CS 3331 - Advanced Object-Oriented Programming

HOMEWORK 5: NETWORK PROGRAMMING

This assignment is to done in your designated team. You need to fill out
the contribution form (see the course blackboard).

In this assignment, you are to extend your HW4 code to create the
ultimate version of the Connect Five application. Your application shall
allow two players connected in a peer-to-peer network to solve a
puzzle together. In a p2p network, each device on the network can
function either as a server or a client. The two players communicate
with each other through a TCP/IP socket by following the protocol
specified in APPENDIX (see below).

Your application shall meet all the relevant requirements from previous homework (HW1, HW3, and HW4). In addition, your application shall
provide a way to play against another player.
The specific, new requirements include:

R1. The application shall provide a way to form a peer-to-peer network
    and play a game between the peers. It shall provide a GUI so
    that the user can specify peer's host name (or IP number) and a
    port number.

R2. The application shall request confirmation from the peer when a
     player attempts to join. If the peer accepts request. The Socket
     connection is left open. Also, peers are allowed to chat (see BONUS section).

R3. The application shall display a GREEN/RED status depending on

```
     the status of the connection. By default the status will be
RED since
     when starting Connect Five the player will not be connected
to any peer.

     Once a player accepts a join request both peer and player
     shall display a GREEN status. Once the connection is
terminated by either
     normal or abnormal conditions, the status should go back to
RED.

R4. The application shall request a confirmation from the peer
when a
    player starts a new game. If the peer accepts the request,
the new
    game is started; otherwise, the two players are disconnected
and
    each plays his or her own game, new or current.

R5. The application shall inform the peer when a player
    places a disc in/from the player's board; the
    peer shall update his or her board accordingly.

R6. The application should provide a way to terminate an
existing game and close the connection or
     simply teminate existing connection if a game hasn't been
started.
     When this happens the peer shall be notified in addition to
the status going back to RED.

R7. You should separate network operations into separate modules
to
    decouple them from the rest of the code.

R8. Write HW5 as an extension of HW4. Create new package for
HW5.
     Refactor HW4 if needed. Then, override HW4 methods to
accomodate HW5 requirements.
     See class slides for examples on this.

1. Design your application and document your design by
   drawing a UML class diagram [Chapter 4 of 1]. You should
focus on
   designing those classes that are modified (from your HW4
design) or
   newly introduced; highlight them in your diagram.
   - Your class diagram should show the main components (classes
and
     interfaces) and their relationships.
   - Your model (business logic) classes should be clearly
separated
     from the view/control (UI) classes with no dependencies
[2].
   - For each class in your diagram, define key (public)
operations
     to show its roles or responsibilities in your application.
   - For each association (aggregate and composite), include at
least
     a label, multiplicities and navigation directions.
   - You should provide a short, textual description of each
class
     appearing in your class diagram.

2. Code your design.


BONUS

     Implement a chat between the players after they accept the
join request.

HINTS

   Reuse your HW4 design and code as much as possible.

TESTING

   Your code should compile and run correctly under Java 8 or
later
   versions.

WHAT AND HOW TO TURN IN

   You should submit a single PDF document of your UML diagrams
along with
   accompanying documents on the due date.
   You should submit a single zip file that contains:

   - design.pdf (UML class diagram along with descriptions)
   - contribution-form.docx
```

- hw4.jar, a runnable jar containing bytecode and support files
  (e.g., images and audio clips)
- src directory of source code files

You should submit your program through Blackboard.

The submission page will ask you to zip your program and upload a single zip file. Your zip file should include only a single directory named YourFirstNameLastName containing all your source code files and other support files needed to compile and run your program. DO NOT INCLUDE BYTECODE (.class) FILES. There is a limit on upload file size and the maximum file size is 2MB. You should turn in your programs by 11:59 pm on the due date.

GRADING

You will be graded on the quality of the design and how clear your code is. Excessively long code will be penalized: don't repeat code in multiple places. Your code should be reasonably documented and sensibly indented so it is easy to read and understand.

Be sure your name is in the comments in your code.

REFERENCES

[1] Martina Seidl, et al., UML@Classroom: An Introduction to Object-Oriented Modeling, Springer, 2015. Free ebook through UTEP library.

[2] Holger Gast, How to Use Objects, Addison-Wesley, 2016. Sections 9.1 and 9.2. Ebook available from UTEP library.

APPENDIX

Use the following communication protocol so that your application can

```
work with the applications written by others. You may use the
NetworkAdpater class available from the course website. This
class
provides an abstraction of a socket (TCP/IP and Bluetooth) to
send and
receive Sudoku messages, and it implements all the messages
defined
below.

Peers communicate with each other by sending and receiving
messages
through a socket. Each message is one line of text, a sequence
of
characters ended by the end-of-line character, and consists of a
header and a body. A message header identifies a message type
and ends
with a ":", e.g., "fill:". A message body contains the content
of a
message. If it contains more than one element, they are
separated by a
",", e.g., "1,2,3". There are seven different messages as
defined
below.

  join: -- request to join the peer

  join_ack: n [,s,b] -- acknowledge a join request, where n
(response)
    is either 0 (declined) or 1 (accepted), s is a board size,
and b
    is a sequence of non-empty squares of a board, each encoded
as:
    x,y,v,f (x, y: 0-based column/row indexes, v: contained
value, f:
    1 if the value is given/fixed or 0 if filled by the user.
The
    size (s) and board (b) are required only when n is 1.

  new: s,b -- request to start a new game, where s is a board
size,
    and b is a board encoded in the same way as the join_ack
message.

  new_ack: n -- ack new game request, where n (response) is
    either 0 (declined) or 1 (accepted).
  fill: x, y, v -- fill a square (a disc), where x and y are 0-
based
    column/row indexes of a square and v is a number.

  fill_ack: x, y, v -- acknowledge a fill message.

  quit: -- leaves a game by ending the connection.
```

Two players communicate with each other as follows. One of the
players (client) connects to the other (server) and requests to
join
the current game of the server; the player who initiates the
connection must send a join message, as the other player will be
waiting for it. If the server accepts the request, it sends its
puzzle (board) to the client. Now, both players can solve the
shared
puzzle by sending and receiving a series of fill and fill_ack
messages. A player may quit a shared game or make a request to
play a
new shared game by sending a new puzzle.

```
1. Joining a game (accepted)
   Client    Server
    |------------->|  join: -- request to join
    |<-------------|  join_ack:1,9,0,0,2,1,... -- accept the
request
    |------------->|  fill:3,4,2 -- client fill
    |<-------------|  fill_ack:3,4,2 -- server ack
    |<-------------|  fill:2,3,5 -- server fill
    |------------->|  fill_ack:2,3,5 -- client ack
    ...

2. Joining a game (declined)
   Client    Server
    |------------->|  join: -- request to join
    |<-------------|  join_ack:0 -- decline the request
(disconnected!)


3. Starting a new game (accepted)
   Client    Server
    |------------->|  join: -- request to join
    |<-------------|  join_ack:1,9,0,0,2,1,... -- accept the
request
   ...
    |------------->|  new: 9,1,1,2,1,... -- request for a new game
    |<-------------| new_ack:1 -- accept the request
    |<-------------| fill:3,3,5 -- server fill
    |------------->| fill_ack:3,3,5 -- client ack
    ...

4. Starting a new game (declined)
   Client   Server
    |------------->| join: -- request to join
    |<-------------| join_ack:1,9,0,0,2,1,... -- accept the
request
    ...
    |------------->| new: 9,1,1,2,1,... -- request for a new game
    |<-------------| new_ack:0 -- decline the request
(disconnected!)

5. Quitting a game
   Client   Server
    |------------->| join: -- request to join
    |<-------------| join_ack:1,9,0,0,2,1,... -- accept the
request
    ...
    |------------->| quit: -- quit the game (disconnected!)
```