# Reflection

**200333470**

**yhuang64**

**Yuanmin Huang**

To me, my engine truly began to evolve from homework 2. For each homework, I implemented:

- HW1: raw game objects, raw component model
- HW2: time system, networking system
- HW3: component model, game objects based on components
- HW4: event management system, (replay system)
- HW5: scripting management system

To be honest, at the first homework assignment, I didn't know at all what I was doing. I just follow the instructions and met the requirements. As time goes, I began to realize that I was implementing my own game engine system together with a game based on it.

It wasn't until that point did I begin to generify my design and try to make it reusable instead of being specific to the 2D platformer I was working on, in case the engine would be used for implementing another game in someday, which became the homework 5 assignment in the end.

Though I have tried hard to make my design as general as possible, when implementing the second game, it turned out that my design for some part wasn't good enough, which made my second game design somehow painful.

Therefore, let me go through each part of my engine to describe their good and bad one by one.

1. **Time system**
   This part is not surprisingly the most reusable part in the engine. I didn't modify it a line when I was implementing the second game, which is fair because the time system is independent from the games based on the it. At least, no specific functionality on times is required for both of my games.
2. **Component model**
   Component model is a part that is quite reusable in my engine. As I have mentioned in the diff part of my writeup, the generic component class hasn't been changed at all to implement the second game. The three components, renderable, movable and collidable, are also sufficient and essential for both of the games.
   Which has to be modified is the implementation of behaviors of movable and collidable components, which is unavoidable because the game has changed after all. It's just the expected outcome of the generic component model design.

However, this can be better in my opinion. If we make the implementation of functional codes become scripts or binary codes independent of the component itself, we can just plug different files in to achieve different behaviors of the component, in which we no longer need to touch the native code to fit into different games.

3. **Game objects**

   Game objects make use of components. Thus, they are well-modularized to some extent. To build a new game, creating new game objects is also essential, but with modularized design, we can make the creating work more convenient.

   For my engine, the work has already been quite convenient. To create a new object, I first need to determine which functionalities it should have. Then, I just add the corresponding component to its map of generic components. Additionally, for some specific functionalities that are not general enough to build a component, I just put into the object as its attributes.

   This has made it hard for my game object design to become the most flexible one in which game objects can be created using configuration files, but in this way, I can have access to the attributes of game objects in other subsystems more easily. Given another chance, I may choose this design again.

4. **Event management system**

   This subsystem can be divided into events, handler and manager.

   a) For events, it's relatively similar to game objects design. Since I design the container for attributes of events to be variants, it becomes really simple to create new event type classes. All I need to do is to add a new enumeration element to the enum event_t, and then take in arguments to create variants to be stored.

      If I want to make event representations more general and easier to extend, I can use string for event type representation. Then, configuration files can be used for creating event representations. But, this will cause more inconveniency in event handling and managing. Hence, I would rather not do that.

   b) For handler, I'm using a giant handler with multiple handling functions to deal with different types of events. I feel that its good and bad.

      The good point is that it simplifies the design of the management system: there will no longer be structures like map<event_t, list<handler>> in the system. Instead, there will only be one handler.

      The bad point is that every time I have new event types and corresponding handlers, I have to modify the switch-case block manually and add those handler codes manually. If I could implement new sub handlers for new events instead of deleting the old part of codes and replace them with new ones, that would be much better in the context of software engineering. This approach preserves the original functionalities of the code.

   c) For manager, I'm using GVTs and queues to manage events, which turns out to be a quite good way to implement a manager. The reason why I draw this conclusion is that when I switch from my old game to the new one, only the replay related codes of the event manager were removed, and the rest of them remained unchanged. Despite the newly-added event types, the manager worked pretty well. This is the result of the independency of the manager from the handlers and event representations.

5. **Networking system**

   As is mentioned in the writeup, this part is the most painful part for me to transfer from one game to another.

   For my system, I fully redesigned the messages sent and receive for every stage of the client-server communication through the connection out to the information exchange and the disconnection. Actually, there are some part remaining, which would be the object movement event information, which are always sent and received by both sides of the network. Besides this and the code for ZMQ framework, all the other parts of codes are redesigned.

   What makes me even sadder is that I cannot come up with a way to fix this. I realized that for each game developed, there has to be a specific way of communicating through the network. Even though I know that the common stages are connecting, sending, receiving and finally disconnecting, different games have different characteristics that may ask for different content to be teleported through network.

   I believe there must be some mature models for networking in model game engine designs. I would touch them hopefully in the future.