SETUP INSTRUCTIONS

IDE Used: Eclipse

Setup Instruction

Please import the following Java source files to a new project in Eclipse.

In order to run the program run the source files “B_ServerMain” and “B_ClientMain”. You must run four instances of the client with the server, in order to start a Bomberman game. Test case have been included in either of client and network parts of the program.

Java Source Files:

'package sysc3303.networking':

ClientNetworkHandler.java

DoubleBuffer.java

NetworkHandler.java

Sendable.java

ServerNetworkHandler.java

Subscriber.java

‘package sysc3303.project.BomermanNetworkGame’:

B_ClientNetworkHandler.java

B_NetworkPacket.java

B_NetworkPlayerController.java

B_ServerMain.java

B_ServerNetworkHandler.java

B_TestDriver.java

B_TestServerNetwork.java

'**package sysc3303.project.BombermanGame**':

B_Controller.java

B_Packet.java

B_Player.java

B_PlayerController.java

B_View.java

Bomb.java

Entity.java

GridGenerator.java

PlayerCommand.java

PlayerCommandType.java

PlayerName.java

Point.java

Powerup.java

Utils.java

ViewRenderer.java

World.java

WorldActionOutcome.java

General Description:

- This program uses UDP method of communication.

- Every client send an array of player, command and objects (list of changes to make) to the server. The server goes through changes and applies to world and send a new char[][] (world grid) to all clients that are subscribers.
- Communication protocol: “BombermanServerNetworkHandler” is how the program communicate changes.
- The main threading solution runs inside the “BombermanServerMain”. It creates a logger and creates the thread and joins them inside of it.

Description of Types and their Methods:

Network Handler<Send, Receive>

Receive:
First the server tries to receive from a socket (this process loops forever) and once it receives the data, it stores it in the receive write buffer.

Receive Thread Method:
This method calls preprocessed packet, receives from socket and locks on receive write buffer and writes the data to buffer. This method is synchronized.

Sender Thread Method:
This Method grabs send read buffer and synchronizes on the buffer and parses to turn the sent object into byte array.

Send Data Method:
This method takes an array list of data type send and synchronizes on send write buffer. It loops through data and adds it to the buffer. At the end it swaps the send buffer and takes data from outside class and adds to send write buffer.

Get Data Method:
This is a public method which is called from outside and locks on receive buffer and returns the data. At the end it swaps buffer list.

Bomberman Server Network Handler:

This type adds everything that server handler didn’t add. Type “S” as world and “R” as player command. It also implements all abstract methods.

Bomberman Client Network Handler:

This type extends client network handler by typing “S” for PlayerCommandArray and “R” for char[]][]. This type also implements abstract methods.

Entity
This is the base class for a component with a dynamic position within the world. It has a name, a character and X and Y positions.

Diagrams:

- UML Diagram.pdf
- Diagram.pdf
- Sequence Diagram.pdf

Test Framework:

In order to test the project that was built, the following tests were provided in order to examine the written code.

Whole tests were built on the “junit.Assert” libraries. These libraries were utility classes that assist in validating arguments. This library is useful for identifying programmer errors early and clearly at runtime. Couple of certain methods were used inside each of the tests which are as following:

1- AssertEquals() : This method asserts that two object are equal.
2- AssertNull(): This method asserts that an object is null.
3- AssertArrayEquals(): Asserts that two arrays are the same and if not an AssertionError is thrown.
4- Fail(): It fails a test with no message. Inserted intentionally.
5- AssertSame(): Asserts that two objects refer to the same object.

For each type certain tests were provided in order to examine the written code:

Server concurrency:

1- Network initialization:
   In order to test if the network has been initialized, first we use the methods inside that certain class in order to see whether the network has been initialized. Also we compare the received data with the data that has been gathered from network. This allows us to see first if the data has been sent and two, if they have been fully received (Packets have been dropped or not).

2- Run Test:
   This type of test allows us to run multiple threads in order to see if the server part of the program works when under answering to multiple requests. First we should check whether commands have been received or if the server is waiting for them to arrive. Second, we make sure that the server is accepting requests by not shutting down. Third, it accepts players as well as accepting the commands that each player is sending. Fourth allows us to check whether the position of the players on the map is the same with the movement that they have made. Finally we check whether the received data is the same as the grabbed data from the network. It also checks whether the server is capable of actually updating and stopping the network. The whole purpose of this test was to see how the server handles multiple threads. If it is one, then it just gets executed after being read from the buffer. If there is more than one, then the order of the execution will be First In First Out (FIFO). Handling of the concurrency of the server will be based on which package has priority over the other ones.

NetworkHandler:

1- Sending/Receiving:
   In order to see if the network part of our program is starting, first we have to check whether our sending and receiving threads are up and running and second we have to check that

they are being sent and receive continuously. In other methods we check for packets of data to being parsley sent and received and they should be checked whether they match the returning data. In the last step we check for sending and receiving methods and whether the packets are not null and if the preprocessed Packets match the type that are being needed.

DoubleBuffer:

1- Swapping Buffer:
Since the reading and writing buffers are being read in order (First they write into the buffer, then they swap and the system reads the buffer). In order to do this part we had to make sure that the assertion for equality of buffers fail since they should not be the same thing. If this test fails it shows that the swapping actually works between two buffers (and when one gets read it gets flushed so that it would be available to store data again) and it does the reading and writing in order on different buffers. Synchronization is being used in the code to not allow the corruption of data while one is being read and the other gets written to. This issue also arises from the fact that ArrayDeques can grow extensively and don’t have any size limits.

2- Testing reading and writing on buffers:

In order to test reading and writing on buffers we have to check whether buffers (both reading and writing) actually match the data that is being returned. By concatenating the read and write in an array in order and comparing this final array with the return value of our data we can conclude whether the sent and received data are the same as the data that is being processed. Also by checking the sent data and that the data that is being written (and vice versa, by checking the receiving data and the data that is being sent) we can see that whether they are equal and conclude the fact that buffers work perfectly.