試験報告書

プロジェクト 1　グループ 9

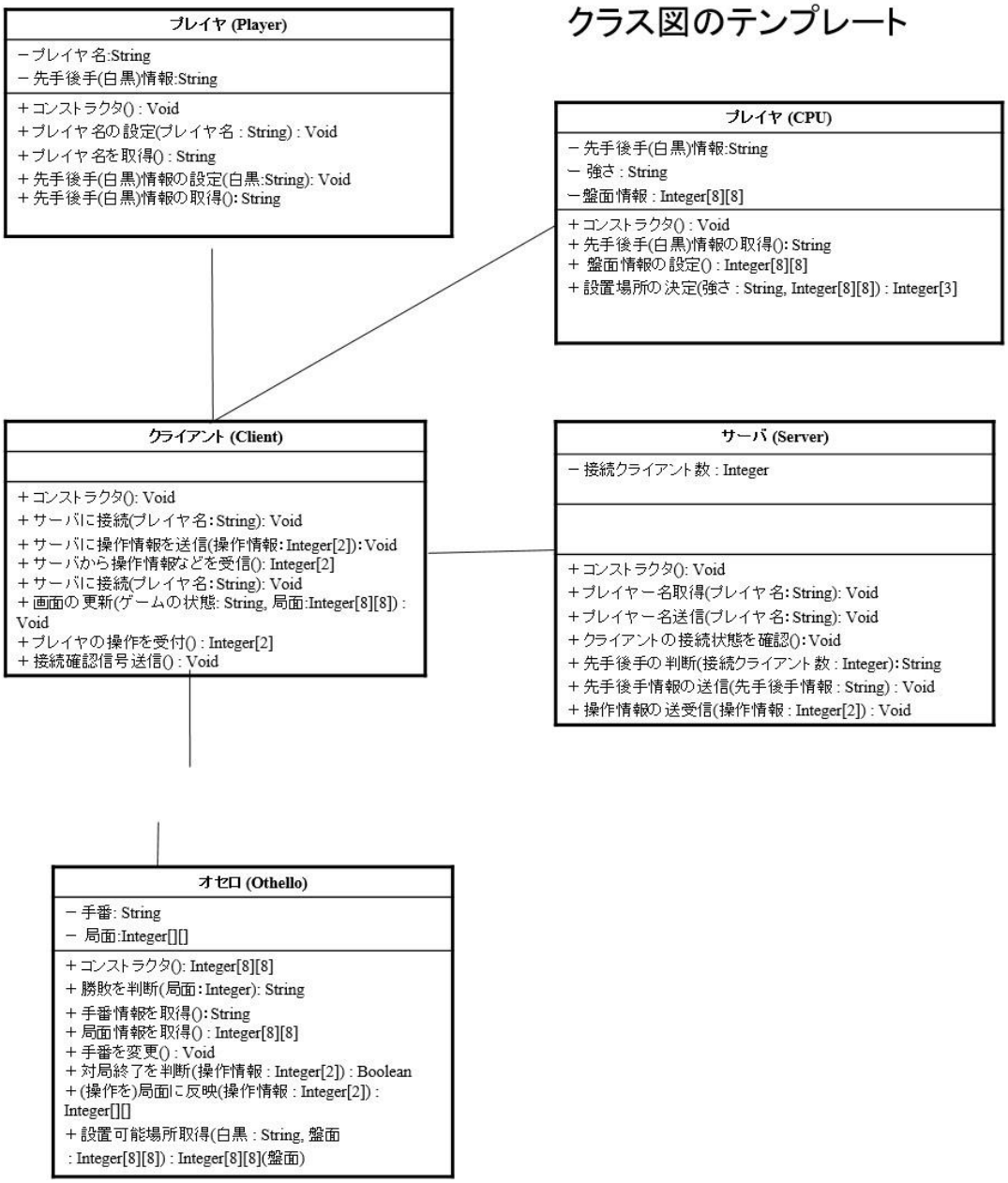
提出日：2025 年 5 月 23 日

1　はじめに

本報告書は，ネットワーク対戦型リバーシゲームシステムの試験方法および試験結果を記すものである．試験対象であるシステムのソースコードおよびクラス図は以下の通りである．

ソースコード一覧：

- Player.java　PlayerDriver.java（テスト用ドライバ）
- Client.java　ClientTestDriver.java（テスト用ドライバ）
- Othello.java OthelloDriver.java（テスト用ドライバ）
- Server.java　ServerDriver.java（テスト用ドライバ）
- CPU.java CPUDriver.java（テスト用ドライバ）
- View.java ViewDriver.java（テスト用ドライバ）



2　単体テスト

2.1 Player クラス

以下のドライバ（PlayerDriver.java）を用いて Player クラスの単体テストを行った。なお、スタブは使用していない。

```
import java.util.Scanner;

import java.lang.System;

public class PlayerDriver {

    public static void main(String[] args) {
```

```
System.out.println("Player クラスの単体テストを開始します。");

// Player オブジェクトの作成

Player player = new Player();

System.out.println("¥n1. Player オブジェクトの作成テスト");

if (player != null) {

    System.out.println("    Player オブジェクトが正常に作成されました。");

} else {

    System.out.println("    Player オブジェクトの作成に失敗しました。");

}

// プレイヤ名の設定と取得テスト

String testName = "電情太郎";

player.setPlayerName(testName);

System.out.println("¥n2. プレイヤ名の設定（setPlayerName）と取得（getPlayerName）テスト");

System.out.println("    設定したプレイヤ名: " + testName);

System.out.println("    取得したプレイヤ名: " + player.getPlayerName());

if (testName.equals(player.getPlayerName())) {

    System.out.println("    プレイヤ名の設定と取得が正常に動作しています。");

} else {

    System.out.println("    プレイヤ名の設定と取得に問題があります。");

}

// 先手後手(白黒)情報の設定と取得テスト

String testColorBlack = "黒";

player.setStoneColor(testColorBlack);

System.out.println("¥n3. 石の色の設定（setStoneColor）と取得（getStoneColor）テスト - 黒");

System.out.println("    設定した石の色: " + testColorBlack);

System.out.println("    取得した石の色: " + player.getStoneColor());

if (testColorBlack.equals(player.getStoneColor())) {

    System.out.println("    石の色の設定と取得が正常に動作しています（黒）。");

} else {
```

```
        System.out.println("    石の色の設定と取得に問題があります（黒）。");
    }
    String testColorWhite = "白";
    player.setStoneColor(testColorWhite);
    System.out.println("¥n4. 石の色の設定（setStoneColor）と取得（getStoneColor）テスト - 白");
    System.out.println("    設定した石の色: " + testColorWhite);
    System.out.println("    取得した石の色: " + player.getStoneColor());
    if (testColorWhite.equals(player.getStoneColor())) {
        System.out.println("    石の色の設定と取得が正常に動作しています（白）。");
    } else {
        System.out.println("    石の色の設定と取得に問題があります（白）。");
    }
    // 相手の色取得テスト
    System.out.println("¥n5. 相手の色の取得（getOpponentColor）テスト");
    player.setStoneColor("黒");
    System.out.println("    自分の石の色が「黒」の場合、相手の色: " + player.getOpponentColor());
    if ("白".equals(player.getOpponentColor())) {
        System.out.println("    「黒」に対する相手の色が正常に取得されました。");
    } else {
        System.out.println("    「黒」に対する相手の色取得に問題があります。");
    }
    player.setStoneColor("白");
    System.out.println("    自分の石の色が「白」の場合、相手の色: " + player.getOpponentColor());
    if ("黒".equals(player.getOpponentColor())) {
        System.out.println("    「白」に対する相手の色が正常に取得されました。");
    } else {
        System.out.println("    「白」に対する相手の色取得に問題があります。");
    }
    player.setStoneColor("不正な色");
```

```
        System.out.println("    自分の石の色が「不正な色」の場合、相手の色: " + player.getOpponentColor());

        if ("?".equals(player.getOpponentColor())) {

            System.out.println("    不正な色に対する相手の色が正常に処理されました。");

        } else {

            System.out.println("    不正な色に対する相手の色処理に問題があります。");

        }

        System.out.println("¥nPlayer クラスの単体テストが完了しました。");

    }

}
```

試験結果 ：

```
PS C:\Users\Misaki Nishimura\code\YNU_Othello\src> java PlayerDriver
Playerクラスの単体テストを開始します。

1. Playerオブジェクトの作成テスト
   Playerオブジェクトが正常に作成されました。

2. プレイヤ名の設定 (setPlayerName) と取得 (getPlayerName) テスト
   設定したプレイヤ名: 電情太郎
   取得したプレイヤ名: 電情太郎
   プレイヤ名の設定と取得が正常に動作しています。

3. 石の色の設定 (setStoneColor) と取得 (getStoneColor) テスト - 黒
   設定した石の色: 黒
   取得した石の色: 黒
   石の色の設定と取得が正常に動作しています (黒)。

4. 石の色の設定 (setStoneColor) と取得 (getStoneColor) テスト - 白
   設定した石の色: 白
   取得した石の色: 白
   石の色の設定と取得が正常に動作しています (白)。

5. 相手の色の取得 (getOpponentColor) テスト
   自分の石の色が「黒」の場合、相手の色: 白
   「黒」に対する相手の色が正常に取得されました。
   自分の石の色が「白」の場合、相手の色: 黒
   「白」に対する相手の色が正常に取得されました。
   自分の石の色が「不正な色」の場合、相手の色: ?
   不正な色に対する相手の色が正常に処理されました。

Playerクラスの単体テストが完了しました。
```

2.2 Othello クラス

以下のドライバ（OthelloDriver. java）を用いて Othello クラスの単体テストを行った。なお、スタブは使用していない。

```
import java.util.Scanner;
```

```java
public class OthelloDriver {

    public static final int SIZE = 8; // 盤面のサイズ

    final static int EMPTY = 0; // 設置されていない

    final static int BLACK = 1; // 黒の石が置かれている

    final static int WHITE = 2; // 白の石が置かれている

    final static int CANPLACE = 3; // 設置可能


    public static void main(String[] args) {

        Integer[][] board = new Integer[SIZE][SIZE];

        Integer[][] validMoves = new Integer[SIZE][SIZE];

        String turn = "";

        Scanner scanner = new Scanner(System.in);


        System.out.println("Othello クラスのテスト(OthelloDriver.java)");


        // 盤面の初期化

        Othello.initBoard(board);

        // プレイヤーの初期化

        turn = Othello.initTurn(turn);


        while (true) {

            System.out.println(turn + "の番です");


            // 有効な手の取得

            validMoves = Othello.getValidMovesBoard(board, turn);


            // 盤面の表示

            printBoard(Othello.getBoard(board, validMoves));

            // 諸情報の表示
```

```
            System.out.println("諸情報の表示");
            System.out.println("自分の手番: " + turn);
            System.out.println("相手の手番: " + Othello.opponentTurn(turn));
            System.out.println("現在の石の数（相手）: " + Othello.numberOfStone(board,
Othello.getStoneColor(Othello.opponentTurn(turn))));
            System.out.println("現在の石の数（自分）: " + Othello.numberOfStone(board,
Othello.getStoneColor(turn)));

            // 有効手があるかどうかチェック
            if (!Othello.hasValidMove(board, turn)) {
                System.out.println(turn + "はパスします。");
                turn = Othello.changeTurn(turn);

                // 相手もパスなら終了
                if (!Othello.hasValidMove(board, turn)) {
                    System.out.println("両者ともパス。ゲーム終了！");
                    break;
                }
                continue;
            }

            // 入力受付
            int x = -1, y = -1;
            while (true) {
                try {
                    System.out.print("x 座標（0-7）を入力: ");
                    x = scanner.nextInt();
                    System.out.print("y 座標（0-7）を入力: ");
                    y = scanner.nextInt();
```

```
                    if (!Othello.isValidMove(board, x, y, turn)) {
                        System.out.println("その位置には置けません。別の場所を選んでください。");
                        continue;
                    }

                    // 有効な手が選ばれた場合、ループを抜ける
                    break;
                } catch (Exception e) {
                    System.out.println("無効な入力です。もう一度試してください。");
                    scanner.nextLine(); // 入力バッファのクリア
                }
            }

            // 石を置く
            Othello.makeMove(board, x, y, turn);

            // 手番交代
            turn = Othello.changeTurn(turn);
        }

        // 結果表示
        printBoard(board);
        System.out.println("ゲーム終了");
        System.out.println("現在の石の数（相手）：" + Othello.numberOfStone(board,
Othello.getStoneColor(Othello.opponentTurn(turn))));
        System.out.println("現在の石の数（自分）：" + Othello.numberOfStone(board,
Othello.getStoneColor(turn)));
        String winner = Othello.judgeWinner(board);
```

```
        System.out.println("勝者: " + winner);


        scanner.close();
    }


    // テスト用盤面表示メソッド
    public static void printBoard(Integer[][] currentBoard) {


        System.out.println("現在の盤面:");


        // 盤面の表示
        System.out.print("  ");
        for (int i = 0; i < SIZE; i++)
            System.out.print(i + " ");
        System.out.println();


        for (int i = 0; i < SIZE; i++) {
            System.out.print(i + " ");
            for (int j = 0; j < SIZE; j++) {
                if (currentBoard[i][j] == CANPLACE) {
                    System.out.print("◎" + " ");
                } else if (currentBoard[i][j] == BLACK) {
                    System.out.print("○" + " ");
                } else if (currentBoard[i][j] == WHITE) {
                    System.out.print("●" + " ");
                } else if (currentBoard[i][j] == EMPTY) {
                    System.out.print("  " + " ");
                } else {
                    System.out.print("?");
```

```
                }

            }

            System.out.println();

        }

        System.out.println();

    }

}
```

試験結果 ：

2.3 Client クラス

以下のドライバ（ClientTestDriver.java）を用いて Player クラス(ただし、メソッドを一時的に public に設定)の単体テストを行った。なお、他のクラス(Othello, Player, CPU, View)が完全な実装として利用し、スタブは使用していない。

```java
import java.util.Arrays;



// Client.java のメソッドや必要なフィールドがテスト用に public になっているを想定

// Othello, Player, CPU, View クラスが完全な実装として利用可能であるを想定



public class ClientTestDriver {



    // これらの定数は Client.java のもの

    private static final Integer SIZE = 8;

    private static final Integer EMPTY = 0;

    private static final Integer BLACK = 1;

    private static final Integer WHITE = 2;



    private Client client;

    private View view; // View の実オブジェクト



    // テスト結果のカウンター

    private static int testsPassed = 0;

    private static int testsFailed = 0;

    private static int testsRun = 0;
```

```
    // 手動セットアップ（@BeforeEach に相当）
    public void setUp() {
        view = new View(); // View の実オブジェクト
        client = new Client(view, "localhost", 10000);
        view.setClient(client);
    }

    // 手動ティアダウン（@AfterEach に相当）
    public void tearDown() {
        if (client != null) {
            client.shutdown();
        }
        if (view != null && view instanceof java.awt.Window) {
            ((java.awt.Window) view).dispose();
        }
    }

    // --- カスタムアサーションヘルパー ---
    private void assertTrue(String message, boolean condition) {
        testsRun++;
        if (condition) {
```

```
            System.out.println("成功: " + message);
            testsPassed++;
        } else {
            System.err.println("失敗: " + message);
            testsFailed++;
        }
    }

    private void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }

    private void assertEquals(String message, Object expected, Object actual) {
        testsRun++;
        if (expected == null && actual == null || (expected != null && expected.equals(actual)))
{
            System.out.println("成功: " + message + " (期待値: " + expected + ", 実際値: " + actual
+ ")");
            testsPassed++;
        } else {
            System.err.println("失敗: " + message + " (期待値: " + expected + ", 実際値: " + actual
```

```
+ ")");
            testsFailed++;
        }
    }

    private void assertNotNull(String message, Object object) {
        testsRun++;
        if (object != null) {
            System.out.println("成功: " + message);
            testsPassed++;
        } else {
            System.err.println("失敗: " + message + " (オブジェクトが null でした)");
            testsFailed++;
        }
    }

    private void assertNotSame(String message, Object unexpected, Object actual) {
        testsRun++;
        if (unexpected != actual) {
            System.out.println("成功: " + message);
            testsPassed++;
```

```
        } else {
            System.err.println("失敗: " + message + " (オブジェクトが同じインスタンスでした)");
            testsFailed++;
        }
    }

    private void assertArrayEquals(String message, Object[] expected, Object[] actual) {
        testsRun++;
        if (Arrays.equals(expected, actual)) {
            System.out.println("成功: " + message);
            testsPassed++;
        } else {
            System.err.println("失敗: " + message + " (期待値: " + Arrays.toString(expected) + ",
実際値: " + Arrays.toString(actual) + ")");
            testsFailed++;
        }
    }


    // --- テストメソッド ---
    public void testClientConstructorInitialization() {
```

```
        System.out.println("¥n テスト実行中: testClientConstructorInitialization...");
        setUp();
        try {
            assertNotNull("盤面の状態が初期化されている。", client.boardState);
            assertEquals("盤面が正しいサイズである。", SIZE, client.boardState.length);
            assertEquals("初期盤面 [3][3]", WHITE, client.boardState[3][3]);
            assertEquals("初期盤面 [3][4]", BLACK, client.boardState[3][4]);
            assertEquals("初期盤面 [4][3]", BLACK, client.boardState[4][3]);
            assertEquals("初期盤面 [4][4]", WHITE, client.boardState[4][4]);

            assertNotNull("人間プレイヤーが初期化されている。", client.getHumanPlayer());
            assertNotNull("対戦相手プレイヤーが初期化されている。",
client.getCurrentOpponentPlayer());
            assertEquals("サーバーアドレス", "localhost", client.getServerAddress());
            assertEquals("サーバーポート", 10000, client.getServerPort());
            assertFalse("ゲームが初期状態ではアクティブでない。", client.gameActive);
        } catch (Exception e) {
            System.err.println("testClientConstructorInitialization でエラー: " + e.getMessage());
            e.printStackTrace();
            testsFailed++; // エラーを失敗としてカウント
        } finally {
```

```
            tearDown();
        }
    }

    public void testToOthelloColor() {
        System.out.println("¥n テスト実行中: testToOthelloColor...");
        setUp(); // 非 static メソッドのため Client インスタンスが必要
        try {
            assertEquals("'黒'からオセロの色へ変換", "Black", client.toOthelloColor("黒"));
            assertEquals("'白'からオセロの色へ変換", "White", client.toOthelloColor("白"));
        } catch (Exception e) {
            System.err.println("testToOthelloColor でエラー: " + e.getMessage());
            e.printStackTrace();
            testsFailed++;
        } finally {
            tearDown();
        }
    }

    public void testFromOthelloColor() {
        System.out.println("¥n テスト実行中: testFromOthelloColor...");
```

```
        setUp();
        try {
            assertEquals("オセロの色'Black'から変換", "黒", client.fromOthelloColor("Black"));
            assertEquals("オセロの色'White'から変換", "白", client.fromOthelloColor("White"));
        } catch (Exception e) {
            System.err.println("testFromOthelloColor でエラー: " + e.getMessage());
            e.printStackTrace();
            testsFailed++;
        } finally {
            tearDown();
        }
    }


    public void testCopyBoard() {
        System.out.println("¥n テスト実行中: testCopyBoard...");
        setUp();
        try {
            Integer[][] originalBoard = new Integer[SIZE][SIZE];
            for (int i = 0; i < SIZE; i++) {
                Arrays.fill(originalBoard[i], EMPTY);
            }
```

```
            originalBoard[0][0] = BLACK;



            Integer[][] copiedBoard = client.copyBoard(originalBoard);



            assertNotSame("コピーされた盤面は新しいオブジェクトである。", originalBoard, copiedBoard);

            assertNotSame("コピーされた盤面の各行は新しい配列である。", originalBoard[0],
copiedBoard[0]);

            assertArrayEquals("コピー後、行の内容が同一である。", originalBoard[0], copiedBoard[0]);

            assertEquals("コピーされた盤面の要素を確認", BLACK, copiedBoard[0][0]);



            copiedBoard[0][0] = WHITE;

            assertEquals("コピーを変更した後、元の盤面が変更されていない。", BLACK,
originalBoard[0][0]);

        } catch (Exception e) {

            System.err.println("testCopyBoard でエラー: " + e.getMessage());

            e.printStackTrace();

            testsFailed++;

        } finally {

            tearDown();

        }

    }
```

```
    public void testStartGame_CPU_HumanBlack() {
        System.out.println("¥n テスト実行中: testStartGame_CPU_HumanBlack...");
        setUp();
        try {
            client.startGame(true, "黒", "Easy", null);

            assertTrue("ゲーム開始後、ゲームがアクティブである。", client.gameActive);
            assertFalse("CPU 対戦である。", client.isNetworkMatch());
            assertEquals("人間プレイヤー名", "You", client.getHumanPlayer().getPlayerName());
            assertEquals("人間プレイヤーの石の色", "黒", client.getHumanPlayer().getStoneColor());
            assertEquals("対戦相手名", "CPU (Easy)", client.opponentName);
            assertEquals("対戦相手プレイヤーの石の色", "白",
client.getCurrentOpponentPlayer().getStoneColor());
            assertNotNull("CPU の思考エンジンが初期化されている。", client.cpuBrain);
            assertNotNull("CPU 実行サービスが初期化されている。", client.cpuExecutor);
            assertEquals("現在の手番が黒である。", "黒", client.currentTurn);
        } catch (Exception e) {
            System.err.println("testStartGame_CPU_HumanBlack でエラー: " + e.getMessage());
            e.printStackTrace();
            testsFailed++;
```

```
        } finally {
            tearDown();
        }
    }

    public void testStartGame_CPU_HumanWhite_CPUMakesFirstMove() {
        System.out.println("¥n テスト実行中: testStartGame_CPU_HumanWhite_CPUMakesFirstMove...");
        setUp();
        try {
            client.startGame(true, "白", "Easy", null); // 人間が白、CPU（黒）が先手

            assertTrue("ゲームアクティブチェック", client.gameActive);
            assertEquals("人間プレイヤーの色", "白", client.getHumanPlayer().getStoneColor());
            assertEquals("CPU プレイヤーの色", "黒",
client.getCurrentOpponentPlayer().getStoneColor());
            assertEquals("初期手番（CPU 黒）", "黒", client.currentTurn);

            System.out.println("testStartGame_CPU_HumanWhite: CPU の最初の着手を待機中（2 秒)...");
            Thread.sleep(2000); // CPU が着手するのを待つ（不安定な可能性あり）

            assertEquals("CPU の着手後の手番（人間 白）", "白", client.currentTurn);
```

```
            int pieceCount = 0;
            for (int i = 0; i < SIZE; i++) {
                for (int j = 0; j < SIZE; j++) {
                    if (client.boardState[i][j] != EMPTY) pieceCount++;
                }
            }
            assertTrue("CPU の着手後、盤面の石の数が 4 より大きい", pieceCount > 4);
        } catch (InterruptedException e) {
            System.err.println("testStartGame_CPU_HumanWhite_CPUMakesFirstMove で Thread.sleep が
中断されました: " + e.getMessage());
            testsFailed++;
        } catch (Exception e) {
            System.err.println("testStartGame_CPU_HumanWhite_CPUMakesFirstMove でエラー: " +
e.getMessage());
            e.printStackTrace();
            testsFailed++;
        } finally {
            tearDown();
        }
    }
```

```
    public void testHandleCpuPlayerMove_ValidMove() {

        System.out.println("¥n テスト実行中: testHandleCpuPlayerMove_ValidMove...");

        setUp();

        try {

            client.startGame(true, "黒", "Easy", null); // 人間が黒

            assertTrue("ゲームアクティブ", client.gameActive);

            assertEquals("現在の手番は黒", "黒", client.currentTurn);



            int testRow = 2, testCol = 3;

            boolean isValid = Othello.isValidMove(client.boardState, testRow, testCol, "Black");

            assertTrue("(2,3) は初期状態で黒の有効な手である。", isValid);



            if (isValid) {

                client.handlePlayerMove(testRow, testCol);

                assertEquals("マス (" + testRow + "," + testCol + ") が黒である。", BLACK,
client.boardState[testRow][testCol]);

                assertEquals("マス (3,3) が黒に反転している。", BLACK, client.boardState[3][3]);

                assertEquals("手番が白（CPU）に切り替わる。", "白", client.currentTurn);

            } else {

                System.err.println("有効と想定した手が有効でなかったため、
testHandleCpuPlayerMove_ValidMove の一部をスキップしました。");
```

```
            testsFailed++; // または特定のテスト失敗として処理
        }
    } catch (Exception e) {
        System.err.println("testHandleCpuPlayerMove_ValidMove でエラー: " + e.getMessage());
        e.printStackTrace();
        testsFailed++;
    } finally {
        tearDown();
    }
}

public void testShutdownLogic() {
    System.out.println("¥n テスト実行中: testShutdownLogic...");
    setUp();
    try {
        client.startGame(true, "黒", "Easy", null);
        client.gameActive = true;
        assertNotNull("シャットダウン前の CPU 実行サービス", client.cpuExecutor);

        client.shutdown();
```

```
            assertFalse("シャットダウン後、ゲームが非アクティブである。", client.gameActive);
            if (client.cpuExecutor != null) {
                assertTrue("CPU 実行サービスがシャットダウンされている。",
client.cpuExecutor.isShutdown());
            }
            // ハートビート実行サービスはネットワーク対戦でのみアクティブ。
            // client.socket が null でない場合、client.socket.isClosed()を確認するも可能
        } catch (Exception e) {
            System.err.println("testShutdownLogic でエラー: " + e.getMessage());
            e.printStackTrace();
            testsFailed++;
        } finally {
            // tearDown() は既に呼び出されているが、client.shutdown() がテスト対象のメソッド
        }
    }



    // --- テスト実行用 main メソッド ---
    public static void main(String[] args) {
        System.out.println("手動クライアントテストドライバーを開始します...");
        ClientTestDriver driver = new ClientTestDriver();
```

```
        // テストメソッドの呼び出し
        driver.testClientConstructorInitialization();
        driver.testToOthelloColor();
        driver.testFromOthelloColor();
        driver.testCopyBoard();
        driver.testStartGame_CPU_HumanBlack();
        driver.testStartGame_CPU_HumanWhite_CPUMakesFirstMove(); // このテストは Thread.sleep() を
使用します
        driver.testHandleCpuPlayerMove_ValidMove();
        driver.testShutdownLogic();

        System.out.println("¥n-----------------------------結果-----------------------------
-");
        System.out.println("実行テスト総数: " + testsRun);
        System.out.println("成功: " + testsPassed);
        System.out.println("失敗: " + testsFailed);
        System.out.println("------------------------------------------------------------------
---");

        if (testsFailed > 0) {
```

```
            System.out.println("¥n いくつかのテストが失敗しました！");

        } else if (testsRun > 0) {

            System.out.println("¥n 実行されたすべてのテストが成功しました！");

        } else {

            System.out.println("¥n テストは実行されませんでした。");

        }

    }

}
```

```
PS C:\Users\Misaki Nishimura\code\YNU_Othello\src> java ClientTestDriver
手動クライアントテストドライバーを開始します...

テスト実行中: testClientConstructorInitialization...
Board is initialized.
成功: 盤面の状態が初期化されている。
成功: 盤面が正しいサイズである。 (期待値: 8, 実際値: 8)
成功: 初期盤面 [3][3] (期待値: 2, 実際値: 2)
成功: 初期盤面 [3][4] (期待値: 1, 実際値: 1)
成功: 初期盤面 [4][3] (期待値: 1, 実際値: 1)
成功: 初期盤面 [4][4] (期待値: 2, 実際値: 2)
成功: 人間プレイヤーが初期化されている。
成功: 対戦相手プレイヤーが初期化されている。
成功: サーバーアドレス (期待値: localhost, 実際値: localhost)
成功: サーバーポート (期待値: 10000, 実際値: 10000)
成功: ゲームが初期状態ではアクティブでない。

テスト実行中: testToOthelloColor...
Board is initialized.
成功: '黒'からオセロの色へ変換 (期待値: Black, 実際値: Black)
成功: '白'からオセロの色へ変換 (期待値: White, 実際値: White)

テスト実行中: testFromOthelloColor...
Board is initialized.
成功: オセロの色'Black'から変換 (期待値: 黒, 実際値: 黒)
成功: オセロの色'White'から変換 (期待値: 白, 実際値: 白)

テスト実行中: testCopyBoard...
Board is initialized.
成功: コピーされた盤面は新しいオブジェクトである。
成功: コピーされた盤面の各行は新しい配列である。
成功: コピー後、行の内容が同一である。
成功: コピーされた盤面の要素を確認 (期待値: 1, 実際値: 1)
成功: コピーを変更した後、元の盤面が変更されていない。 (期待値: 1, 実際値:

テスト実行中: testStartGame_CPU_HumanBlack...
Board is initialized.
Board is initialized.
CPU: turn = White, level = Easy, depth = 4, threshold = 8
成功: ゲーム開始後、ゲームがアクティブである。
成功: CPU対戦である。
成功: 人間プレイヤー名 (期待値: You, 実際値: You)
成功: 人間プレイヤーの石の色 (期待値: 黒, 実際値: 黒)
成功: 対戦相手名 (期待値: CPU (Easy), 実際値: CPU (Easy))
成功: 対戦相手プレイヤーの石の色 (期待値: 白, 実際値: 白)
成功: CPUの思考エンジンが初期化されている。
成功: CPU実行サービスが初期化されている。
成功: 現在の手番が黒である。 (期待値: 黒, 実際値: 黒)

テスト実行中: testStartGame_CPU_HumanWhite_CPUMakesFirstMove...
Board is initialized.
Board is initialized.
CPU: turn = Black, level = Easy, depth = 4, threshold = 8
成功: ゲームアクティブチェック
成功: 人間プレイヤーの色 (期待値: 白, 実際値: 白)
成功: CPUプレイヤーの色 (期待値: 黒, 実際値: 黒)
成功: 初期手番 (CPU 黒) (期待値: 黒, 実際値: 黒)
testStartGame_CPU_HumanWhite: CPUの最初の着手を待機中 (2秒)...
CPU: Selected move: [2, 3], Score: 0
成功: CPUの着手後の手番 (人間 白) (期待値: 白, 実際値: 白)
成功: CPUの着手後、盤面の石の数が4より大きい

テスト実行中: testHandleCpuPlayerMove_ValidMove...
Board is initialized.
Board is initialized.
CPU: turn = White, level = Easy, depth = 4, threshold = 8
成功: ゲームアクティブ
成功: 現在の手番は黒 (期待値: 黒, 実際値: 黒)
成功: (2,3) は初期状態で黒の有効な手である。
成功: マス (2,3) が黒である。 (期待値: 1, 実際値: 1)
成功: マス (3,3) が黒に反転している。 (期待値: 1, 実際値: 1)
成功: 手番が白 (CPU) に切り替わる。 (期待値: 白, 実際値: 白)

テスト実行中: testShutdownLogic...
Board is initialized.
Board is initialized.
CPU: turn = White, level = Easy, depth = 4, threshold = 8
成功: シャットダウン前のCPU実行サービス
成功: シャットダウン後、ゲームが非アクティブである。
成功: CPU実行サービスがシャットダウンされている。

------------------------------結果------------------------------
実行テスト総数: 44
成功: 44
失敗: 0
----------------------------------------------------------------
```

2.4 View クラス

ドライバを作成しテストを行った。UI の動作を正しく実装されるように画像を利用し結果

を表示する。

2.5 CPU クラス

ドライバ(CPUDriver.java)を用いて CPU クラスの単体テストを行った.

```
import java.util.Scanner;

public class CPUDriver {

    private static final int N_LINE = 8;

    public static void main(String[] args) {

        //テスト用盤面(初期配置)

        Integer[][] testBoard = new Integer[N_LINE][N_LINE];

        Othello.initBoard(testBoard);



        System.out.println("¥nCPU クラスのテスト開始");



        // 1. 初期化テスト

        System.out.println("¥n1. CPU クラスの初期化テスト");

        Scanner scanner = new Scanner(System.in);

        System.out.println("手番を入力してください (1: Black, 2: White)");

        int turnNum = scanner.nextInt();



        System.out.println("CPU の強さを入力してください (0: 弱い, 1: 普通, 2: 強い)");

        int levelNum = scanner.nextInt();
```

```
        System.out.println("以下の内容でCPUが作成されました: ");

        CPU testCPU = null;

        if(turnNum == 1 && levelNum == 0) {

            testCPU = new CPU("Black", "弱い");

        } else if(turnNum == 1 && levelNum == 1) {

            testCPU = new CPU("Black", "普通");

        } else if(turnNum == 1 && levelNum == 2) {

            testCPU = new CPU("Black", "強い");

        } else if(turnNum == 2 && levelNum == 0) {

            testCPU = new CPU("White", "弱い");

        } else if(turnNum == 2 && levelNum == 1) {

            testCPU = new CPU("White", "普通");

        } else if(turnNum == 2 && levelNum == 2) {

            testCPU = new CPU("White", "強い");

        } else {

            testCPU = new CPU("Black", "普通");

        }

        // インスタンス作成時にログ出力があるので、それで確認することとする。

        scanner.close();



        // 2. 操作決定のテスト
```

```
        System.out.println("¥n2. CPU クラスの操作決定テスト");

        System.out.println("操作前の盤面(初期配置)");

        printBoard(testBoard);



        Integer[][] cpuBoard = new Integer[N_LINE][N_LINE];

        for (int i = 0; i < N_LINE; i++) {

            cpuBoard[i] = testBoard[i].clone();

        }

        int[] cpuMove = testCPU.getCPUOperation(testBoard);



        if (cpuMove == null || cpuMove[0] == -1) {

            System.out.println("CPU はパスしました。");

            System.out.println("¥n 操作後の盤面");

            printBoard(testBoard);

        } else {

            System.out.println("CPU の選択: (" + cpuMove[0] + ", " + cpuMove[1] + ")");

            Othello.makeMove(testBoard, cpuMove[0], cpuMove[1], (turnNum == 1) ? "Black" :
"White");

            System.out.println("¥n 操作後の盤面");

            printBoard(testBoard);

        }
```

```
        System.out.println("¥nCPU クラスのテストを終了します。");
    }

    private static void printBoard(Integer[][] board) {
        System.out.print("  ");
        for (int i = 0; i < N_LINE; i++) {
            System.out.print(i + " ");
        }
        System.out.println("¥n-----------------");
        for (int i = 0; i < N_LINE; i++) {
            System.out.print(i + "|");
            for (int j = 0; j < N_LINE; j++) {
                char c = '.';
                if (board[i][j] == 1)
                    c = 'B'; // 黒
                if (board[i][j] == 2)
                    c = 'W'; // 白
                System.out.print(c + " ");
            }
            System.out.println();
```

```
        }

        System.out.println("------------------¥n");

    }

}
```

試験結果:

```
Board is initialized.

CPUクラスのテスト開始

1. CPUクラスの初期化テスト
手番を入力してください (1: Black, 2: White)
1
CPUの強さを入力してください (0: 弱い, 1: 普通, 2: 強い)
1
以下の内容でCPUが作成されました:
CPU: turn = Black, level = 普通, depth = 4, threshold = 8

2. CPUクラスの操作決定テスト
操作前の盤面(初期配置)
  0 1 2 3 4 5 6 7
------------------
0|. . . . . . . .
1|. . . . . . . .
2|. . . . . . . .
3|. . . W B . . .
4|. . . B W . . .
5|. . . . . . . .
6|. . . . . . . .
7|. . . . . . . .
------------------

CPU: Selected move: [2, 3], Score: 0
CPUの選択: (2, 3)

操作後の盤面
  0 1 2 3 4 5 6 7
------------------
0|. . . . . . . .
1|. . . . . . . .
2|. . . B . . . .
3|. . . B B . . .
4|. . . B W . . .
5|. . . . . . . .
6|. . . . . . . .
7|. . . . . . . .
------------------


CPUクラスのテストを終了します。
```

2.6 Server クラス

以下のドライバ（ServerDriver.java）を用いて Server クラスの単体テストを行った。なお、スタブは使用していない。

```java
import java.io.*;
import java.net.*;


public class ServerDriver {
    public static void main(String[] args) throws Exception {
        int port = 3000;


        // サーバーを別スレッドで起動
        Thread serverThread = new Thread(() -> {
            Server server = new Server(port);
            server.acceptClient();
        });
        serverThread.start();


        // 少し待機してサーバ起動を待つ
        Thread.sleep(1000);


        // クライアント1接続
        Socket client1 = new Socket("localhost", port);
        PrintWriter out1 = new PrintWriter(client1.getOutputStream(), true);
        BufferedReader in1 = new BufferedReader(new InputStreamReader(client1.getInputStream()));
        out1.println("Rikuo"); // 名前送信


        // クライアント2接続
        Socket client2 = new Socket("localhost", port);
        PrintWriter out2 = new PrintWriter(client2.getOutputStream(), true);
        BufferedReader in2 = new BufferedReader(new InputStreamReader(client2.getInputStream()));
        out2.println("Kazuki");
```

```
        // プレイヤー名の確認
        System.out.println("[Client1] Received: " + in1.readLine()); // OPPONENT:Kazuki
        System.out.println("[Client1] Received: " + in1.readLine()); // YOUR COLOR:黒

        System.out.println("[Client2] Received: " + in2.readLine()); // OPPONENT:Rikuo
        System.out.println("[Client2] Received: " + in2.readLine()); // YOUR COLOR:白

        // メッセージ送信テスト
        out1.println("Hello from Rikuo");
        System.out.println("[Client2] Message received: " + in2.readLine());

        client1.close();
        client2.close();
        System.exit(0); // サーバ終了
    }
}
```

試験結果：

```
r'
サーバが起動しました。
両方のプレイヤーが未接続です。2人分の接続を待ちます...
プレイヤ 0 が接続しました。
プレイヤ 0 の名前: Rikuo
プレイヤ 1 が接続しました。
プレイヤ 1 の名前: Kazuki
[Client1] Received: OPPONENT:Kazuki
[Client1] Received: YOUR COLOR:黒
[Client2] Received: OPPONENT:Rikuo
[Client2] Received: YOUR COLOR:白
[Client2] Message received: Hello from Rikuo
```

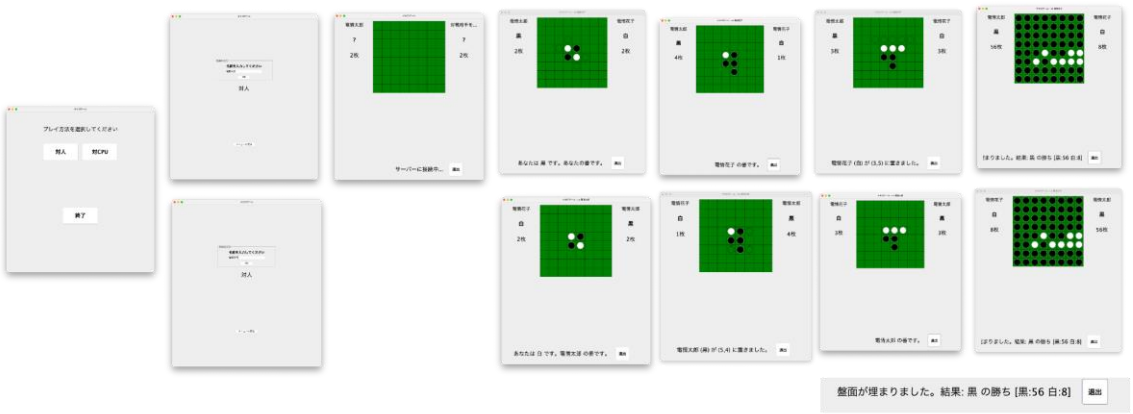
3　結合テスト

以下の手順に従い，結合テストを行った.

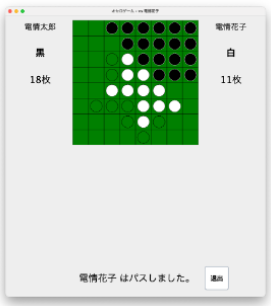
1. Server プログラムを起動
2. Client プログラム１を起動し、「対人」を選択、プレイヤ名「電情太郎」を入力
3. Client プログラム２を起動し、「対人」を選択、プレイヤ名「電情花子」を入力
4. ゲームを進行
5. ゲーム終了後、Client プログラム１を終了し、サーバから切断
6. ゲーム終了後、Client プログラム２を終了し、サーバから切断

試験結果を以下に示す.

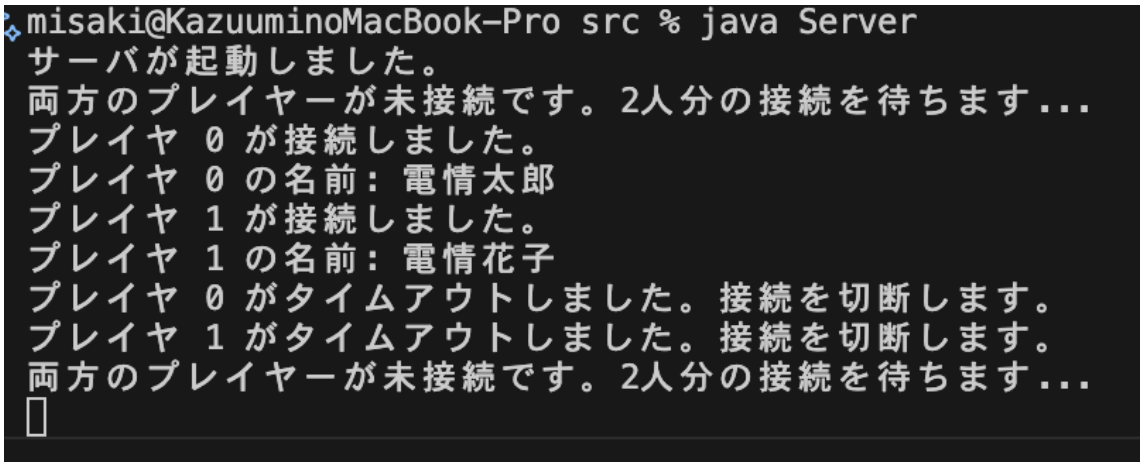
クライアント側：



パス盤面



サーバ側：


misaki@KazuuminoMacBook-Pro src % java Server
サーバが起動しました。
両方のプレイヤーが未接続です。2人分の接続を待ちます...
プレイヤ 0 が接続しました。
プレイヤ 0 の名前: 電情太郎
プレイヤ 1 が接続しました。
プレイヤ 1 の名前: 電情花子
プレイヤ 0 がタイムアウトしました。接続を切断します。
プレイヤ 1 がタイムアウトしました。接続を切断します。
両方のプレイヤーが未接続です。2人分の接続を待ちます...