Programming Assignment #4 (Java)

CS–671

due 30 Mar 2014 11:59 PM

1. Implement the class `pig.UserTerminalStrategy`. 20 pts
2. Implement the class `pig.server.MatchMaker`. 30 pts
3. Implement the class `pig.server.Message`. 10 pts
4. Implement the class `pig.server.ProtocolToStrategy`. 20 pts
5. Implement the class `pig.server.StrategyToProtocol`. 20 pts

This assignment illustrates client-server programming and uses Java's socket-level networking features. The objective is to implement a server (and its supporting classes) that can be used as a matchmaker to play competitions of a game known as *Pig*. Pig is a two-player game played with one regular 6-face die. A player accumulates points by repeatedly rolling the die and can stop at any time. Rolling a 1, however, loses the turn with no accumulated points. So, the decision after each non-1 roll is between holding (and adding to the total the points accumulated during this turn) or rolling (to try to increase the number of points for the turn). Players take turns until one reaches a total of at least 100 and wins the game.

Central to this implementation of the game is the `Strategy` interface and its two playing methods: `roll` and `opponentPlay`. An implementation of this interface represents an entity that can play the game of Pig: playing decisions are made with method `roll`, and `opponentPlay` is used to analyze the opponent's play. The playing "entity", however, can pretty much be anything: an automated program running locally, a program running remotely, a human player interacting via a local terminal, a human player interacting via a GUI, a remote human player, etc. Anything that implements the `Strategy` interface is a player.

All these implementations of the `Strategy` interface can compete thanks to a `Competition` class that implements a game authority: It rolls the die, informs each player of the decisions of his opponent, and ends the game when a player wins. Furthermore, a `MatchMaker` server allows players to compete remotely. The server forms pairs of players and assigns a game authority to each pair.

The server and the game authorities are not aware of the nature of the players. They interact with them through the `Strategy` interface and have no knowledge of how the interface is implemented. The following classes, in particular, contribute to this opaqueness:

- `UserTerminalStrategy` implements a strategy by interacting with a user via a terminal. It displays on the terminal all the relevant information regarding the game, and the "smarts" come from the flesh-and-blood user who types the answers. Fig. 1 shows an example of such an interaction. The second line, for instance, corresponds to an "implementation" of the `Strategy.startGame` method, which takes a boolean argument (the "not" in "you will *not* play first") and returns a boolean (the "y" typed by the user).
- `ProtocolToStrategy` implements a strategy by writing and reading messages according to a protocol defined below. In this case, the "smarts" come from whoever is at the end of the pipes, man or machine.
- `StrategyToProtocol` implements the reverse functionality: It brings the "smarts" to the pipes. More precisely, it sends and receives messages based on decisions made by its own underlying strategy, which itself can be anything (i.e., any implementation of the `Strategy` interface).

The protocol used by these last two classes is as follows:

- When initially contacted, the server sends a first message `"INIT:..."` with a brief description of the server features. A client must reply with a message `"INIT:name"` that includes his name.
- The server then sends a second message with the list of players currently waiting in the server. This message has the form: `"INIT:p1,p2,p3,..."` with a comma-separated list of all the player names. The client can then do one of two things:

```
You are playing against John.
Do you want to start a game (you will not play first)?  y

Your opponent plays: 6 2 3 6 6.  His/her/its score is now 23.

You roll 5.  Your score is 0+5=5.  What's your decision (r/h)? r
You roll 6.  Your score is 0+11=11.  What's your decision (r/h)? r
You roll 5.  Your score is 0+16=16.  What's your decision (r/h)? r
You roll 2.  Your score is 0+18=18.  What's your decision (r/h)? r
You roll 4.  Your score is 0+22=22.  What's your decision (r/h)? h

Your opponent plays: 6 5 4 1.  His/her/its score stays at 23.

You roll 6.  Your score is 22+6=28.  What's your decision (r/h)? r
You roll 5.  Your score is 22+11=33.  What's your decision (r/h)? r
You roll 2.  Your score is 22+13=35.  What's your decision (r/h)? r
You roll 3.  Your score is 22+16=38.  What's your decision (r/h)? r
You roll 1.  You lose your turn.  Press return to continue.

Your opponent plays: 3 1.  His/her/its score stays at 23.

You roll 4.  Your score is 22+4=26.  What's your decision (r/h)?
...
```

Figure 1: Sample interaction from `UserTerminalStrategy`.

  - send an empty initialization message `"INIT:"`; the client is then added to the list of players waiting in the server, or
  - send a message with the name of another player, of the form `"INIT:p"`, where `"p"` is one of the names from the list obtained from the server. If the pairing fails (player `"p"` has left or has started a different match during the exchange), the server sends an updated list of players present and the process continues. There can be as many rounds of this selection process as the player wishes. At any time, the player can decide to send `"INIT:"` to join the list of players instead of starting a match.

- When a pair is formed (between a player who was waiting in the server and another who chose him), the server sends a message to each player, of the form `"INIT:p"` to player `"q"` and `"INIT:q"` to player `"p"`. This message terminates the initialization stage of the protocol, which then continues as a game-playing protocol.

- At the beginning of each game, the server sends a `START` message to both players: One player gets `"START:YES"` and the other gets `"START:NO"`. The player who gets `"START:YES"` will play first (if the game is played).

- Both players reply with `START` messages: `"START:YES"` if they wish to play the game, `"START:NO"` if they wish to end the match. If at least one player sends `"START:NO"`, the match ends and the connections to the server are closed. If both players send `"START:YES"`, the game is played as a series of rounds.

- In each round, the server sends to the player whose turn it is, a message `"DIE:n"` with the value of a roll of the die. If the value is `"1"`, the round ends (with no points accumulated) and it is the other player's turn to play. Otherwise, the server waits for a decision from the player: `"DECIDE:HOLD"` ends the turn (with points) and `"DECIDE:ROLL"` continues the round with another roll.

- At the end of a round (and unless the game is finished with a score at least equal to 100), the server sends to the other player a message `"DICE:xyz"` with the values just played by his opponent, followed by a message `"DIE:n"` with his own roll of the die. The round continues as before until the server sends `"DIE:1"` or the player sends `"DECIDE:HOLD"`.

- When a player reaches 100, the server sends each player a message `"END:your_score/opponent_score"` where the first score is the score of the recipient of the message. It then sends `START` messages as before (to alternate starting players, the server should send `"START:YES"` to the player who had `"START:NO"` the time before, and

vice-versa). The `END` message is also used when a game is prematurely interrupted (e.g., I/O exception). In this case the message is of the form `"END:x/y"` where both `x` and `y` are less than 100.

One noteworthy peculiarity of this protocol is that a player *does not send a reply to a message* `"DIE:1"`. This is in contrast with method `Strategy.roll`, which returns a useless boolean in this case. Note also that there is no way, for a player who decided to wait for a partner, to leave the server until he is challenged by another player, after which he will have to leave the server and will need to initiate a new connection if he wants to return to the waiting list.

The `MatchMaker` server is multi-threaded: a new thread is created with each incoming connection by a listener thread, whose sole task it is to listen and create these threads. The thread that is associated with a connection runs the first few steps of the protocol (registration and matchmaking). Once a pair is formed, however, only one thread is needed to create a game authority (an instance of `Competition`) and to run it. One possibility is to keep one player thread to run the game and to terminate the other thread. The thread that is terminated is then used as a passive object (it still has the socket needed to communicate with the second player). For instance, a thread can be terminated immediately after the corresponding player enters the waiting area of the server (i.e., after it sends `"INIT:"`). The thread of the player who challenges him will be used to run the game.

Notes:

- Classes to be implemented as well as provided classes and interfaces are described at:

  `http://www.cs.unh.edu/~cs671/Java`

- Source code is provided for `Coin.java`, `Competition.java`, `Die.java`, `TerminalClient.java`, `Strategy.java` and `Utils.java`. In particular, the client side of the initialization protocol is implemented in `TerminalClient` (the client side of the game playing part of the protocol needs to be implemented in `StrategyToProtocol`). These classes don't need to be modified (unless they have bugs).

- Classes `Competition` and `Message` make use of *enums*, a feature introduced in Java 1.5. In `Message` they are used in their simplest form (look at the source code of `Competition` for slightly more advanced usage). Here's the declaration I have in my `Message` class (with the javadoc comments removed):

  ```
  public enum Header {
   ERROR, DECIDE, INIT, DIE, DICE, START, END;
  }
  ```

  Enums are singletons, so `==` can be used for comparison:

  ```
  if (m.header == Message.Header.START) {...}
  ```

  is a valid way to test for a particular header in a message `m`. Enums can also be retrieved by name:

  ```
  Message.Header.valueOf("START") == Message.Header.START
  ```

  is true.

- A correct server is under no obligation to be kind to clients that do not follow the protocol correctly. It is acceptable for your server to close a connection as soon as a client is sending a wrong message (kinder servers could ask the client to retry). However, the server should attempt to send an `ERROR` message with an explanation before closing the connection, so the client understands what went wrong (this will make client debugging easier).

- This assignment is not conceptually difficult. The main difficulty, and it's not negligible, is in testing and debugging. One strategy is to make the server print a short message on the terminal with every event that happens: a message arrives, a connection is opened or closed, a game finishes, a player joins or quits, a thread terminates, ...

- Code like this:

  ```
  } catch (java.io.IOException e) {}
  ```

  is a recipe for disaster in this assignment.

- To facilitate testing and debugging, I will try to keep a server running at all times on `berlioz.cs.unh.edu`, using port 54321. This server is mean (no retries) but sends `ERROR` messages with explanations:

```
> nc berlioz.cs.unh.edu 54321
INIT:Welcome to Pig Server!
INIT:Mike
INIT:Black Jack,Chick,Crazy Joe,Doc,One Round,Zero
START:YES
ERROR:No INIT: or INIT:name message.  Access denied.
```

  Remember that `nc` (or other similar tools) is very handy when debugging a test-based protocol. Use it liberally.

- Once `UserTerminalStrategy` and `StrategyToProtocol` are implemented, `TerminalClient` implements a full terminal-based client program, which can be started this way:

```
> java cs671.pig.server.TerminalClient -name Mike berlioz.cs.unh.edu 54321
```

- This terminal-based client can play a series of game in rapid succession if it is given an automatic strategy. This can be useful to put some stress on the server for debugging purposes. For instance, the command below was used to run 100 games in a few seconds:

```
> java cs671.pig.server.TerminalClient -count 100 -strategy MyAutomaticStrategy \
  berlioz.cs.unh.edu 54321
...
The following players are in the lobby: Black Jack,Chick,Crazy Joe,Doc,One Round,Quit,Zero
Pick one, or press return to join them:  Black Jack
You are playing against Black Jack.
Game 1/100: Black Jack wins.  Score: 100 to 28.
Game 2/100: charpov wins.  Score: 102 to 69.
Game 3/100: charpov wins.  Score: 101 to 48.
Game 4/100: charpov wins.  Score: 100 to 25.
Game 5/100: Black Jack wins.  Score: 100 to 56.
Game 6/100: charpov wins.  Score: 104 to 22.
Game 7/100: charpov wins.  Score: 105 to 21.
Game 8/100: charpov wins.  Score: 101 to 45.
Game 9/100: Black Jack wins.  Score: 102 to 32.
Game 10/100: charpov wins.  Score: 101 to 23.
Game 11/100: charpov wins.  Score: 103 to 49.
...
```

- `UserTerminalStrategy` does not need to behave exactly as in Fig. 1, but it must be usable and display all the relevant information (like intermediate scores).