# GLADDOS GAME ENGINE DOCUMENTATION

## ARCHITECTURE

---

**THE MODEL VIEW CONTROLLER DESIGN PATTERN:**
The model view controller design pattern was applied to the overall architecture for the game.

***Model***
The model in the Gladdos game engine is the shared world state between the client and the server. The server maintains this model and ensures that it is updated and accurate. The server also acts as a controller in that it will have AI (monsters) that will change the world state (Model).

***View***
The view is represented by the client's presentation of the world state to the player through the use of our GUI.

***Controller***
The controller is also represented by the client in that they will be sending new events to the server that will change the model(world state).

## UPDATES AND EVENTS

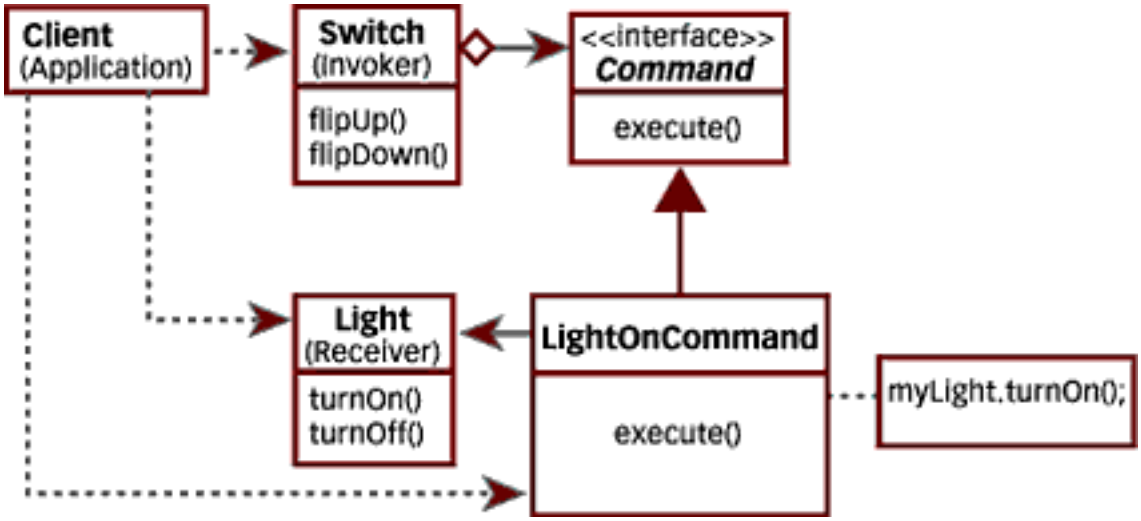
---

In the Gladdos game engine, the way that clients communicate with the server are through events. The way that the server updates clients is through updates.

In order to provide an efficient and elegant OO design, the events and updates design implemented the Command design pattern. This design patter allowed our implementation to be easily extensible and flexible.

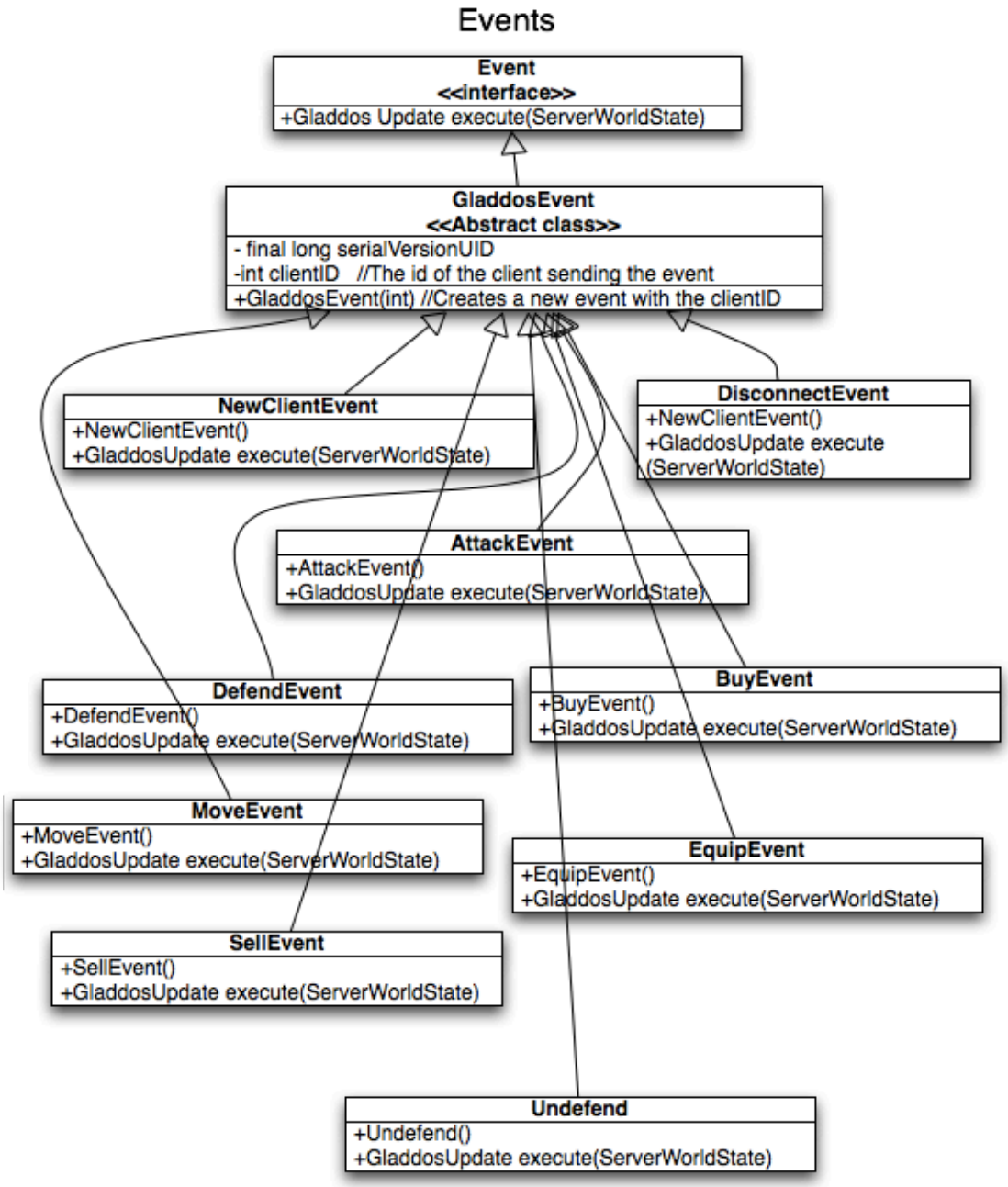
**The Command Design Pattern:**
The idea behind the command design pattern is that it allows total decoupling between an executor and a receiver. It makes it so that the server does not need to know the implementation behind an execute() command on an event. Therefore, it can call event.execute() on every event and not get involved on the details of performing the event. In the same sense, it allows our client to be unaware of how updates are performed, it can simply call update.execute().

An example of this design pattern with light switches:

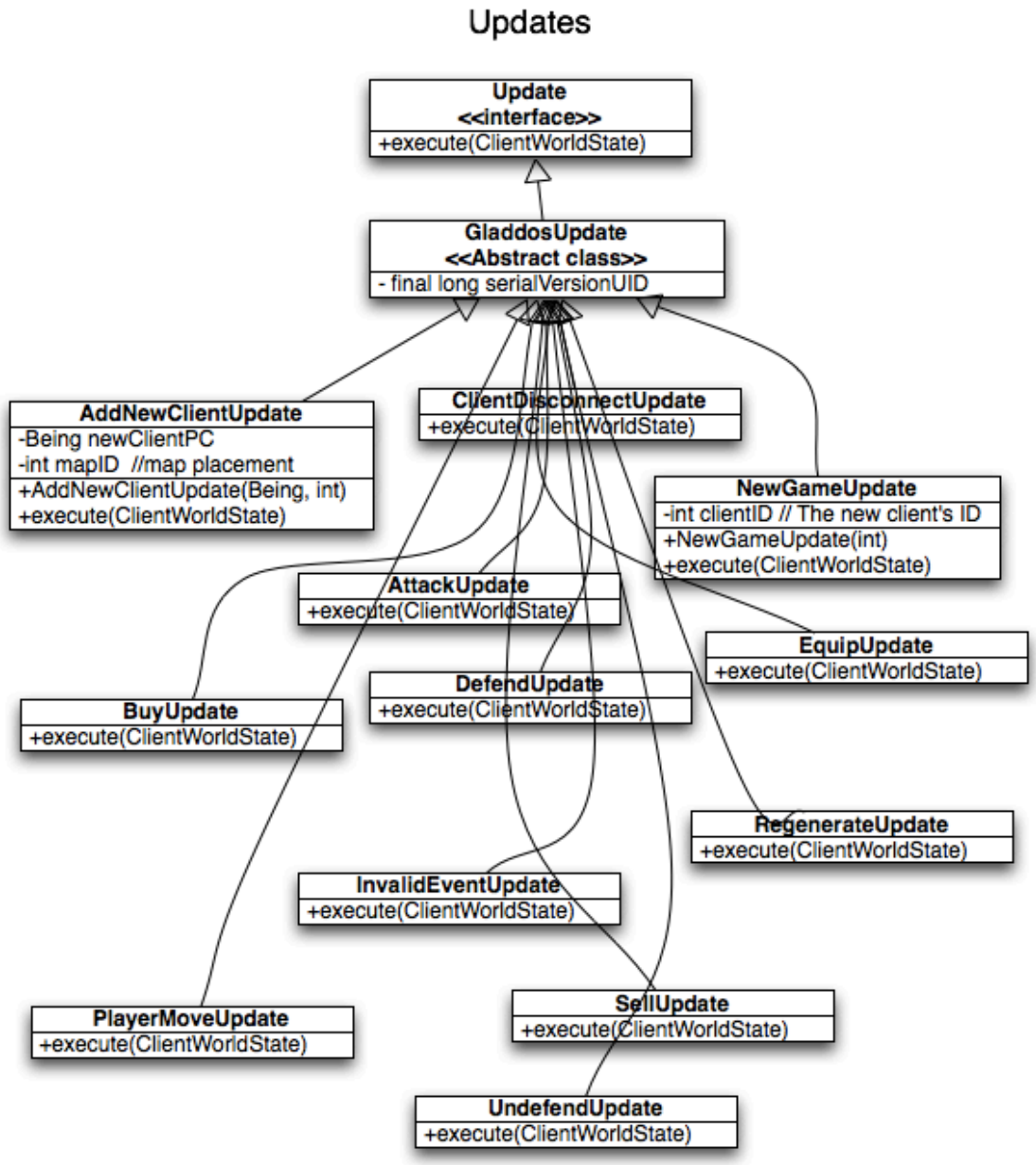



(http://www.javaworld.com/)

The way that we implemented this design into our game was using Events and Updates as our commands. In order to make the engine as extensible as possible, it was setup as follows. We had 2 interfaces, "Events", and "Commands". There were then 2 abstract classes that actually implemented these for use for our particular game. A new game could use these interfaces as well for extensibility. The two abstract classes were GladdosEvent and GladdosUpdate which allowed them to perform the operations on our particular game world states. We then had all of the game events and updates extend from these two abstract classes in order to provide each type of event's and update's functionality. The below UML diagram outlines the events and updates used in our game:




In addition, every Event can be relied on to provide the appropriate checking before modifying any of the world state objects. Therefore the Server doesn't have to worry about checking each Event, it can simply call execute() and the Event will determine if it is a valid event or not.




## WORLD STATES

---

In the Gladdos Game Engine, there are two world state objects. There is a ClientWorldState and a ServerWorldState. Essentially they both should always contain the same game objects, Updates will ensure this

syncing. Because both world states have very similar functionality except for a few key differences, a ***WorldState*** abstract class was created. Two concrete classes, ***ClientWorldState*** and ***ServerWorldState*** then extended ***WorldState*** and added on their specific functionality that they needed.

***Server World State***
The Server world state will be responsible for modeling the game in it's entirety. This world state lives on the server and is the world state that all the clients will be syncing with. **Events** will modify the ***ServerWorldState*** object.

***Client World State***
An instance of this world state lives on each client, and each client has his or her own separate ***ClientWorldState*** object. They all sync their world states with the server's, so they all should roughly be equivalent depending on where a client is in the sync process. **Updates** will modify the ***ClientWorldState***

## THREADS

---

The Gladdos game engine utilizes parallel processing in order to best maximize network and game performance. Threads are created in the following objects in a typical instantiation of the Gladdos game.

***Server threads***
1) Server
2) ClientListener (1 for each client)
3) ServerGameController

***Client threads***
1) Client
2) ClientGameController
3) The AWT thread for the client's GUI

The role of each is as follows: The server thread's only job is to listen for new client connections. When a client connects, this thread creates a new ClientListener thread and then places the client into the broadcast client list.

The ClientListener thread's job is to listen for incoming events from the Client Socket that it is attached to. When it receives an event, it places the event into the server mailbox to be processed by the server. This created a synchronization design that we had to take into consideration. Since multiple ClientListener threads would access this mailbox simultaneously, a synchronized data structure had to be used. We choose the LinkedBlockingQueue for this purpose. We choose this data structure because it is an efficient data structure with a high throughput. We ensure that it doesn't halt on blocking by checking to ensure the size isn't 0 whenever we take events from it.

The other synchronization issue we had to consider was the fact that multiple threads could tamper with the Broadcast object. The way that they can alter the broadcast object is through the addOut and removeOut methods. Therefore we synchronize on both methods to ensure that no concurrency issues are created.

Client-side, there were 3 threads we had to consider. The client thread's job was to listen for updates from the server and to send those updates to the ClientGameController. The ClientGameController thread's job was to continually update the client world state and display it to the user. The AWT thread's role was all Swing interaction. There was only one synchronization issue to consider, and that was on the sendEvent() method in the Client object. This method would write a serialized object to the Server's ObjectOutputStream. Both the client thread and the ClientGameController thread make use of this method. To solve this synchronization issue, we synchronized on the method, sendEvent().

**Note, when we say synchronized on the method, we imply that we add the synchronized keyword to the method to ensure that a lock is obtained on the class in which the method resides.*

## CHARACTER CREATION

---

The Gladdos Game Engine utilizes the Factory Method design pattern to allow dynamic character creation. In our CharacterCreationWindow, the client is allowed to select on of three classes. Depending on which class they select, a factory instance is assigned for that specific class type. It is

setup as follows, there is a PCFactory interface that contains a single method:

PC createPlayerClass(…)

Three concrete classes implement this interface, ThiefFactory, MageFactory, and FighterFactory. Each of them return the specific type of PC class the client specified. This is allowed since each of our 3 PC classes extend type PC.

## Game Controllers

---

The GladDos Game Engine maintains the visualization and the rate at which things are displayed in the form of two game loops. These are both controlled by **ServerGameController** and **ClientGameController**

### ServerGameController

This is where the main server game loop resides. This game loop is what controls the ***GLOBAL CLOCK TICK*** rate and determines the speed at which the game runs. Each iteration of the game loop empties the mailbox of all events received from clients and bots. It processes each event and places it on the Global Event Scheduler. It then checks to see if there is an event that takes place at the current time step. If there is, it executes the Event and then broadcasts an Update that reflects the changes the Event incurred. Then any events that execute for more than one time tick are executed, and events that had reset times are checked to see if they have finished updating. The game time increments at the end of each loop iteration

### ClientGameController

This is the tool that clients use to interact with the server and displays the current world state to the client. Each client has their own clientGameController instance. The role of the ClientGameController is to receive updates from the Client thread and to process them and update the Client World State from the updates. It then continually updates the visualization of the world state for the client.

**Cast**

The casting implementation is setup  with an interface from which all of our cast types extend from.  Each cast then implement and form the concrete implementation of it's specific cast type.

The way that cast's are sent to the server is through a CastEvent event type. This Even takes in a cast type as an argument into it's constructor and then it calls cast.execute in the event's execute method. There is additionally a CastUpdate that does the same thing.