# Distributed Systems 2nd Homework

Artjom.Lind@ut.ee

November 11, 2015

The deadline for submitting is the 25th of November 2015. You can work in teams of 2. Do not forget to submit the names of your team members.

## 1 Preconditions

We all did great job implementing the game for the first homework[2]. Next, we are going to improve it using the knowledges we acquired during the last seminars. At this point we mastered following topics:

- Remote Procedure Calls
- Remote Method Invocation and Remote Objects
- Indirect Communication
  - Publish Subscribe Pattern
  - Message Queue
  - Event-driven communication

Practical of each above listed we covered using the following Python libraries:

- XML-RPC Lib for Python
- Python Remote Objects (PyRO)
- SnakeMQ message queue framework[3]
- Python Twisted event driven networking engine[1]

That's right we are going to use those technologies in order to improve our game. In order to do that we obviously need to find the disadvantages of the current design and implementation. As the amount of disadvantages is non-uniformly distributes across the the submitted homeworks, we start with our implementation. This will put everybody on the same starting conditions. In the next chapter we will refer to it as to FCF Game version 1.0.

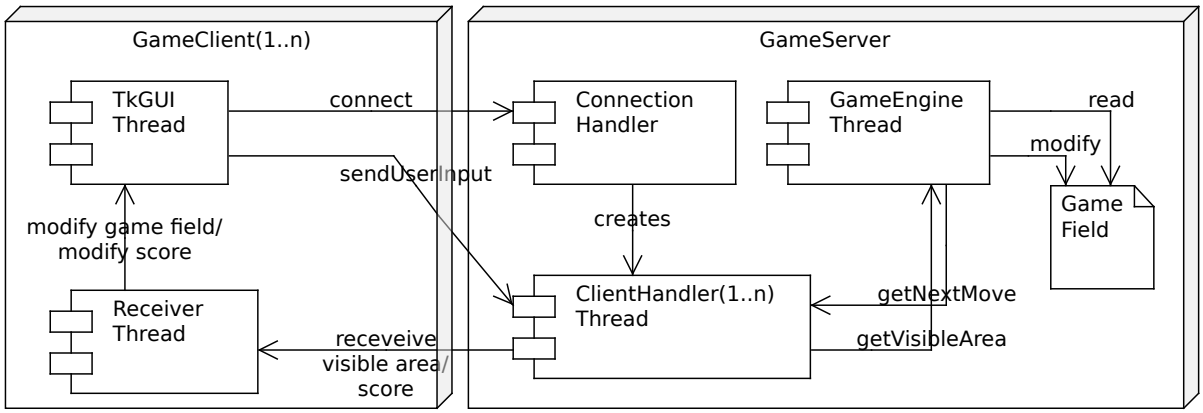


Figure 1: FCF Game Components

# 2 FCF Game 1.0

Here I provide with our implementation of the the corresponding game of the first homework[2]. Figure 1 illustrates the design and the focuses on the components of the system.

## 2.1 The game engine component

The server application contains the static game field component and one instance (singleton) of the game engine, which refers to the game field. The engine is programmed in very pragmatic fashion, it contains only one package called engines, which contains 2 classes: the Game and the Player. The rest is implemented relying on primitives, hence there are many engine related constants (Figure 2). The class diagram of the engine (refer to Figure 3).

## 2.2 Network protocol

The network protocol of the F2F 1.0 is very primitive, and is carried by 1 TCP connection between player client and game server. The whole protocol can be described in 2 parts, assuming both client and server applications have been started:

1. User provides server address and port and clicks Connect:
   (a) Server accepts connection, creates nameless player object, assigns random UUID and creates client connection handler by associating created player's UUID to connected user's socket.
   (b) Server starts corresponding client connection handler
      i. Handler attempts to receive messages from connected user and processes them.

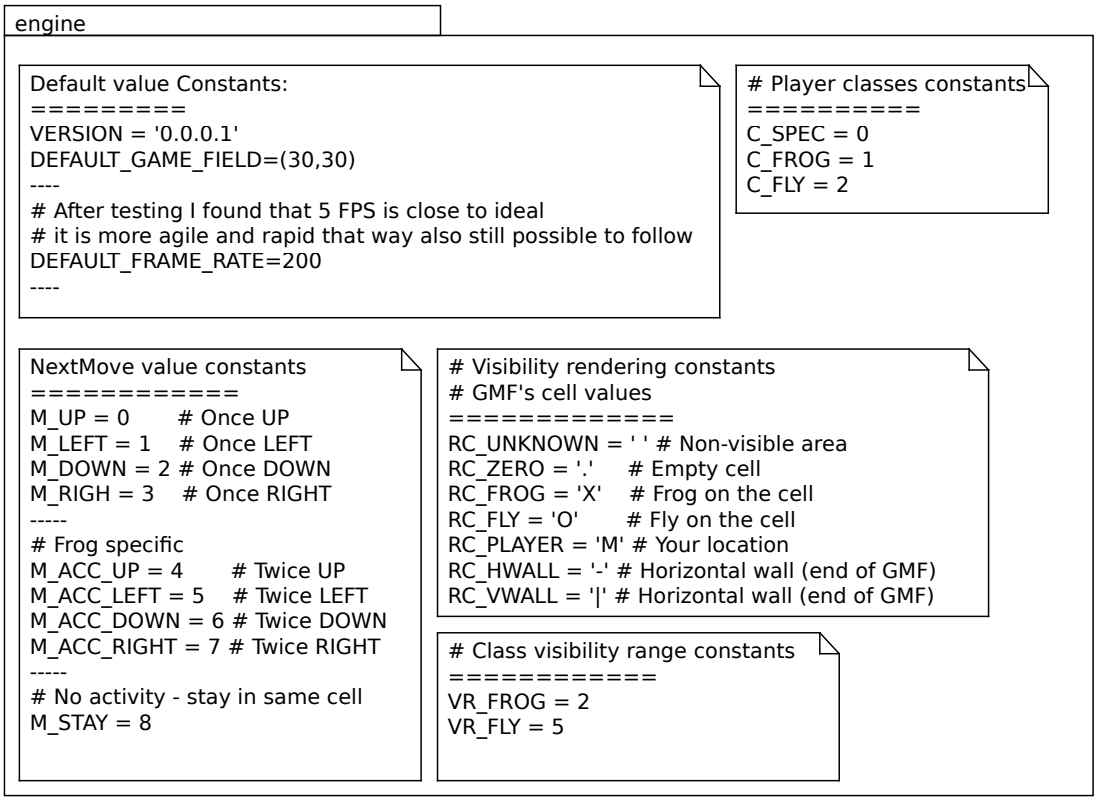


Figure 2: FCF Game engine related constants

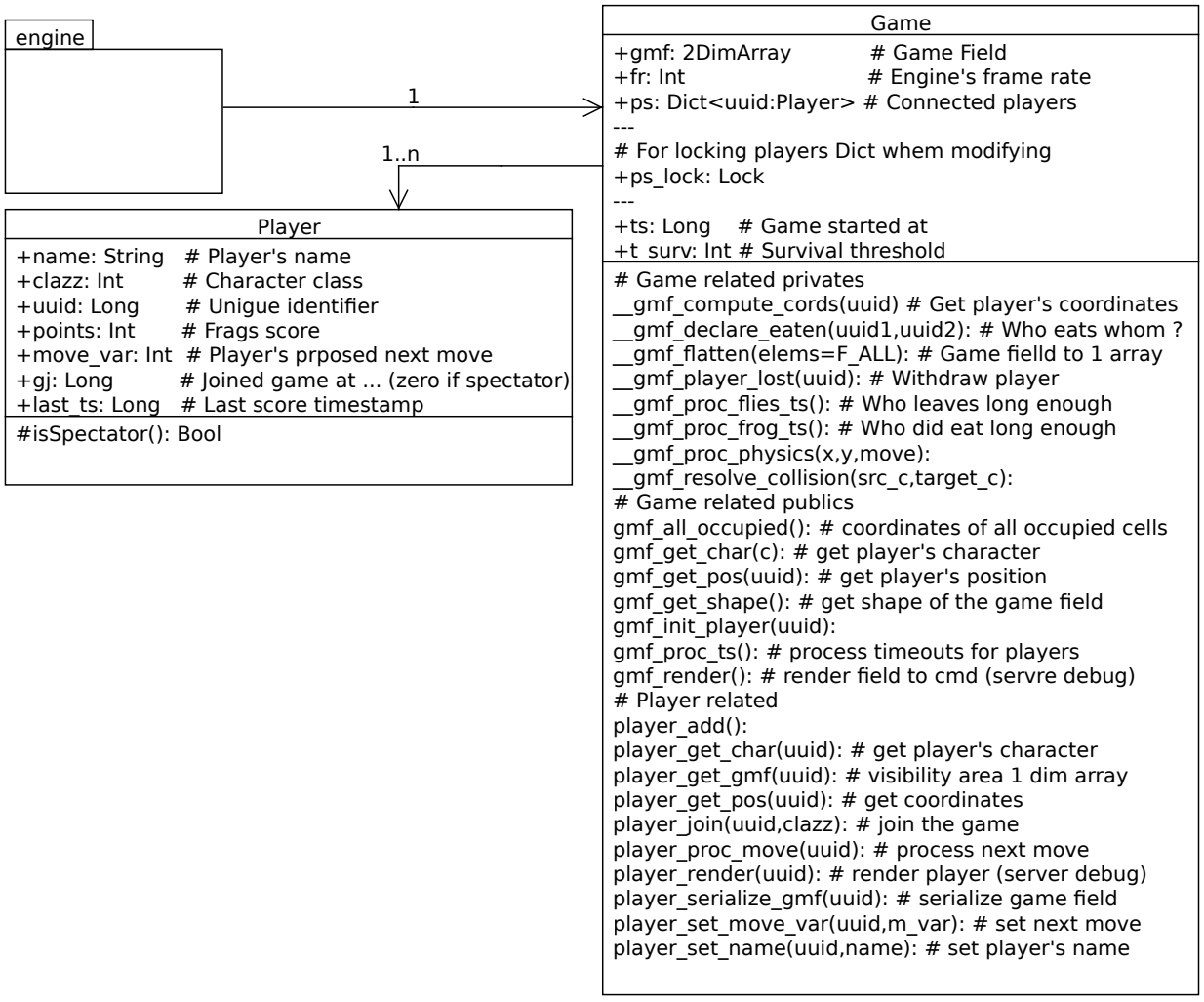


Figure 3: FCF Engine Class Diagram

2. User provides player's name and clicks register Register

   (a) Server assigns player's object with name, adds player to spectators queue, starts sending visibility of the player each 200 ms. For spectators an entire game field is sent. Server also starts sending the scores of players periodically in 200ms.

   (b) Client application on user side starts receiver thread. Receiver thread attempts to receive server messages:

      i. If visibility are was received -> update the game field, redraw the graphics

      ii. If score is received -> update the players score list

3. User provides his character class and clicks Join

   (a) Server assigns player's object with defined character class and randomly positions the joined player into the game field.

   (b) Client application on user side now receives only his visible are and not the entire field.

   (c) Client application activates keyboard input processing for user to control his character. Each time control button is hit the suggested 'move' is recorded and store for sending. Each 500ms the suggested 'move' is sent to the server.

   (d) Client handler reads the user's suggested move and modifies his character object.

   (e) Game processor thread periodically (200ms) checks all player objects for suggested 'move' and recomputes the positions of the characters on the game field. If the collisions happen, game engine follows the game logic to score or withdraw the players.

## 2.3 Network messages

Client server are relying on messages of the following format when communicating:

- Client->Server
  - Template
    - control <payload>
  - Registering player:
    - connect <player name string>
  - Joining the game
    - join <character class integer>
  - Player's next move

        * control <next move integer>

- Server->Client
  - Template
    * control <payload>
  - Player's visibility area
    * objects V H <cell:isMe:i:j:name>;<cell:isMe:i:j:name>;<cell:isMe:i:j:name>...
      · V - game field vertical size
      · H - game field horizontal size
      · cell - empty, frog, fly
      · isMe - indicates cell is the player himself
      · i,j - coordinates of the cell
      · name - name of the player
  - Score
    * players <name:points>;<name:points>;<name:points>...
      · name - name of the player
      · points - score of the player

# 3 FCF Game 2.0

In this part we will focus on what needs to be improved. In general we will apply Object Oriented design and replace TCP socket communication with Remote Objects and Remote Method Invocation. The figure 4 illustrates the new design.

## 3.1 Object oriented design

Implement engine part so that it follows properly the object oriented design. Currently there are almost no methods in the Player class, however the Game class contains way to many method and even those not related to game objects. Another thing is that all the players are referred by UUID hence all the player methods carry this UUID as an argument. Why can't we just refer to player by player object, in this case there will be no need for UUID in the methods signature and we could use 'player.get_char()' explicitly on player's object. But we still need UUID for the game field remember ? The game field is MxN array of integers and contains UUID of player for player's position or zero if cell is not occupied. Why can't we just use objects in place of primitives here? Lets say game field is an MxN array of objects and its either instance of Player class or Null if empty. So the first task will be to refactor the engine's code towards object oriented design.

Refactor the code:

- Remove not needed methods

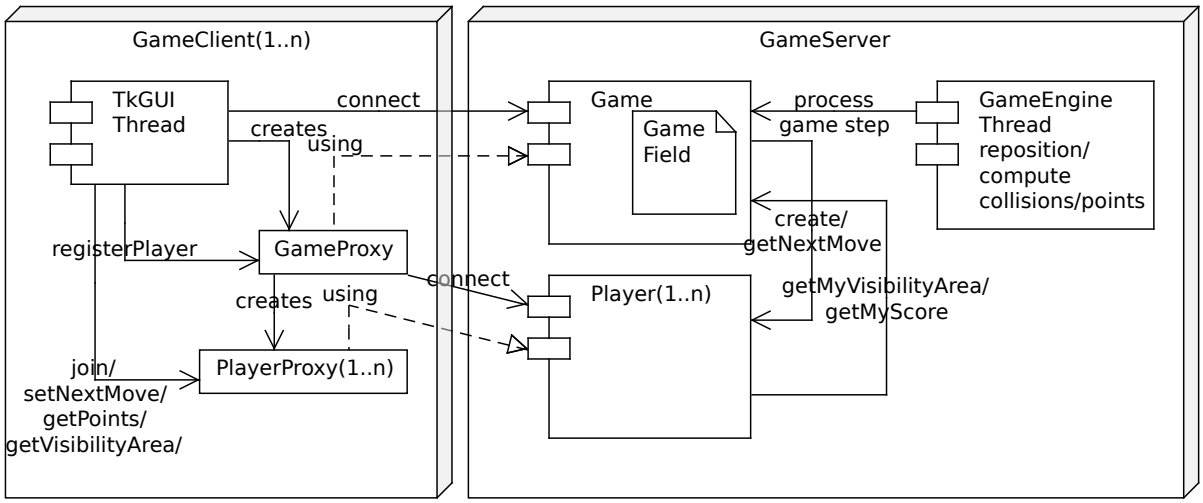


Figure 4: FCF 2.0 Components

- Refer to player by objects (instance of class Player or null). Remove the usage of UUID everywhere.
- Reorganize the methods so that all relevant methods are in corresponding class.
- Remove not needed constants

## 3.2 Remote objects

After we improved the engine API towards object oriented design, the networking part may be reduced drastically if we apply Remote Objects here. We may apply the same Proxy pattern we applied in the Chat application in Remote Method Invocation seminar (ref.). In this case on the server side we create the Game object that is exposed over the network to the clients. When client connects, it obtains a proxy of a the Game object and may now register issuing 'Player register(String name)' method which returns Player proxy object. Now over this proxy object the user may interact we the game. Player can select his class and join the game by calling 'join(Boolean frog)' method. Next move may be specified explicitly calling 'setNextMove' method on player proxy object. The visibility area of the player in respect to his current locations may be obtained by calling 'getVisibilityArea' explicitly on player proxy object. The score of corresponding player may be obtained in the same fashion over 'getScoreMethod'

Now as you can see the old networking part of server and client components become useless, and in fact there should be no calls of socket API left. The client side GUI part however remaining the same, what needs to be done is to properly associate the user's input from the keyboard (up,down,left,right etc.)

to the corresponding calls on player's proxy object. The GUI buttons shall be also properly associated with the game or player objects. Server address has now different format as we use explicitly Game object's - we need to provide it's URI to get it's proxy.

Button associations:

- Connect: takes Game object's URI, connects and retrieves Game object's proxy
- Register: takes Player name, calls game.register(name) retrieves Player object's URI, connects and retrieves player object's proxy
- Join: calls game.join(Boolean frog) method dependent on user's character selection.

# References


[1] Twisted Community. Twisted - event-driven networking engine for Python. http://twistedmatrix.com/trac/, 2015. [Online; accessed 30-Oct-2015].

[2] Artjom Lind. 1st Homework . https://courses.cs.ut.ee/2015/ds/fall/uploads/Main/Homeowrk1.pdf, 2015. [Online; accessed 9-Nov-2015].

[3] David Siroky. snakeMQ - message queuing for Python. http://www.snakemq.net, 2015. [Online; accessed 23-Oct-2015].