Spike: 10
Title: Component Pattern

Author: Ford Killeen, 9731822

Goals / deliverables:
Goals this spike aims to achieve:
- Extend upon Zorkish by adding entities that are made up entirely of components
- Entities that receive actions as well as attributes

Deliverables required:
- Code showcasing the component pattern in use
- Spike report
- Any hand written notes or diagrams

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 started by creating the `Component` class, which is the virtual class to be inherited by all other component subclasses. All it contained was a virtual `update()` method.

```
class Component
{
public:
    virtual void update() = 0;
};
```

- Next I set up the `Entity` class, which acts purely as a container of components. It has functionality to add, remove, check and get components within itself.

```
class Entity : public Listener
{
private:
    map<string, Component*> components;
public:
    bool addComponent(string _id, Component* _comp);
    bool removeComponent(string _identifier);
    Component* getComponent(string _identifier);
    bool hasComponent(string _identifier);
    void handleMessage(Message* _msg);
};
```

- Once setting up these two classes, I worked on getting some basic component subclasses in, they were `HealthComponent`, `DamageComponent`, `DamageableComponent` and `PickupableComponent`. The first two agiving an entity some attributes while the latter two allow certain actions to be taken upon the entity.
- I then created a `System` class, with subclasses `HealthSystem` and `DamageSystem`, which would each take care of their respective components and call the `update()` method on them each cycle. Upon creating a component for an entity, it must also be registered with the system that handles those types of components.

```
class System
{
public:
    virtual void update() = 0;
};

class HealthSystem : public System
{
    vector<Component*> components;
public:
    virtual void update();
    void add(Component* cmpt);
};
```

- Now it was testing time, and I had a lot of trouble working out how the individual components within an entity, would instruct the components of another entity to complete an action. Say a sword entity is to inflict damage on the player entity, how would the `DamageComponent` within the sword know how to tell the `HealthComponent` within the player to set the health.
- Since the above situation puzzled me and was making the project look super messy and confusing, I decided to continue on with spike 11 (Messaging) and use that messaging system to handle communications between my components for this spike.

```
if (cmds[1] == "SWORD")
{
    Messenger::instance().sendMessage(new Message(Tag::DAMAGE, "10"));
}
```

- This was then tested and proven to show that the components within the entities could receive an action and handle the actions in their `update()` methods.

```
Starting : Test Adventure

 -> use potion
Entity health: 25

 -> use sword
Entity health: 15

 -> use sword
Entity health: 5

 -> use potion
Entity health: 10

 -> use sword
Entity is dead
```

What we found out:

By completing this spike we found out how to use the Component Pattern, as it enables us to create game 'objects' which are made up entirely of components, instead of multiple inheritance and creating massive classes for each type of entity. We also learnt that this allows us to reuse the code around components for multiple entities, without having to write that specific action into each entity class type.

Issues:

My main issue with this spike was the communication between the the components within entities and calling actions on components in different entities. It was due to this issue that I decided to complete spike 11 in this same project so that I could utilise the messaging system to handle communication between entities. This seemed to work much better and caused less issues as I didn't have to reference or `#include` every other class into my `Component` or `Entity` classes.

```
//Systems
healthSystem = new HealthSystem;
damageSystem = new DamageSystem;
//Player
player = new Entity;
HealthComponent* health = new HealthComponent;
player->addComponent(HealthComponent::IDENTIFIER, health);
healthSystem->add(health);
DamageableComponent* dmgable = new DamageableComponent;
player->addComponent(DamageableComponent::IDENTIFIER, dmgable);
//Sword
sword = new Entity;
DamageComponent* dmg = new DamageComponent;
sword->addComponent(DamageComponent::IDENTIFIER, dmg);
damageSystem->add(dmg);
```