywhere in the code which will go down and be handled by the entities we care about.