Final Activity: Gold Collection

SWEN.383.03 - SW Des Principles and Patterns

Roman Pusec, 2014

Introduction

For this project, we had to incorporate our knowledge of Software Patterns to create the design for a 2D top-down online multiplayer game. Our assignment was to incorporate only the front-end design of the game, therefore I haven't established a concept of how the server-side works, but only how the client-side works. Class, object, and sequence diagrams are shown later in this document. The server-side of the game is illustrated with the SignalR library, which is a library that uses websockets to establish bi-directional client-server communication.

Class Definitions

GameHubRequest – The class which communicates with GameHub. It sends requests to the server whenever the clients interact with their specified player.

GameHubResponse – When the GameHub processes a request from the client, the response is handled by this class. This class also acts as a Facade since it holds a reference to GoldController and PlayersController. It is also the Observable, since it's supposed to update the View of a client.

GoldController – Manipulates with the Gold objects on a client's map.

PlayersController – Manipulates with the Player objects on a client's map.

View – This class contains the reference to GameHubRequest, GameHubResponse, ChatHubRequest and ChatHubResponse. It sends a request to the GameHubRequest class, and retreives a response from the GameHubResponse class. It is also the Observer.

GameElements – Holds all of the collections (players, gold, walls, etc.) which the View uses to determine collisions.

ChatHubRequest – This class communicates with ChatHub. Its purpose is to send requests that are related with client-to-client messaging.

ChatHubResponse – Retreives responses from ChatHub. Its purpose is to update the chat-log for all clients.

Item model – Consists of Player, Gold, Grass, Wall and Ground classes, with the Item class being their superclass.

Collision model – Set of classes which define the affect of the collision between two entities.

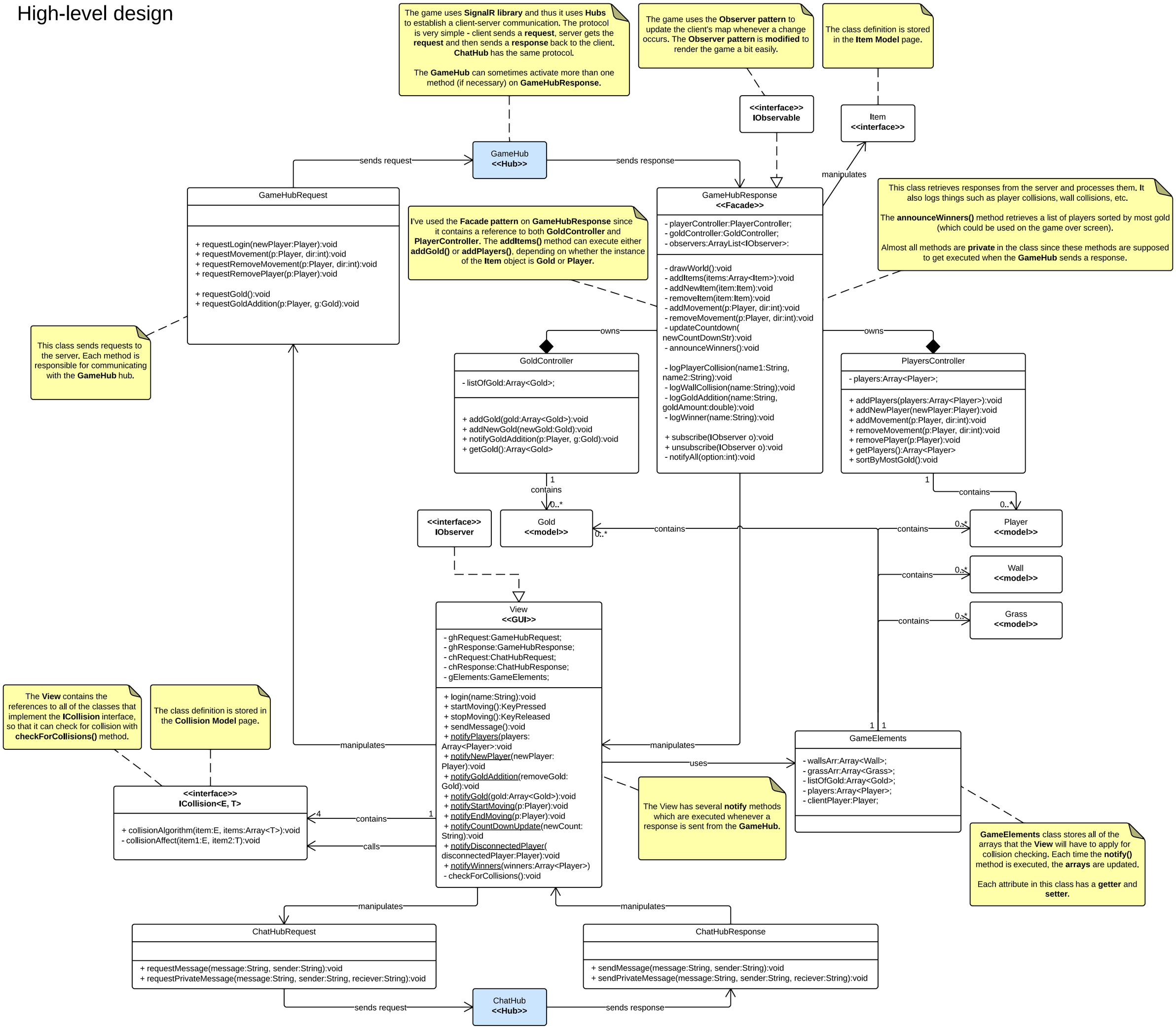
Resulting solution

The design of the game applies the SignalR library, which enables a bi-directional communication between a client and a server. That gave me the opportunity to create one class which communicates with the server(GameHubRequest , ChatHubRequest), and the other which broadcasts and filters the response of the server (GameHubResponse, ChatHubResponse).

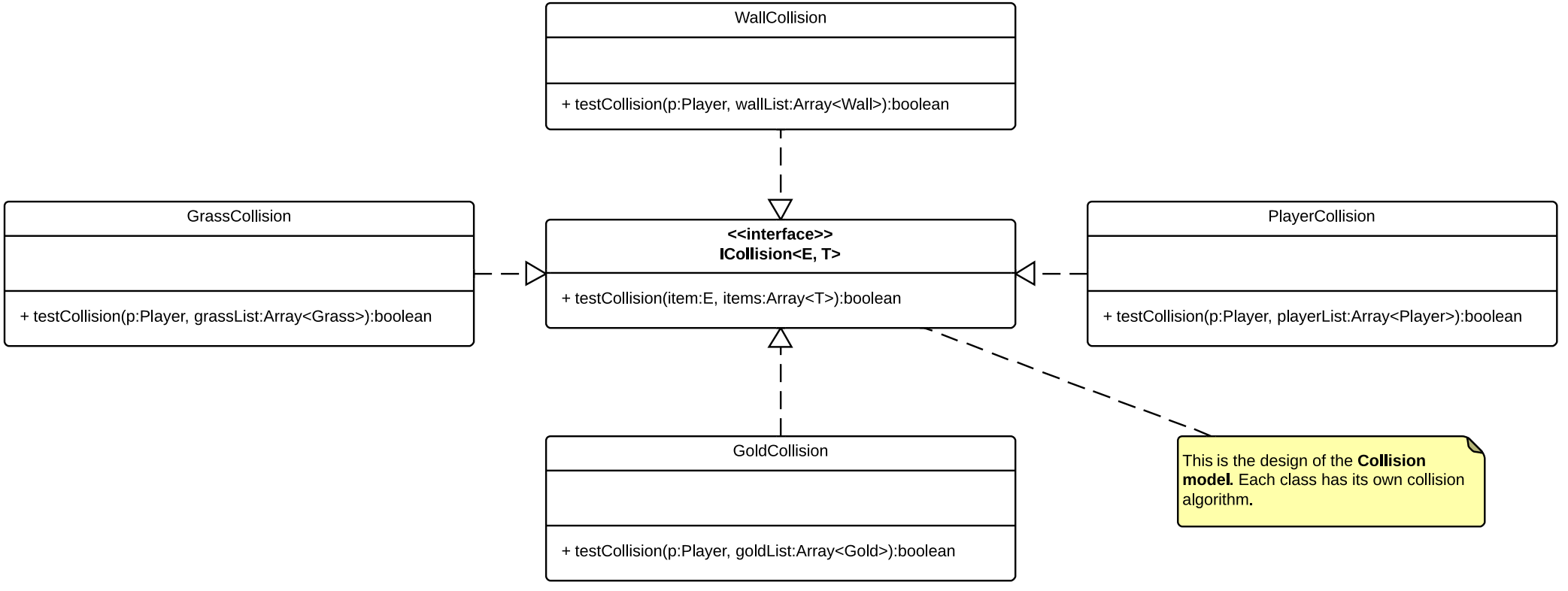
The game uses primarily the Observer pattern to render any response from the server on the client's graphical user interface. GameHubResponse is the Observable, meaning that it informs other Observers that a certain response from the server was established. I've also edited the Observable and Observer interfaces to render the client's graphical user interface more efficiently. The changes are available on the Observer page.

The game uses four different classes to determine collisions between two entities. Each of these four classes have a different collision algorithm, which support different affects when two entities collide. When a collision happenes, the View sends a request to the server that the resulting collision happened, and the server broadcasts it to all the other client.

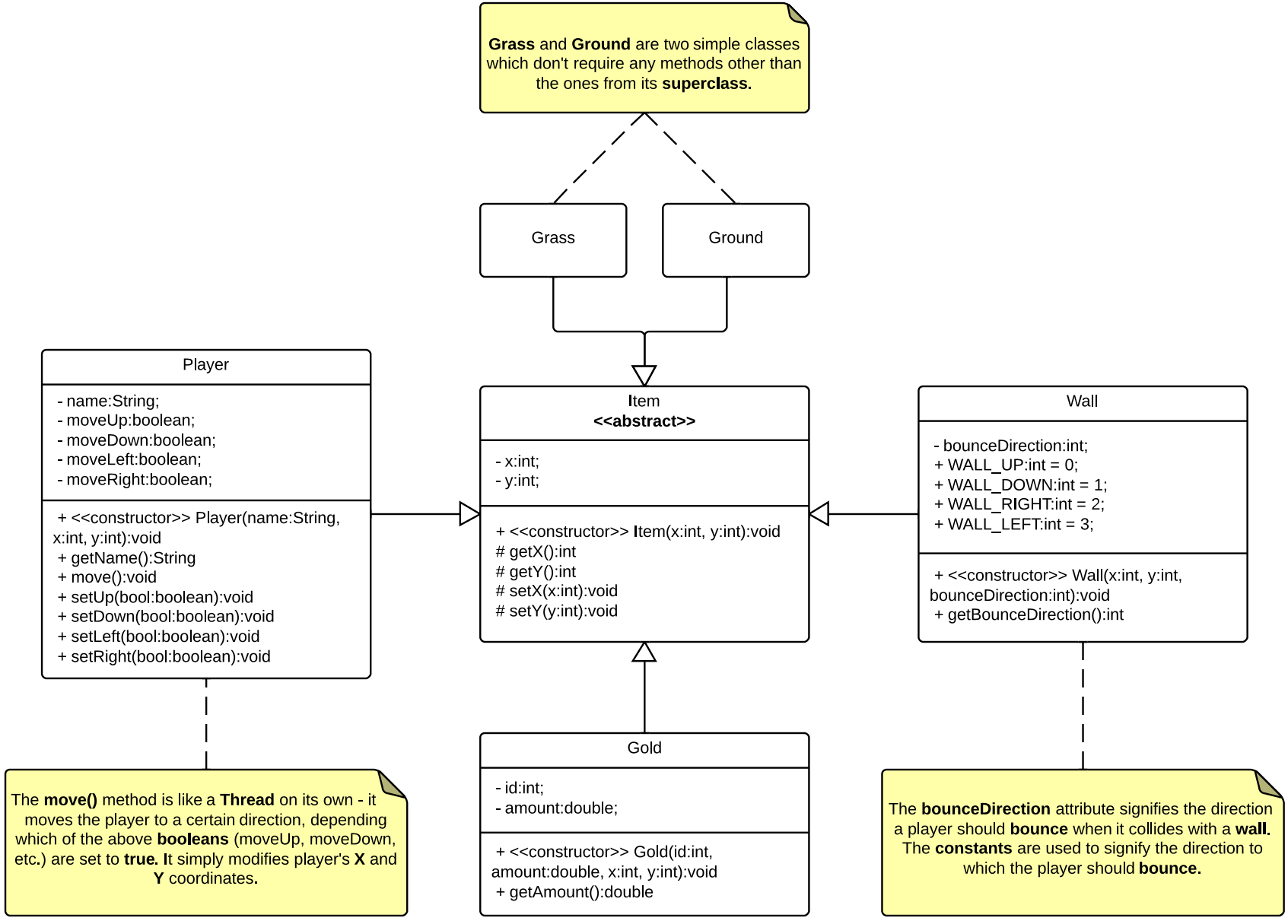
High-level design



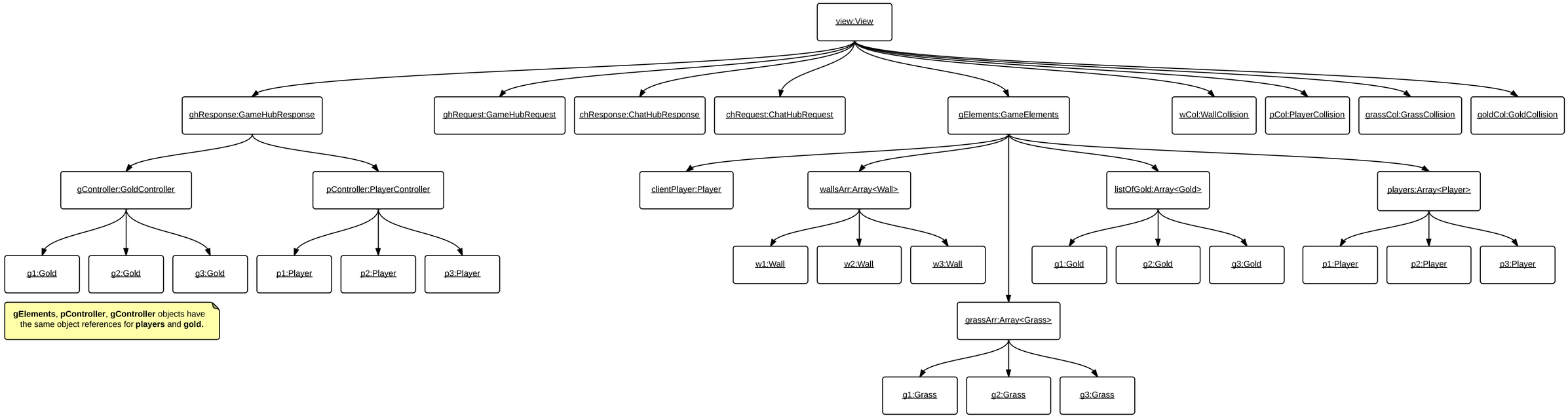
Collision model



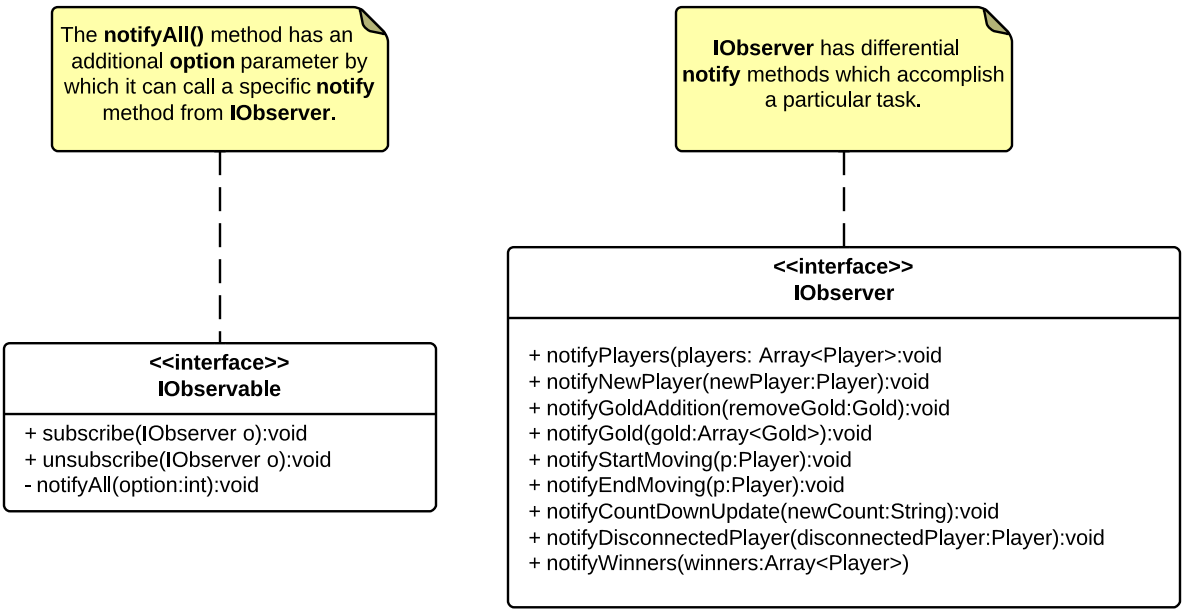
Item model


Grass and Ground are two simple classes which don't require any methods other than the ones from its superclass.
Grass
Ground
Player
- name:String;
- moveUp:boolean;
- moveDown:boolean;
- moveLeft:boolean;
- moveRight:boolean;
+ <<constructor>> Player(name:String, x:int, y:int):void
+ getName():String
+ move():void
+ setUp(bool:boolean):void
+ setDown(bool:boolean):void
+ setLeft(bool:boolean):void
+ setRight(bool:boolean):void
Item
<<abstract>>
- x:int;
- y:int;
+ <<constructor>> Item(x:int, y:int):void
# getX():int
# getY():int
# setX(x:int):void
# setY(y:int):void
Wall
- bounceDirection:int;
+ WALL_UP:int = 0;
+ WALL_DOWN:int = 1;
+ WALL_RIGHT:int = 2;
+ WALL_LEFT:int = 3;
+ <<constructor>> Wall(x:int, y:int, bounceDirection:int):void
+ getBounceDirection():int
Gold
- id:int;
- amount:double;
+ <<constructor>> Gold(id:int, amount:double, x:int, y:int):void
+ getAmount():double
The move() method is like a Thread on its own - it moves the player to a certain direction, depending which of the above booleans (moveUp, moveDown, etc.) are set to true. It simply modifies player's X and Y coordinates.
The bounceDirection attribute signifies the direction a player should bounce when it collides with a wall. The constants are used to signify the direction to which the player should bounce.

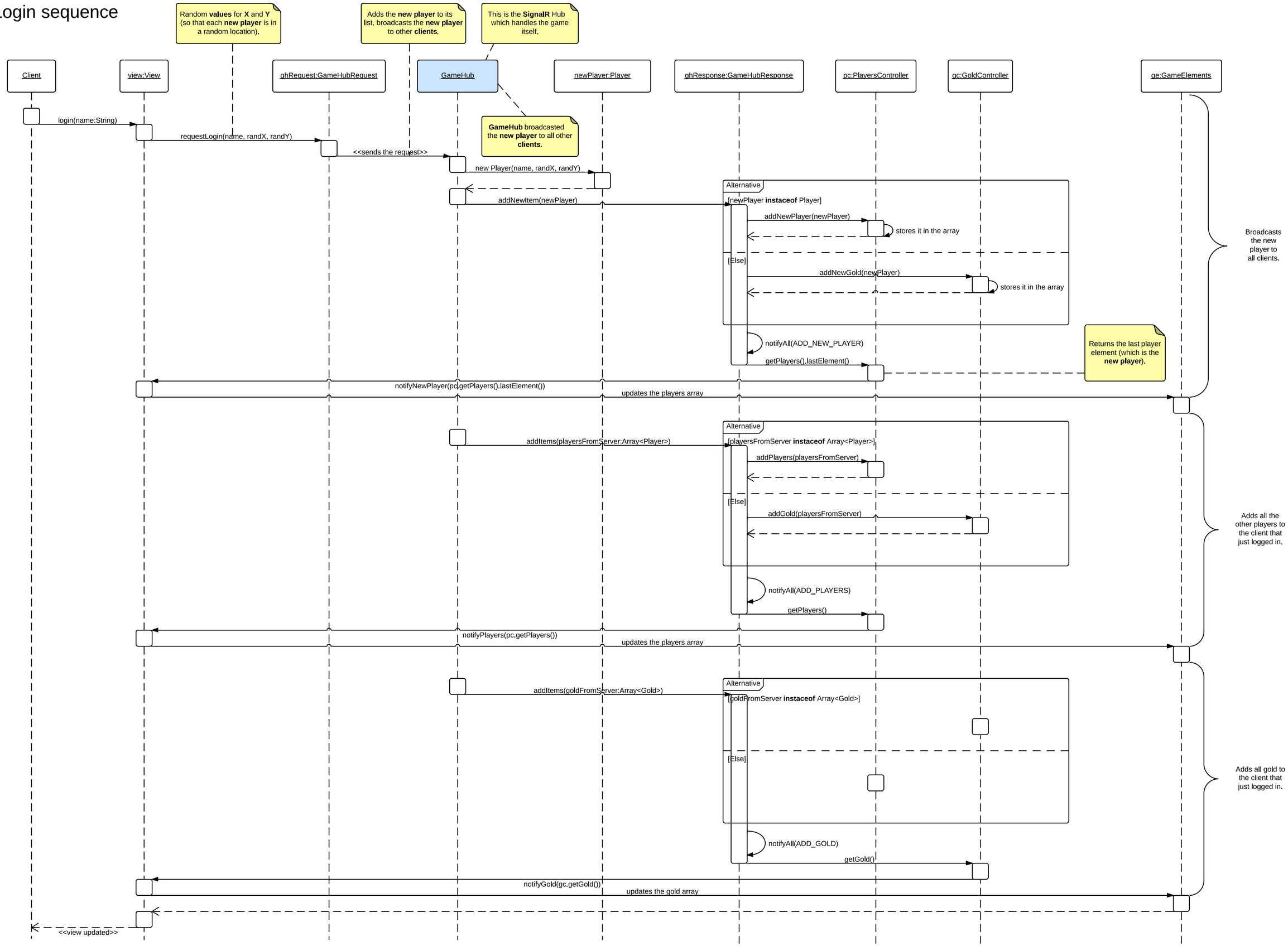
Object diagram



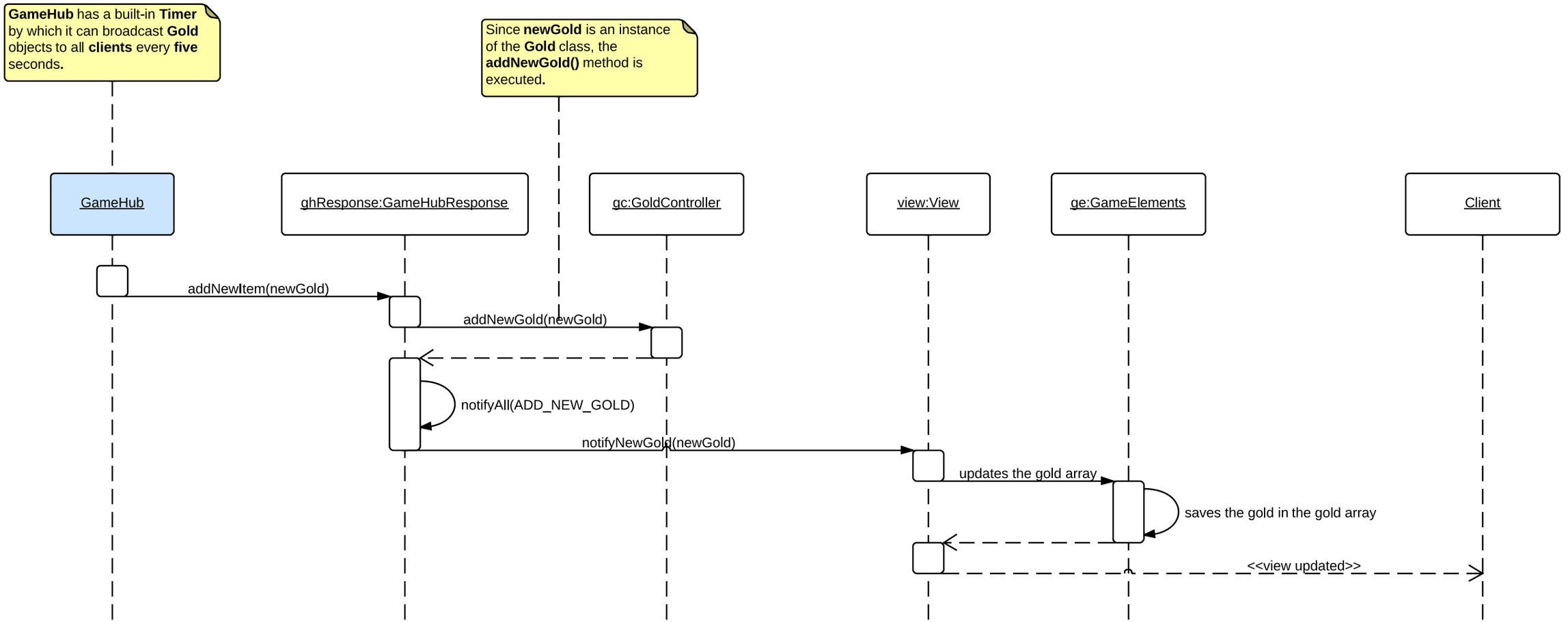
Observer pattern



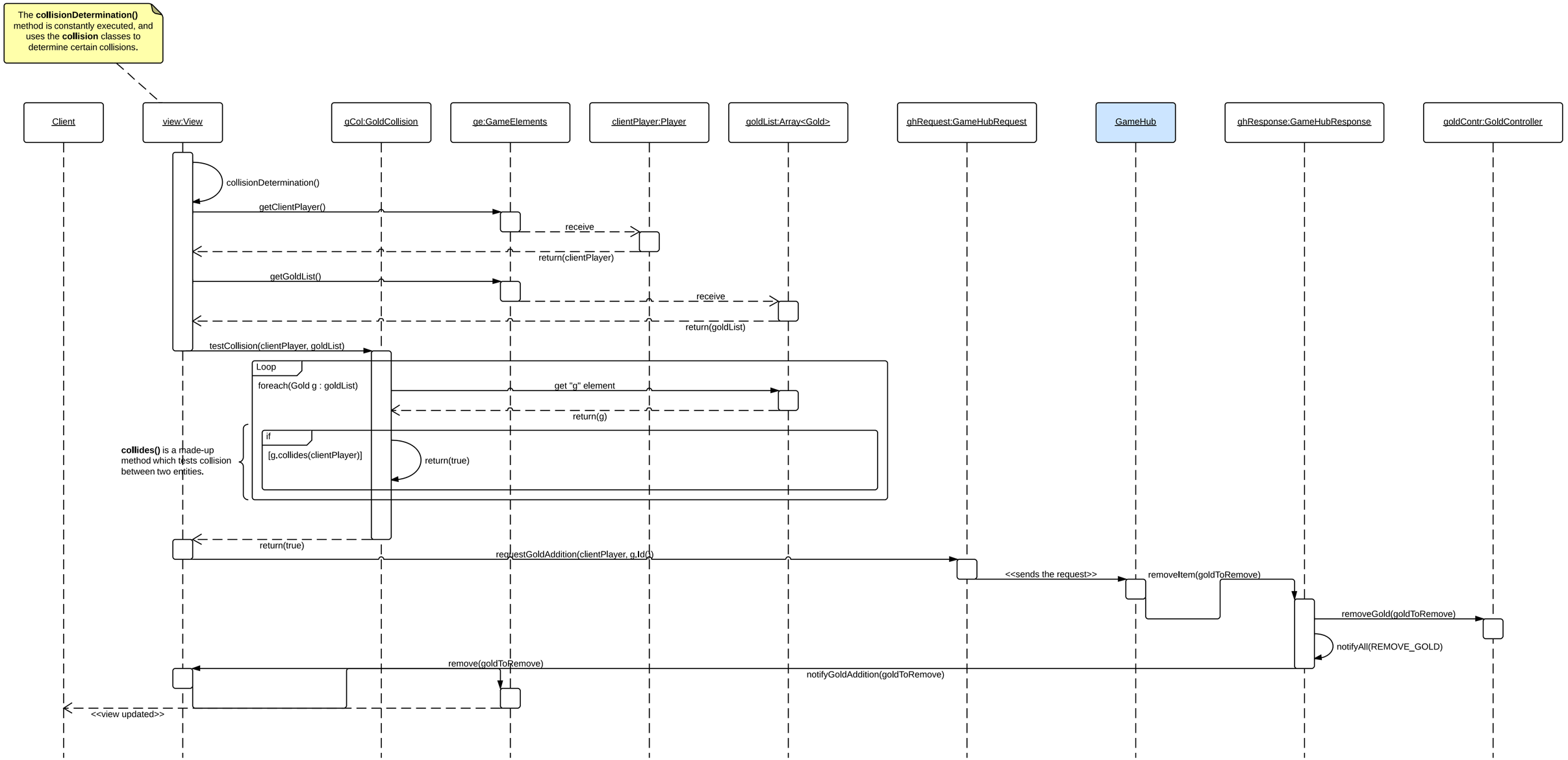
Login sequence



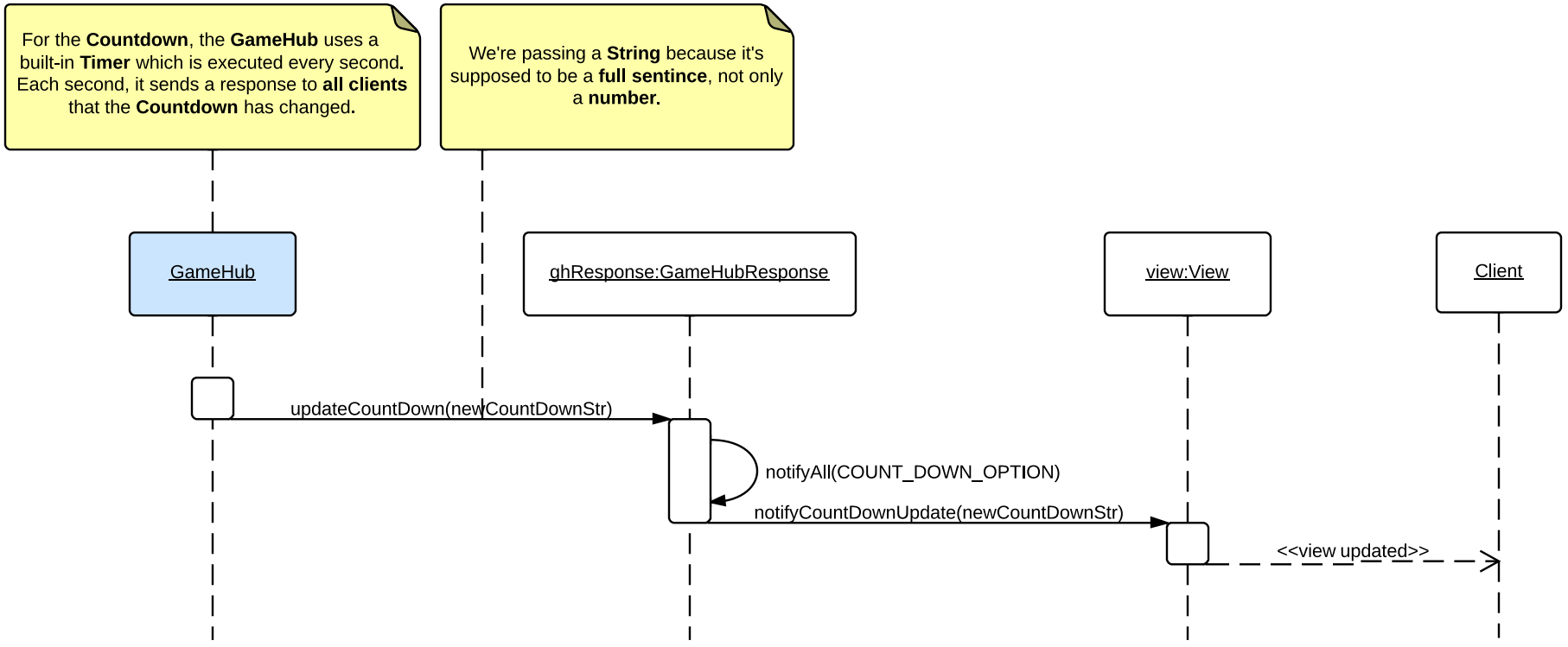
Gold addition sequence



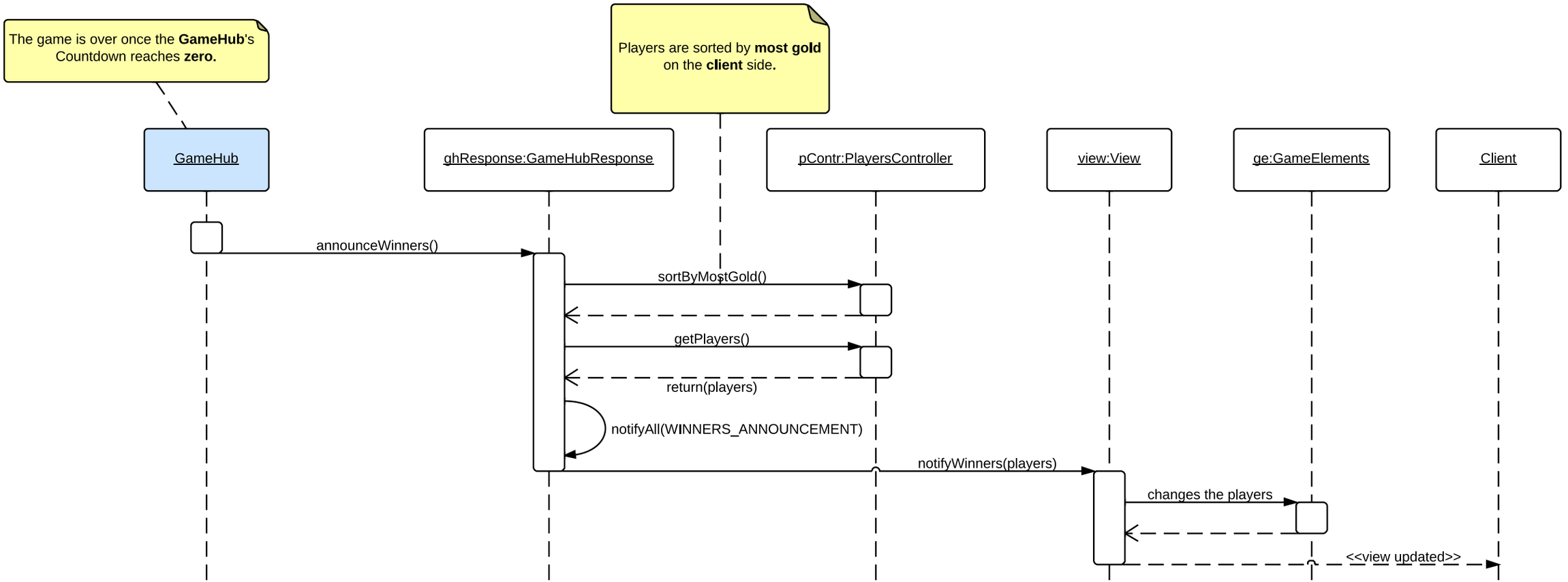
Player-Gold collision sequence



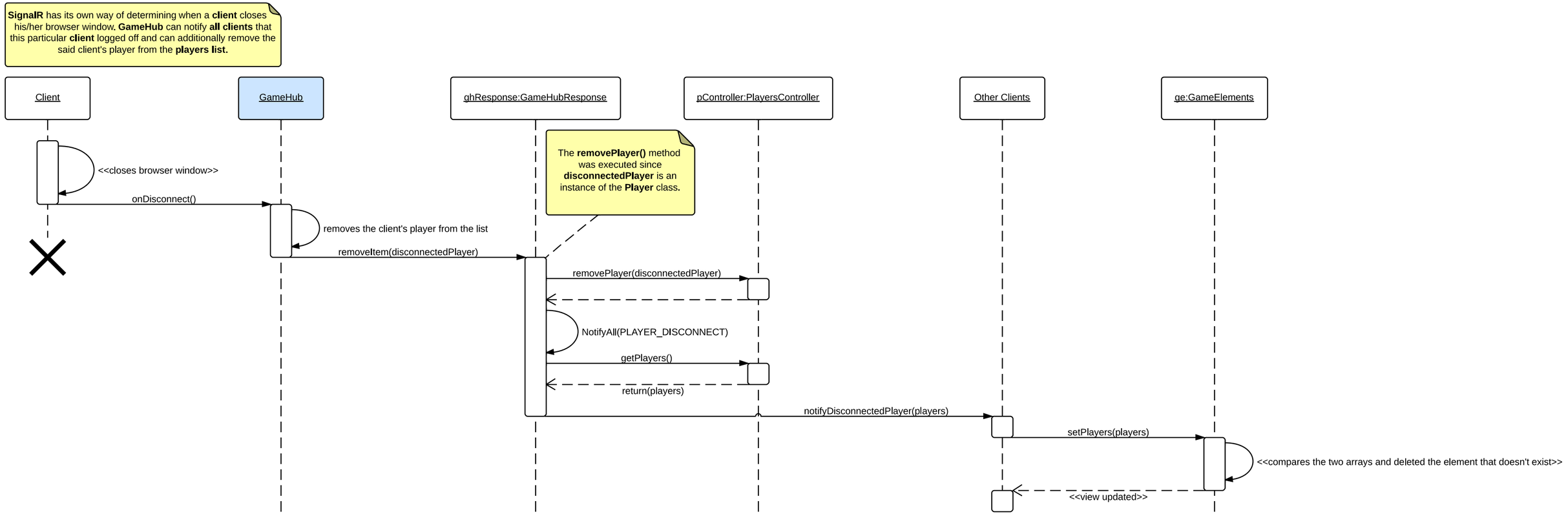
Countdown sequence



Game over sequence



Client disconnect sequence



Player movement sequnece