Tests for Milestone 2 - Bomberman

All tests are done using JUnit as unit tests. Every test runs independently of one-another, meaning that each small functionality of the game can be tested individually. This has given us a fine grain of control over the tests. You can see all of the game functionality tests below.

TestClientServerGameScenarios.java

testClientConnectAsPlayer() - connect as a player. Ensure that connection succeeds.
testClientConnectTwoPlayers() - Connect two players. Ensure connection success.
testClientConnectThreePlayers() - Only two players should be able to connect, not three.
testClientConnectFourPlayers() Only two players should be able to connect, not four.
testClientLoadBoard() - load a game from file and test that load succeeded.
testClientLoadBoardStartGame() - start game after loading board to ensure game is one that was loaded
testClientMoveDown() - send the command for move down, check board state to ensure that player moved down in the game.
testClientMoveUp() - send the command for move up, check board state to ensure that player moved up in the game.
testClientMoveRight() - send the command for move right, check board state to ensure that player moved right in the game.
testClientMoveLeft() - send the command for move left, check board state to ensure that player moved left in the game.
testClientExitDoor() - exit the door in game and ensure that game is finished.
testClientMoveDownIntoBox() - move down into box and ensure that you dont move.
testClientMoveIntoBoardLimitDown() - test to ensure you cannot go down over board boundaries
testClientMoveIntoBoardLimitLeft() - test to ensure you cannot go left over board boundaries
testClientMoveIntoBoardLimitUp() - test to ensure you cannot go up over board boundaries
testClientMoveIntoBoardLimitRight() - test to ensure you cannot go right over board boundaries
testClientPickUpItem() - ensure that you pick up item and item is removed from board
testClientPickUpPowerup() - ensure that when you pick up powerup, you gain a powerup as a player
testClientTwoPlayersColliding() - ensure that two players colliding ends the game
testClientDeployBomb() - deploy bomb and run away and see that it kills the enemies around the bomb
testClientBombKillOtherPlayerGameOver() - deploy bomb and run away, killing other player. Player should be removed from the board.
testClientPlayerTwoBombAvoid() - deploy bomb in range of killing player, run away and ensure that both players are safe after bomb deploys.
testRunIntoEnemyGameOver() - run into an enemy. You should die and game should be over.

TestGame.java

testGameStarted() - Check to ensure that a game can be started.
testAddOnePlayer() - Check to ensure that one player can be added to a game.
testAddOnePlayerArraySize() - Check to ensure that when one player is playing, the player array is of size 1.
testAddTwoPlayers() - Check to ensure that you can add two players to a game.
testAddTwoPlayersArraySize() - Check to ensure that when two players are playing, the player array is of size 2.
testGetBufferStateTwice() - Check the double buffer state. This ensures that the double buffer maintains a proper state of the system at all times, regardless of which buffer is being used.

TestServer.java

testServerIsRunning() - check to ensure that the server is running once we have started it.
testServerPort() - check to ensure that the server is bound to the proper port.
testServerEmptyMessageQueue() - check to ensure that a server has an empty message queue when started.

Functionality Tests

The functional tests can be seen in a high level above. These tests test the core functionality of the game and also test concurrency (bomb detonation, double buffer switch). You can read their descriptions above. Notable tests include testRunIntoEnemyGameOver() to test that a player dies when they run into an enemy and

Concurrency Tests

Testing the concurrency features of the server and client is easy in our system. We currently have tests in place that ensure a broad range of scenarios, including bomb detonation and players moving concurrently. Our system uses a double buffer to hold the state of the game in a non-volatile variable whenever the game is being read by the broadcaster. To update the game state, the game controller will update the double buffer's unused non-volatile variable of the game state and then switch the double buffer to now allow reads from that variable, which no longer can be updated. This ensures that our game state can always be read from a constant value and is not being concurrently updated by another thread. This also allows such modfications to not be blocked by a read, which will happen quite frequently from the broadcaster.

Tests are in place to ensure that bomb detonation, game state update changes and double buffer switch. Notable tests are testGetBufferStateTwice() that ensures that the buffer is updated properly.

Scalability Tests

Scalability is easy for our system. We can just change some constants to scale the number of players possible from 2 to 4. The game functions perfectly with more players however we would have to adjust parts of our test plan so that our test cases pass since currently we test to ensure that adding more than two players fails. Those test cases would need to be re-factored such that more than 4 players failed.

To adjust the rate at which the server broadcasts the messages to the client, we just adjust one variable. Currently our system broadcasts the messages at 100ms intervals, but to change the rate to every 10ms we just change one number from 100 to 10. Our testing showed that a broadcast rate of 100ms was fast enough to make the interaction seamless for human players. When we implement an AI it will have it's own update rate that is independent of the broadcast rate and will inherently be larger than 100ms in order to simulate actual interactions by a human with the computer. To make it "harder" we can increase their reaction rate (aka lower the time they sleep between updates, always ensuring this number is still greater than the broadcast rate), this element of the game is independant of the broadcast rate. :)

If we have AI players that are sending messages to the server very quickly, they are limited by the speed at which the server broadcasts the game state, so the rate is a limiting factor. For this milestone (Milestone 2) we do not have an AI implemented for players or enemies, so the rate is not an issue. For 2 players, our system works well.