# HEX BARON – Programming Tasks (MS)

# Task 1 (4 marks)

**Coding:**

* 1 mark for adding the call to the new subroutine (strip) in playGame
* 1 mark for stripping leading spaces
* 1 mark for stripping trailing spaces

For example:

Additional subroutine strip:

String strip(String s) {

int index;

for (index = 0; index < s.length(); index++) {

if (s.charAt(index) != ' ') {

break;

}

}

s = s.substring(index);

for (index = s.length() - 1; index >= 0; index--) {

if (s.charAt(index) != ' ') {

break;

}

}

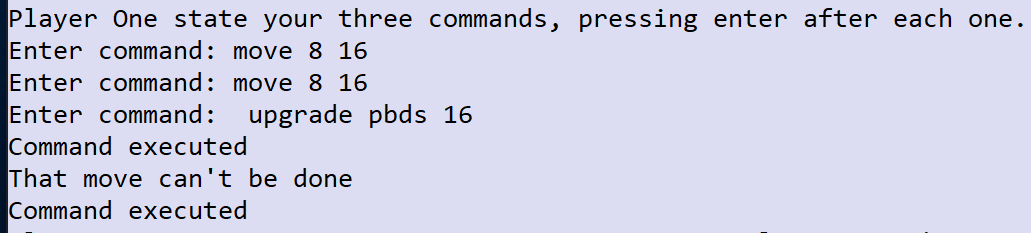
return s.substring(0, index + 1);

}

Changes to playGame:

commands.add(strip(Console.readLine().toLowerCase()));

1. Solutions which manually strip off as many spaces as there are, but not solutions that only check for a single space and then remove it.
2. Solutions which use any of the many libraries available to solve the problem
3. Calls to library subroutines for strip instead of students writing their own subroutine.



**Testing:**

1 mark for showing that the move command is valid the first time and that the upgrade can happen successfully, despite the leading space. For example:

# Task 2 (3 marks)

**Coding:**

* 1 mark for getting the players’ names to be entered by using a **suitable prompt**.
* 1 mark for passing through the names entered correctly to the setUpPlayer method for the relevant player objects.

For example:

String p1name;

String p2name;

HexGrid grid = new HexGrid(gridSize);

Console.write("Player One, please enter your name: ");

p1name = Console.readLine();

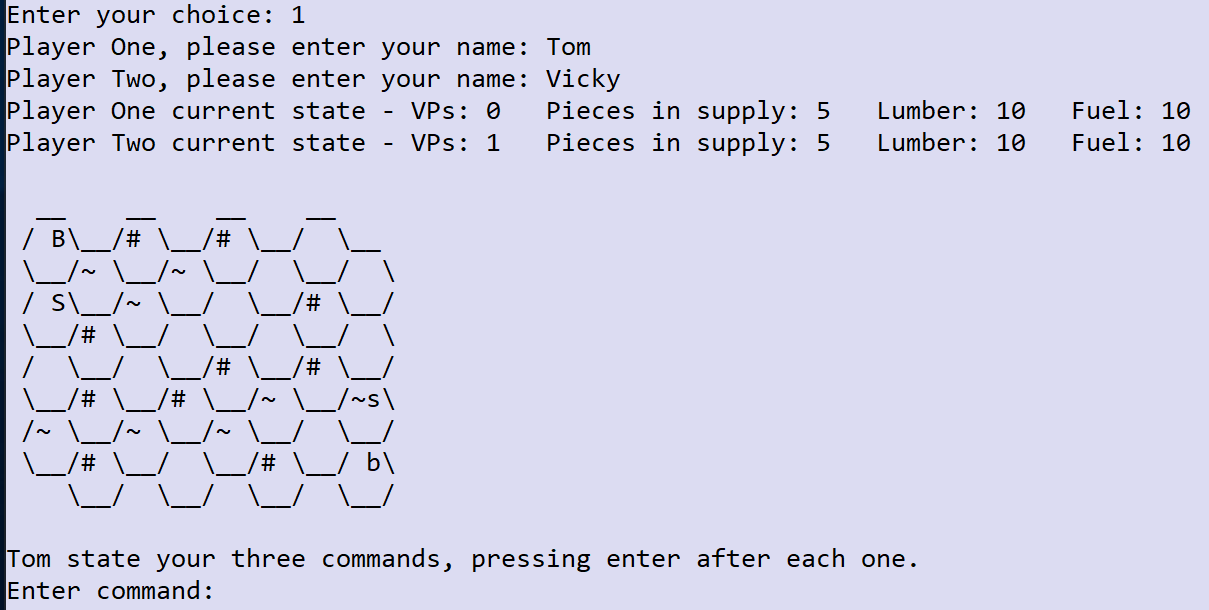
Console.write("Player Two, please enter your name: ");

p2name = Console.readLine();

player1.setUpPlayer(p1name, 0, 10, 10, 5);

player2.setUpPlayer(p2name, 1, 10, 10, 5);

1. Solutions which store the inputs in two variables and then pass those through to the setUpPlayer method when called.

**Testing:**

1 mark for showing a suitable prompt for the names for each player AND using those names when printing out the states and asking for the entry of the commands.

For example:

# Task 3 (4 marks)

**Coding:**

* 1 mark for creating the addPiecesInSupply method correctly to the Player class.
* 1 mark for passing through player1 and player2 to both calls to destroyPiecesAndCountVPs and adding the two parameters to the method.
* 1 mark for calling the addPiecesInSupply method for the correct player so that it adds one piece to their supply.

For example:

addPiecesInSupply method added to the Player class:

public void addPiecesInSupply(int n) {

piecesInSupply += 1;

}

Both calls to destroyPiecesAndCountVPs:

returnObjects = grid.destroyPiecesAndCountVPs(player1VPsGained, player2VPsGained, player1, player2);

Adding parameters to the destroyPiecesAndCountVPs method:

public Object[] destroyPiecesAndCountVPs(int player1VPs, int player2VPs, Player player1, Player player2) {

Modifications to the destroyPiecesAndCountVPs method:

if (thePiece.getBelongsToplayer1()) {

player1.addPiecesInSupply(1);

player2VPs += thePiece.getVPs();

} else {

player2.addPiecesInSupply(1);

player1VPs += thePiece.getVPs();

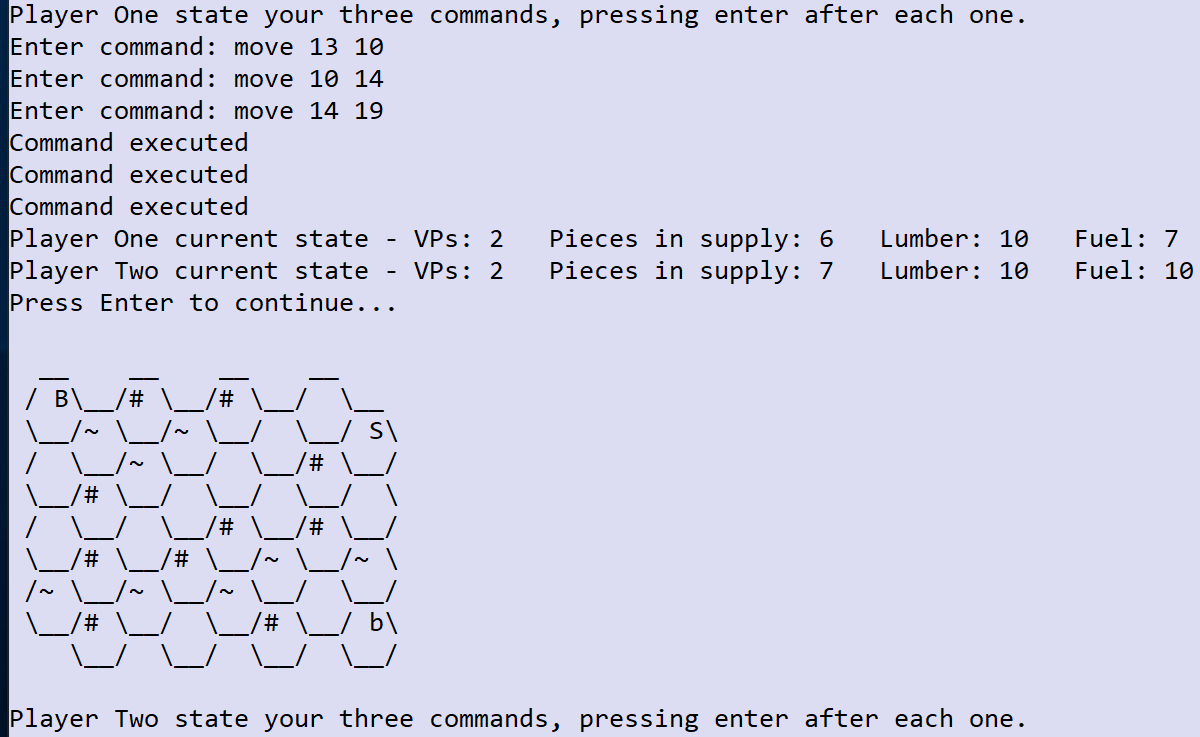
}

1. Versions of the addPiecesInSupply which add a single piece and can’t take a parameter; also accept a slightly modified name such as addPieceInSupply.

**Testing:**

1 mark for showing entry of all three commands correctly and a printout of the pieces in supply and board as shown.

For example:



# Task 4 (4 marks)

**Coding:**

* 1 mark for counting the number of pieces on the board for the current player correctly.
* 1 mark for printing out the message ‘Spawn attempted to exceed max pieces’ when the number of pieces for the player is 6 or more (award even if the first mark calculates the number of pieces incorrectly).
* 1 mark for returning -1 when the number of pieces for the player is 6 or more AND putting this code before the Piece newPiece = new Piece(player1Turn); line of code (anywhere before it in the method).

For example:

int playerPieces = 0;

if (piecesInSupply < 1 || lumberAvailable < 3 || !checkTileIndexIsValid(tileToUse)) {

return -1;

}

Piece thePiece = tiles.get(tileToUse).getPieceInTile();

if (thePiece != null) {

return -1;

}

boolean ownBaronIsNeighbour = false;

List<Tile> listOfNeighbours = new ArrayList<>(tiles.get(tileToUse).getNeighbours());

for (Tile n : listOfNeighbours) {

thePiece = n.getPieceInTile();

if (thePiece != null) {

if (player1Turn && thePiece.getPieceType().equals("B") || !player1Turn && thePiece.getPieceType().equals("b")) {

ownBaronIsNeighbour = true;

break;

}

}

}

if (!ownBaronIsNeighbour) {

return -1;

}

for (Piece p: pieces) {

if (p.getBelongsToplayer1() == player1Turn) {

playerPieces += 1;

}

}

if (playerPieces >= 6) {

Console.writeLine("Spawn attempted to exceed max pieces.");

return -1;

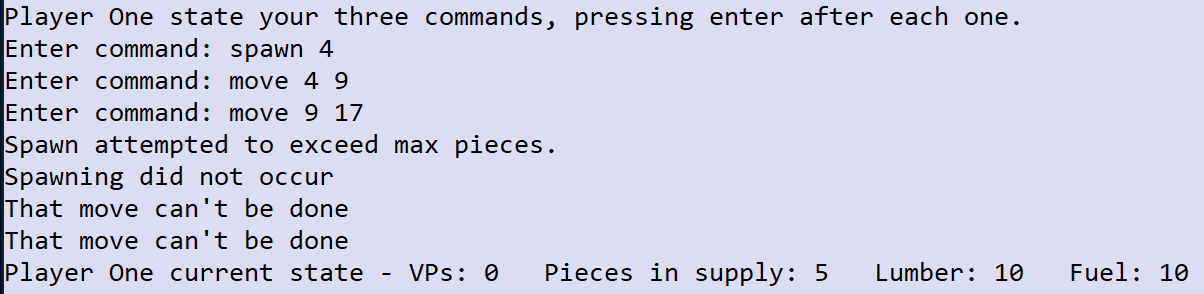
}

Piece newPiece = new Piece(player1Turn);

**Testing:**

1 mark for showing the commands entered correctly and the error message as above along with the existing error message saying that spawning did not occur.

For example:



# Task 5 (9 marks)

**Coding:**

* 1 mark for calling the new method when the ‘hexes’ command is entered.
* 1 mark for making sure that another command can be entered and that the ‘hexes’ command did not count as one of the three commands (no matter how many times they enter it).
* 1 mark for having a variable to track the tile index and incrementing it throughout the process.
* 1 mark for creating the string correctly for the top and bottom lines (only if the format is correct).
* 1 mark for creating the string with the correct index numbers for the even lines (even if the format is off).
* 1 mark for creating the string with the correct index numbers for the odd lines (even if the format is off).

For example:

Modifications to the playGame subroutine:

for (int count = 1; count <= 3; count++) {

String command;

Console.write("Enter command: ");

command = Console.readLine().toLowerCase();

while (command == "hexes") {

grid.getGridAsIndices();

Console.write("Enter command: ");

command = Console.readLine().toLowerCase();

}

commands.add(command);

}

Code for new getGridAsIndices method:

public String getGridAsIndices() {

int listPositionOfTile = 0;

int index = 0;

Object[] returnObjects = createEvenIndicesLine(true, listPositionOfTile, index);

String gridAsString = createTopLine() + returnObjects[0].toString();

listPositionOfTile = (int)returnObjects[1];

listPositionOfTile += 1;

index = (int)returnObjects[2];

returnObjects = createOddIndicesLine(listPositionOfTile, index);

gridAsString += returnObjects[0].toString();

listPositionOfTile = (int)returnObjects[1];

index = (int)returnObjects[2];

for (int count = 1; count < size - 1; count+=2) {

listPositionOfTile += 1;

returnObjects = createEvenIndicesLine(false, listPositionOfTile, index);

gridAsString += returnObjects[0].toString();

listPositionOfTile = (int)returnObjects[1];

listPositionOfTile += 1;

index = (int)returnObjects[2];

returnObjects = createOddIndicesLine(listPositionOfTile, index);

gridAsString += returnObjects[0].toString();

listPositionOfTile = (int)returnObjects[1];

index = (int)returnObjects[2];

}

return gridAsString + createBottomLine();

}

Code for the new createOddLineIndices method:

private Object[] createOddIndicesLine(int listPositionOfTile, int index) {

String line = "";

for (int count = 1; count <= size / 2; count++) {

if (count > 1 && count < size / 2) {

line += "\\\_\_/";

listPositionOfTile += 1;

line += padIndex(index);

index += 1;

} else if (count == 1) {

line += " \\\_\_/" + padIndex(index);

index += 1;

}

}

line += "\\\_\_/";

listPositionOfTile += 1;

if (listPositionOfTile < tiles.size()) {

line += padIndex(index) + "\\\n";

index += 1;

} else {

line += "\\\n";

}

return new Object[]{line, listPositionOfTile, index};

}

Code for the new createEvenLineIndices method:

private Object[] createEvenIndicesLine(boolean firstEvenLine, int listPositionOfTile, int index) {

String line = " /" + padIndex(index);

index += 1;

for (int count = 1; count < size / 2; count++) {

listPositionOfTile += 1;

line += "\\\_\_/" + padIndex(index);

index += 1;

}

if (firstEvenLine) {

line += "\\\_\_\n";

} else {

line += "\\\_\_/\n";

}

return new Object[]{line, listPositionOfTile, index};

}

Code for new padIndex method:

private String padIndex(int n) {

if (n < 10) {

return " " + Integer.toString(n);

} else {

return Integer.toString(n);

}

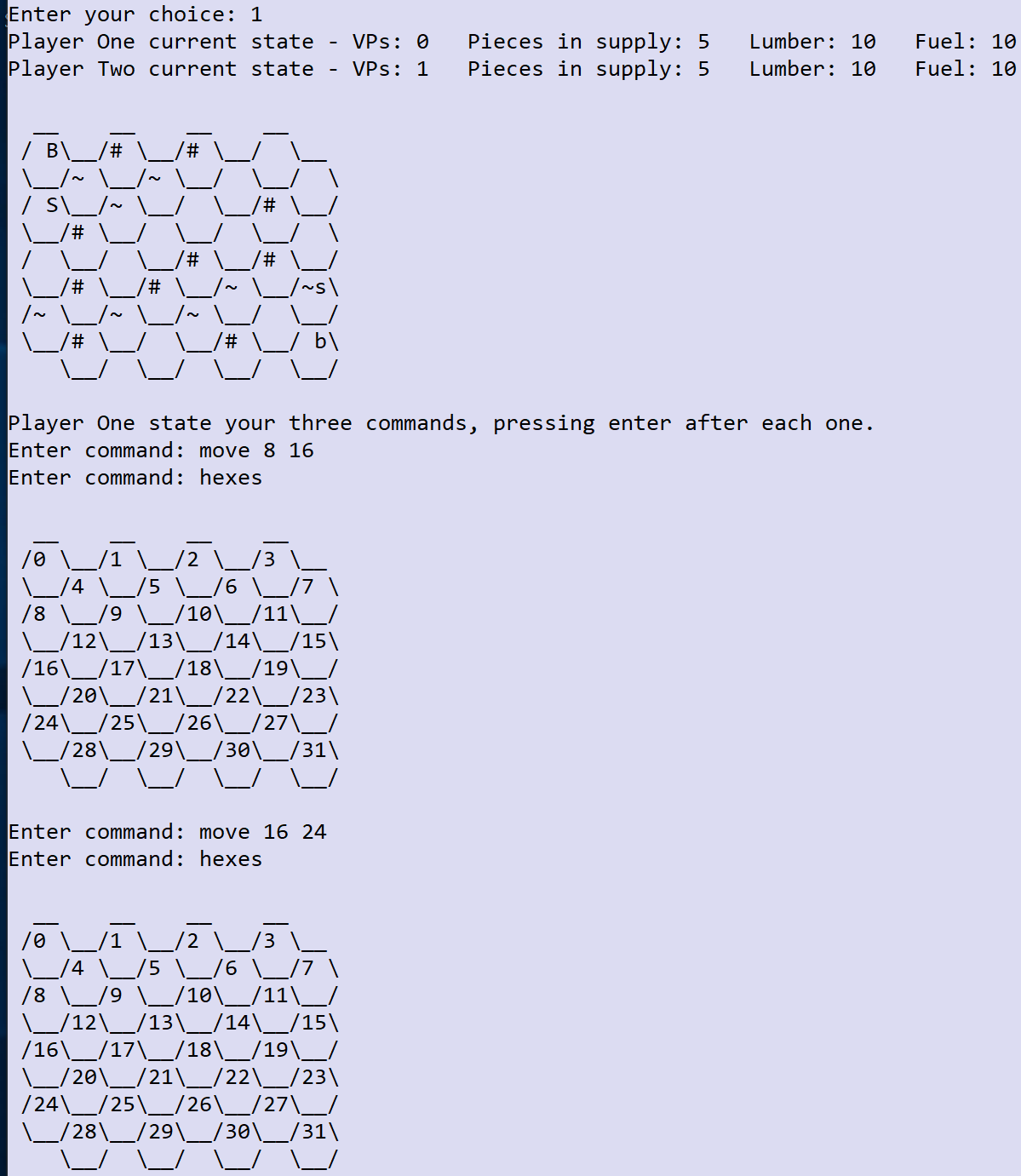
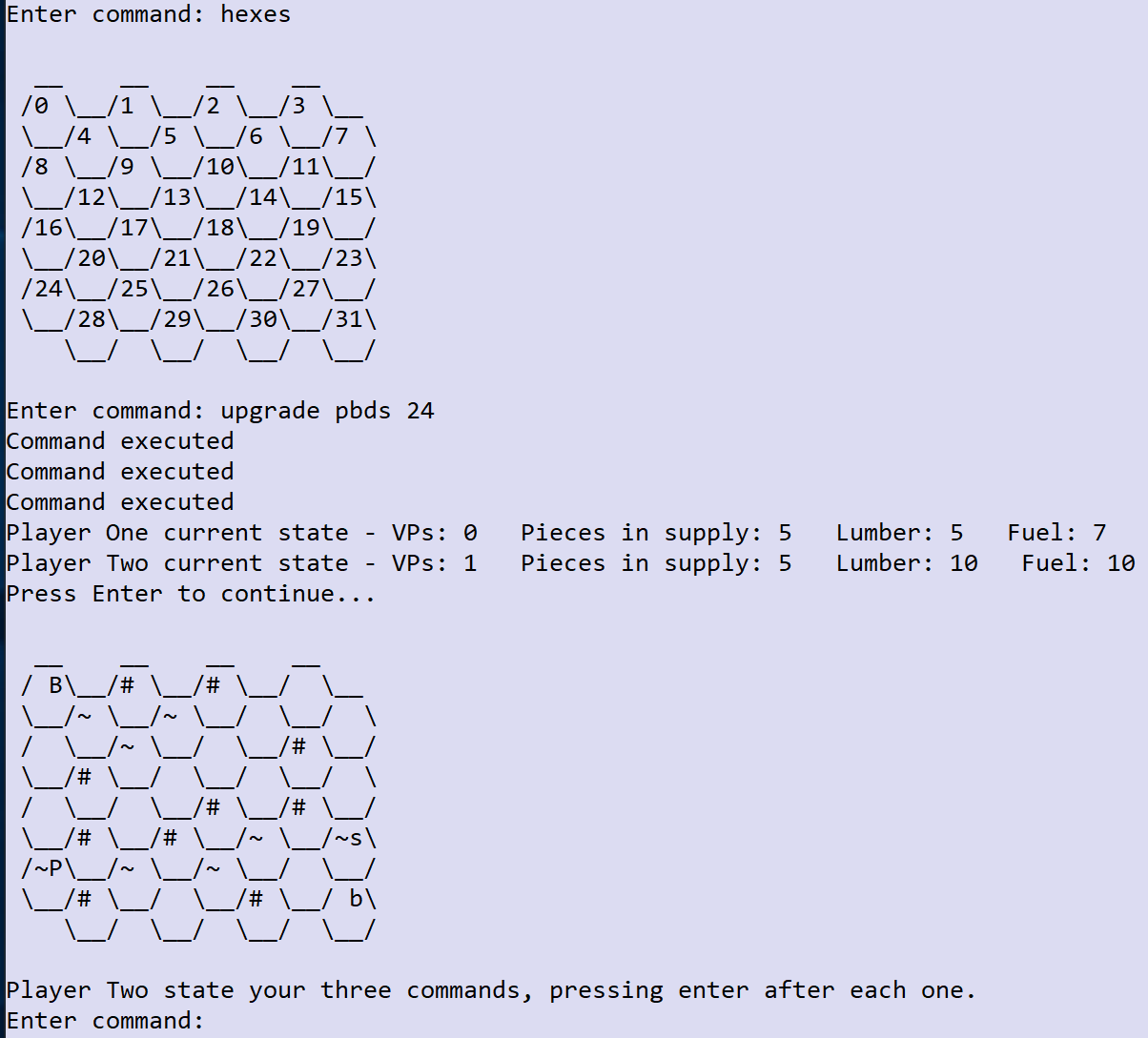
}

1. Alternative solutions that simply print out the grid without modifying or reusing existing code, just award the marks based on the mark scheme above, but allow outputting of the correct line instead of having to store it in a variable.

**Testing:**

* 1 mark for printing out the first line of the grid correctly formatted with the numbers 0‒3 shown for the index numbers.
* 1 mark for printing out the whole of the grid with all the numbers correctly from 0–31 even if the formatting is off.
* 1 mark for printing out the whole of the grid, correctly formatted with the correct numbers using left-aligned numbering for each hex.

For example:



# Task 6 (4 marks)

**Coding:**

* 1 mark for initialising a counter to 0 before the for (String c : commands) loop and then incrementing it inside the loop (even if logic is otherwise incorrect).
* 1 mark for having the correction criteria for a selection structure to check that the player has made three valid move commands.
* 1 mark for giving Player One additional fuel before the third move is made (or discounting the cost of the third move).

For example:

do {

int validMoveCommands = 0;

Console.writeLine(grid.getGridAsString(player1Turn));

if (player1Turn) {

Console.writeLine(player1.getName() + " state your three commands, pressing enter after each one.");

} else {

Console.writeLine(player2.getName() + " state your three commands, pressing enter after each one.");

}

for (int count = 1; count <= 3; count++) {

Console.write("Enter command: ");

commands.add(Console.readLine().toLowerCase());

}

for (String c : commands) {

List<String> items = Arrays.asList(c.split(" "));

validCommand = checkCommandIsValid(items);

if (!validCommand) {

Console.writeLine("Invalid command");

} else {

int fuelChange = 0;

int lumberChange = 0;

int supplyChange = 0;

String summaryOfResult;

Object[] returnObjects;

if (items.get(0).equals("move")) {

validMoveCommands += 1;

if (validMoveCommands == 3) {

if (player1Turn) {

player1.updateFuel(1);

} else {

player2.updateFuel(1);

}

}

}

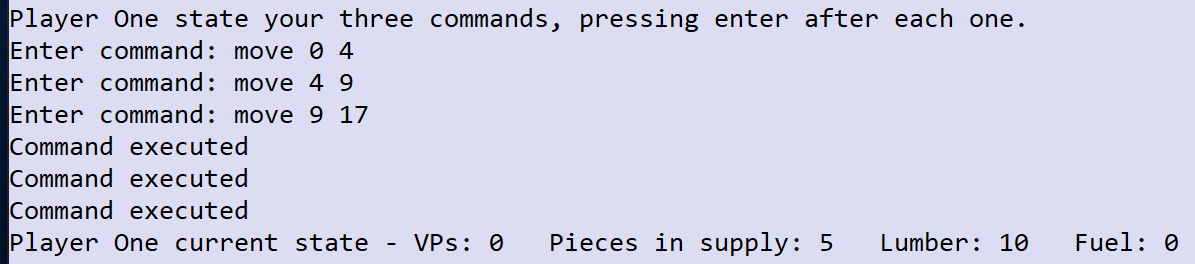
if (player1Turn) {

1. Solutions which iterate over the commands list and count the number of move commands as long as they take account of whether or not they are valid.

**Testing:**

1 mark for showing the commands entered correctly and that the fuel for Player One is now 0 in the printout of their state.

For example:



# Task 7 (7 marks)

**Coding:**

* 1 mark for correctly removing the code from executeCommandInTile.
* 1 mark for initialising variables to track the valid dig and saw commands.
* 1 mark for ensuring that the three commands happened at the same location (before awarding the extra resource and generating the field).
* 1 mark for correctly awarding the extra 2 resources for 3 consecutive dig or saw commands (even if they didn’t check the location).
* 1 mark for calling makeField under the correct circumstances (3 valid saws or digs in the same location in the same turn).
* 1 mark for correctly creating the makeField method.

For example:

Modifications to executeCommandInTile:

} else if (items.get(0).equals("dig")) {

fuel += (int)method.invoke(thePiece, parameters);

}

Code for the modified playGame subroutine:

int validDigCommands = 0;

int validSawCommands = 0;

String digSawLocation = "";

Console.writeLine(grid.getGridAsString(player1Turn));

if (player1Turn) {

Console.writeLine(player1.getName() + " state your three commands, pressing enter after each one.");

} else {

Console.writeLine(player2.getName() + " state your three commands, pressing enter after each one.");

}

for (int count = 1; count <= 3; count++) {

Console.write("Enter command: ");

commands.add(Console.readLine().toLowerCase());

}

for (String c : commands) {

List<String> items = Arrays.asList(c.split(" "));

validCommand = checkCommandIsValid(items);

if (!validCommand) {

Console.writeLine("Invalid command");

} else {

int fuelChange = 0;

int lumberChange = 0;

int supplyChange = 0;

String summaryOfResult;

Object[] returnObjects;

if (digSawLocation.equals("")) {

digSawLocation = items.get(1);

}

if (items.get(0).equals("saw") && items.get(1).equals(digSawLocation)) {

validSawCommands += 1;

} else if (items.get(0).equals("dig") && items.get(1).equals(digSawLocation)) {

validDigCommands += 1;

}

if (player1Turn) {

returnObjects = grid.executeCommand(items, fuelChange, lumberChange, supplyChange, player1.getFuel(),

player1.getLumber(),player1.getPiecesInSupply());

summaryOfResult = returnObjects[0].toString();

fuelChange = (int)returnObjects[1];

lumberChange = (int)returnObjects[2];

supplyChange = (int)returnObjects[3];

player1.updateLumber(lumberChange);

player1.updateFuel(fuelChange);

if (supplyChange == 1) {

player1.removeTileFromSupply();

}

} else {

returnObjects = grid.executeCommand(items, fuelChange, lumberChange, supplyChange, player2.getFuel(),

player2.getLumber(), player2.getPiecesInSupply());

summaryOfResult = returnObjects[0].toString();

fuelChange = (int)returnObjects[1];

lumberChange = (int)returnObjects[2];

supplyChange = (int)returnObjects[3];

player2.updateLumber(lumberChange);

player2.updateFuel(fuelChange);

if (supplyChange == 1) {

player2.removeTileFromSupply();

}

}

if (validSawCommands == 3) {

if (player1Turn) {

player1.updateLumber(2);

} else {

player2.updateLumber(2);

}

grid.makeField(Integer.parseInt(digSawLocation));

} else if (validDigCommands == 3) {

if (player1Turn) {

player1.updateFuel(2);

} else {

player2.updateFuel(2);

}

grid.makeField(Integer.parseInt(digSawLocation));

}

Console.writeLine(summaryOfResult);

Code for new makeField method:

public void makeField(int tileIndex) {

tiles.get(tileIndex).setTerrain(" ");

}

Code for the modified dig method:

public int dig(String terrain) {

if (!terrain.equals("~")) {

return 0;

} else {

return 1;

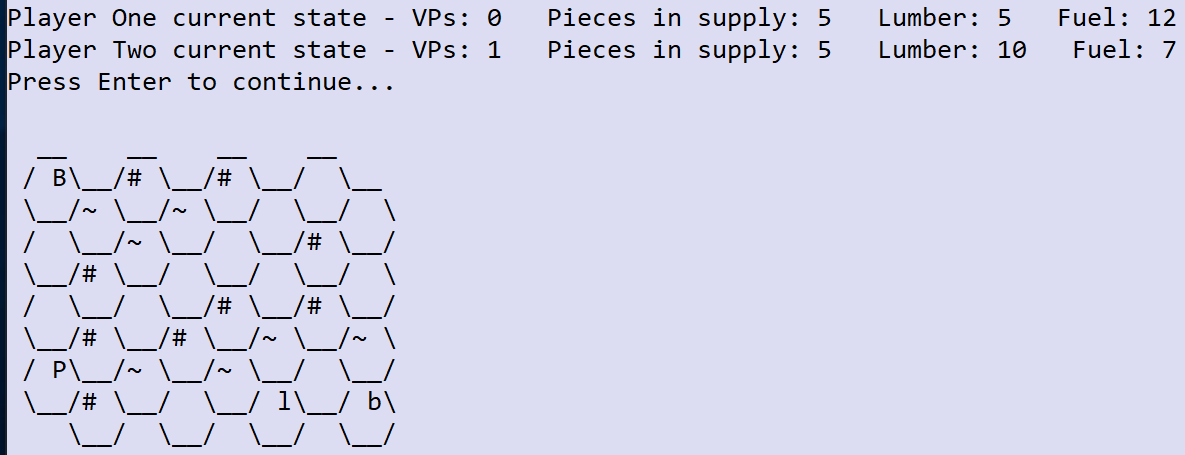
}

}

**Testing:**

1 mark for showing the final board after all 12 commands have been executed including the state of both players.

For example:



# Task 8 (7 marks)

**Coding:**

* 1 mark for correctly modifying the checkCommandIsValid subroutine to recognise downgrade and call the new subroutine.
* 1 mark for creating the new subroutine checkDowngradeCommandFormat and having it return true when there are two items and the second one is an integer, and otherwise returning false.
* 1 mark for modifying the executeCommand method to add the new downgrade command and call the new executeDowngradeCommand method.
* 1 mark for correctly processing the return value from the new method and ensuring that the player will receive a refund of 1 lumber.
* 1 mark for creating the executeDowngradeCommand method and having it check that the piece in the specified tile is either a PBDS or a LESS.
* 1 mark for correctly converting the piece to a Serf and updating the pieces list.

For example:

Code for modified checkCommandIsValid subroutine:

case "downgrade":

return checkDowngradeCommandFormat(items);

Code for new checkDowngradeCommandFormat subroutine:

boolean checkDowngradeCommandFormat(List<String> items) {

int result;

if (items.size() == 2) {

try {

result = Integer.parseInt(items.get(1));

} catch (Exception e) {

return false;

}

return true;

}

return false;

}

Code for modified executeCommand method:

case "downgrade":

lumberCost = executeDowngradeCommand(items);

if (lumberCost == 0) {

return new Object[] {"Downgrade not possible", fuelChange, lumberChange, supplyChange};

}

lumberChange = -lumberCost;

break;

Code for new executeDowngradeCommand method:

private int executeDowngradeCommand(List<String> items) {

int tileToUse = Integer.parseInt(items.get(1));

if (!checkPieceAndTileAreValid(tileToUse)) {

return 0;

} else {

Piece thePiece = tiles.get(tileToUse).getPieceInTile();

if (!thePiece.getPieceType().toUpperCase().equals("P") && !thePiece.getPieceType().toUpperCase().equals("L")) {

return 0;

}

thePiece.destroyPiece();

thePiece = new Piece(player1Turn);

pieces.add(thePiece);

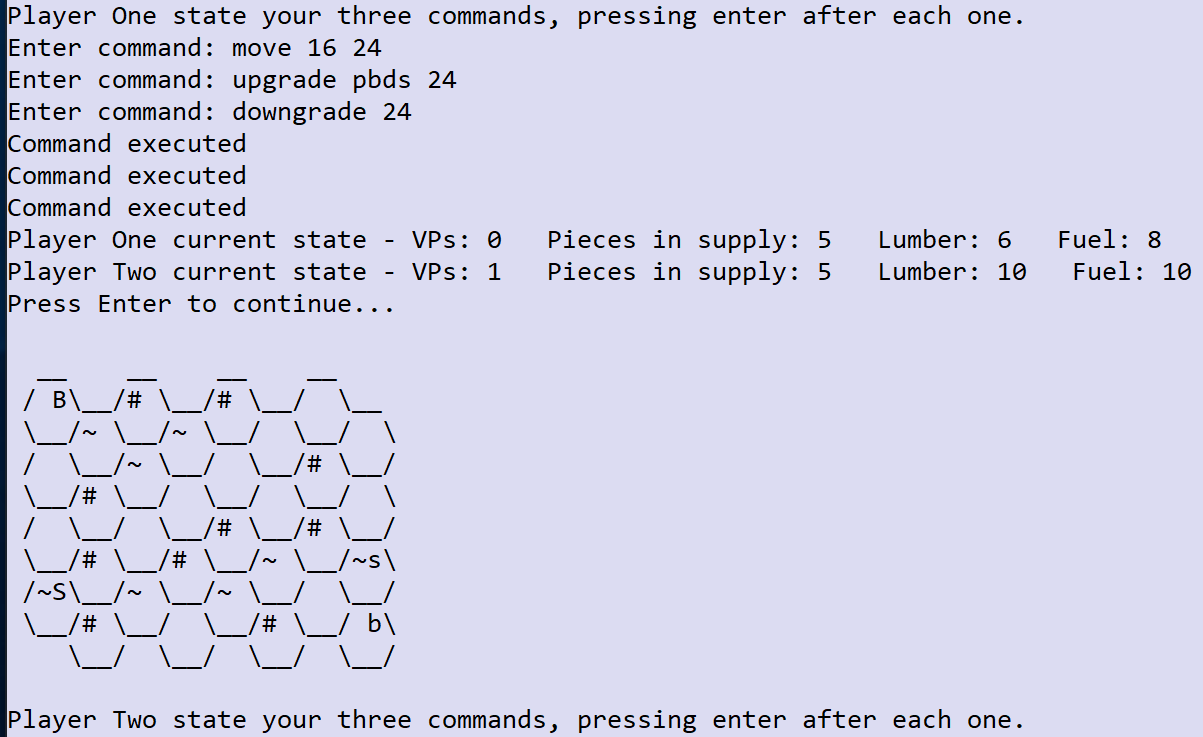
tiles.get(tileToUse).setPiece(thePiece);

return -1;

}

}

1. Solutions which refund the 1 lumber in a different way and hence would have alternative logic/return values from those shown above. Also accept any solution that just adds the downgrade command to the list of standard commands and uses the existing checkStandardCommandFormat – give marks the same as per developing your own method for this.



**Testing:**

1 mark for showing the commands entered correctly and the error message as above along with the system error message saying that spawning did not occur.

For example:

# Task 9 (3 marks)

**Coding:**

* 1 mark for correctly overriding the protected attribute connectionsToDestroy in the BaronPiece class.

For example:

public BaronPiece(boolean player1) {

super(player1);

connectionsToDestroy = 3;

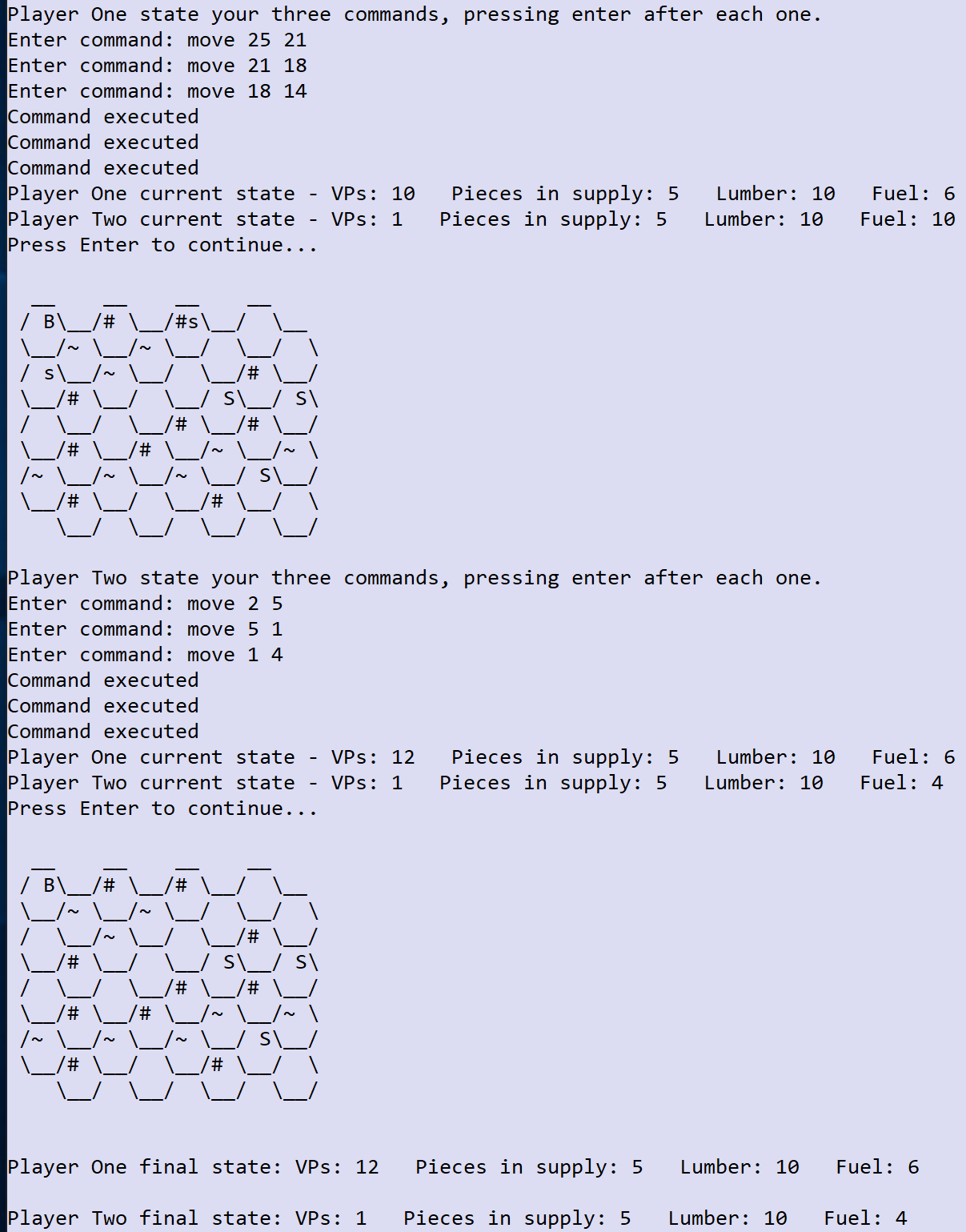
pieceType = "B";

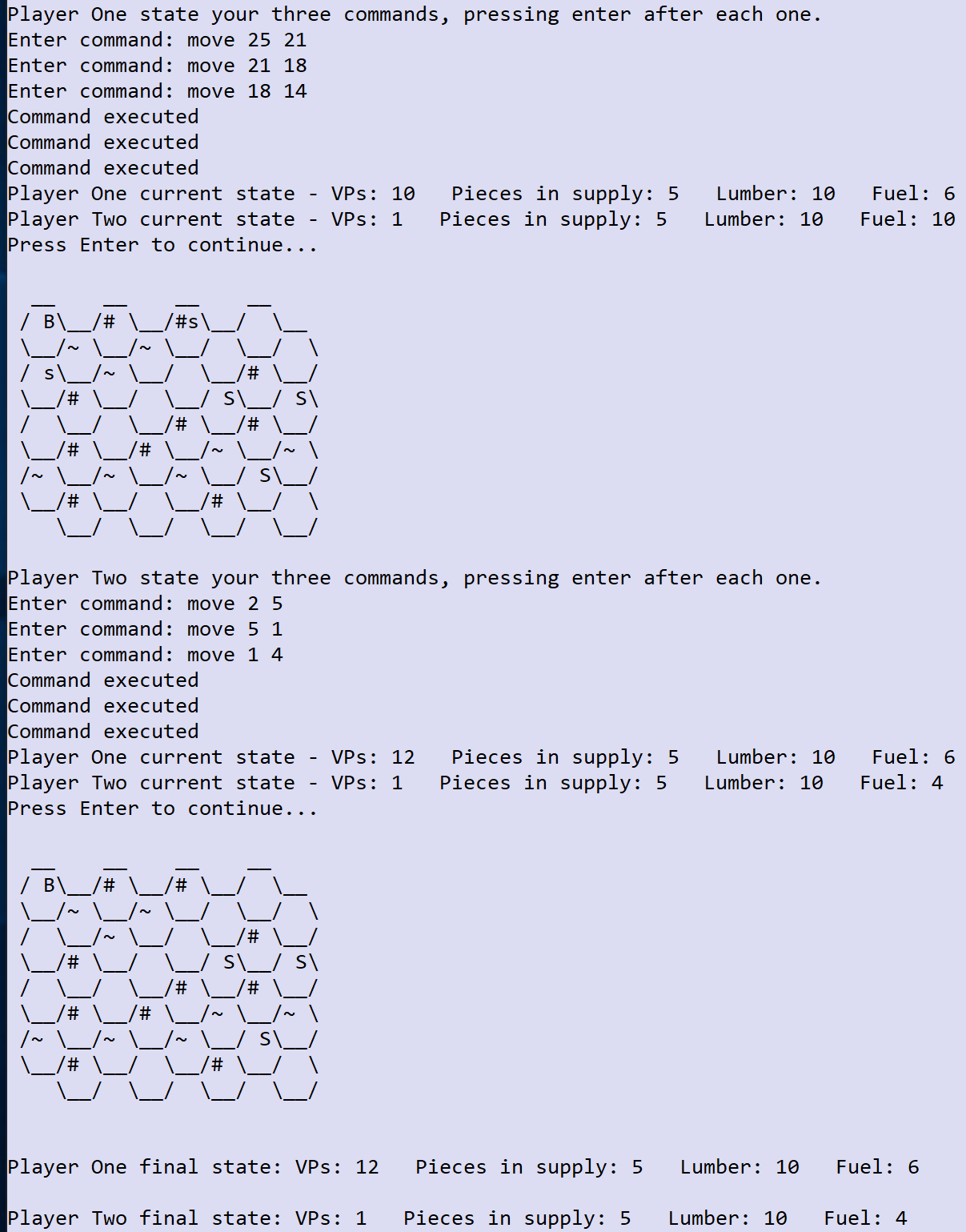
vPValue = 10;

}

**Testing:**

* 1 mark for showing that the Player Two Baron was destroyed.
* 1 mark for showing that the Player Two Serfs were destroyed but that the Player One Baron was not.

For example:



# Task 10 (4 marks)

**Coding:**

* 1 mark for modifying checkCommandIsValid to add ‘salvage’ to the list of commands with a standard format.
* 1 mark for printing out the message ‘Salvaging was not possible’ (or similar) when the command attempts to salvage an empty hex or the Baron or one of the other player’s pieces.
* 1 mark for correctly removing the piece from the board when it is salvaged.

For example:

Code for modified checkCommandIsValid subroutine:

case "dig":

case "salvage":

case "saw":

case "spawn":

return checkStandardCommandFormat(items);

Code for modified executeCommand method:

case "salvage":

lumberCost = executeSalvageCommand(items);

if (lumberCost == 0) {

return new Object[] {"Salvaging was not possible", fuelChange, lumberChange, supplyChange};

}

lumberChange = -lumberCost;

supplyChange = -1;

break;

Code for new executeSalvageCommand method:

private int executeSalvageCommand(List<String> items) {

int tileToUse = Integer.parseInt(items.get(1));

if (!checkPieceAndTileAreValid(tileToUse)) {

return 0;

} else {

Piece thePiece = tiles.get(tileToUse).getPieceInTile();

switch (thePiece.getPieceType()) {

case "L":

case "P":

case "S":

if (player1Turn) {

thePiece.destroyPiece();

return -5;

}

break;

case "l":

case "p":

case "s":

if (!player1Turn) {

thePiece.destroyPiece();

return -5;

}

break;

}

return 0;

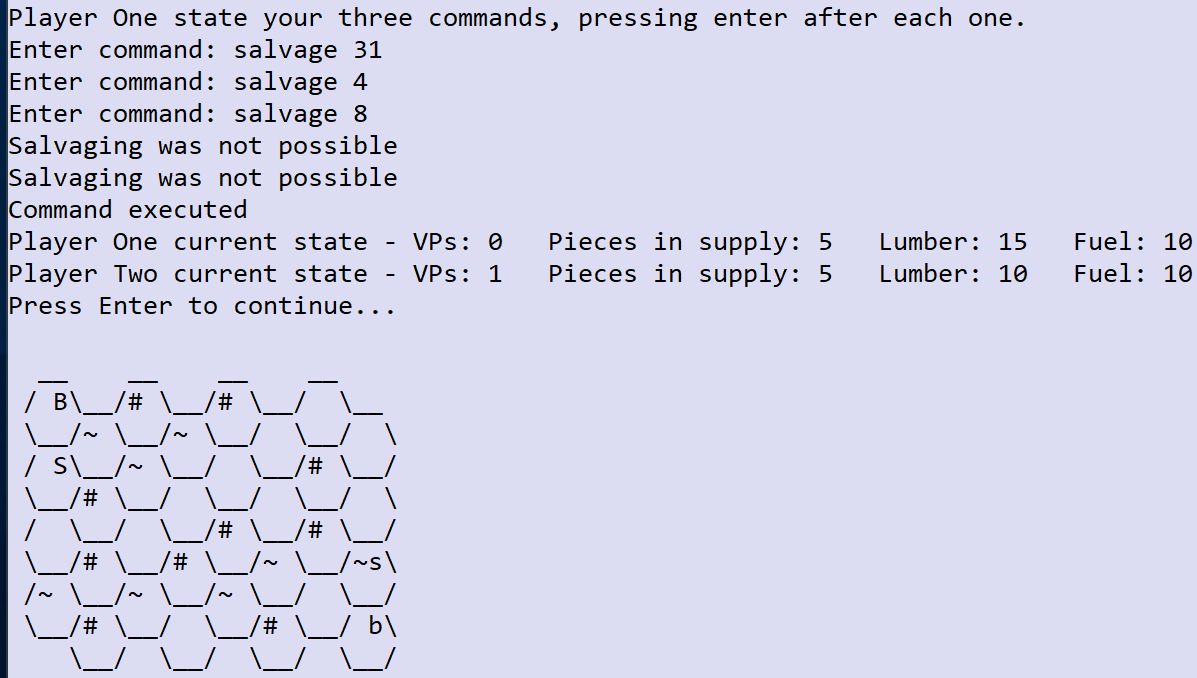
}

}

1. Solutions which are similar to Task 8 and have their own case condition and checking function but only give the same mark as if they solved the problem the short way.

**Testing:**

1 mark for showing the commands entered correctly and the error message as above along with the system error message saying that salvaging was not possible (twice).

 For example:

# Task 11 (10 marks)

**Coding:**

* 1 mark for making the modifications to the Player class so that it now has a private ‘chances’ attribute and three methods to access it: one that will subtract one from the number of chances, one that will return the number of chances and one that will reset the number of chances to 3.
* 1 mark for resetting the chances to 3 at the start of each game for both players.
* 1 mark for allowing a command to be re-entered if it is of the incorrect format and deducting one chance for this (even if it only works in one place).
* 1 mark for allowing a command to be re-entered if it is of the correct format but could not be executed (e.g. move 8 8) and deducting one chance for this (even if it only works in one place).
* 1 mark for making sure that you always have exactly 3 chances.
* 1 mark for printing out an error message every time a command is entered that is invalid or could not be executed correctly.
* 1 mark for ensuring that all command entry prompts and all command error messages have an accurate command number associated with them (from 1 to 3).

For example:

Code for modified Player class:

class Player {

protected int piecesInSupply, fuel, vPs, lumber;

protected String name;

private int chances;

public Player() {}

public void setUpPlayer(String n, int v, int f, int l, int t) {

name = n;

vPs = v;

fuel = f;

lumber = l;

piecesInSupply = t;

resetChances();

}

public int getChances() {

return chances;

}

public void deductChance() {

chances -= 1;

Console.writeLine("You have " + chances + " chances remaining.");

}

public void resetChances() {

chances = 3;

}

Code for modified playGame subroutine:

void playGame(Player player1, Player player2, HexGrid grid) {

boolean gameOver = false;

boolean player1Turn = true;

boolean validCommand;

boolean commandExecuted;

List<String> commands = new ArrayList<>();

Player currentPlayer;

Console.writeLine("Player One current state - " + player1.getStateString());

Console.writeLine("Player Two current state - " + player2.getStateString());

do {

if (player1Turn) {

currentPlayer = player1;

} else {

currentPlayer = player2;

}

Console.writeLine(grid.getGridAsString(player1Turn));

Console.writeLine(currentPlayer.getName() + " state your three commands, pressing enter after each one.");

for (int count = 1; count <= 3; count++) {

Console.write("Enter command: ");

commands.add(Console.readLine().toLowerCase());

}

for (int commandNo = 0; commandNo < commands.size(); commandNo++) {

String command = commands.get(commandNo);

List<String> items = Arrays.asList(command.split(" "));

validCommand = checkCommandIsValid(items);

while (!validCommand && (currentPlayer.getChances() > 0)) {

currentPlayer.deductChance();

Console.writeLine("Command number "+(commandNo+1)+" is an invalid command");

Console.write("Enter new command "+(commandNo+1)+": ");

command = Console.readLine().toLowerCase();

items = Arrays.asList(command.split(" "));

validCommand = checkCommandIsValid(items);

}

if (!validCommand && (currentPlayer.getChances() == 0)) {

Console.writeLine("Command number "+(commandNo+1)+" is an invalid command");

}

commandExecuted = false;

while (!commandExecuted && validCommand) {

int fuelChange = 0;

int lumberChange = 0;

int supplyChange = 0;

String summaryOfResult;

Object[] returnObjects;

returnObjects = grid.executeCommand(items, fuelChange, lumberChange, supplyChange, currentPlayer.getFuel(),

currentPlayer.getLumber(), currentPlayer.getPiecesInSupply());

summaryOfResult = returnObjects[0].toString();

fuelChange = (int)returnObjects[1];

lumberChange = (int)returnObjects[2];

supplyChange = (int)returnObjects[3];

currentPlayer.updateLumber(lumberChange);

currentPlayer.updateFuel(fuelChange);

if (supplyChange == 1) {

currentPlayer.removeTileFromSupply();

}

Console.writeLine("Command "+(commandNo+1)+": "+summaryOfResult);

if (summaryOfResult.equals("Command executed")) {

commandExecuted = true;

} else if (currentPlayer.getChances() > 0) {

currentPlayer.deductChance();

Console.write("Enter new command "+(commandNo+1)+": ");

command = Console.readLine().toLowerCase();

items = Arrays.asList(command.split(" "));

validCommand = checkCommandIsValid(items);

if (!validCommand && (currentPlayer.getChances() == 0)) {

Console.writeLine("Command number "+(commandNo+1)+" is an invalid command");

}

while (!validCommand && (currentPlayer.getChances() > 0)) {

currentPlayer.deductChance();

Console.writeLine("Command number "+(commandNo+1)+" is an invalid command");

Console.write("Enter new command "+(commandNo+1)+": ");

command = Console.readLine().toLowerCase();

items = Arrays.asList(command.split(" "));

validCommand = checkCommandIsValid(items);

if (!validCommand && (currentPlayer.getChances() == 0)) {

Console.writeLine("Command number "+(commandNo+1)+" is an invalid command");

commandExecuted = true;

validCommand = true;

}

}

} else {

break;

}

}

}

commands.clear();

player1Turn = !player1Turn;

int player1VPsGained = 0;

int player2VPsGained= 0;

Object[] returnObjects;

if (gameOver) {

returnObjects = grid.destroyPiecesAndCountVPs(player1VPsGained, player2VPsGained);

} else {

returnObjects = grid.destroyPiecesAndCountVPs(player1VPsGained, player2VPsGained);

gameOver = (boolean)returnObjects[0];

}

player1VPsGained = (int)returnObjects[1];

player2VPsGained = (int)returnObjects[2];

player1.addToVPs(player1VPsGained);

player2.addToVPs(player2VPsGained);

Console.writeLine("Player One current state - " + player1.getStateString());

Console.writeLine("Player Two current state - " + player2.getStateString());

Console.write("Press Enter to continue...");

Console.readLine();

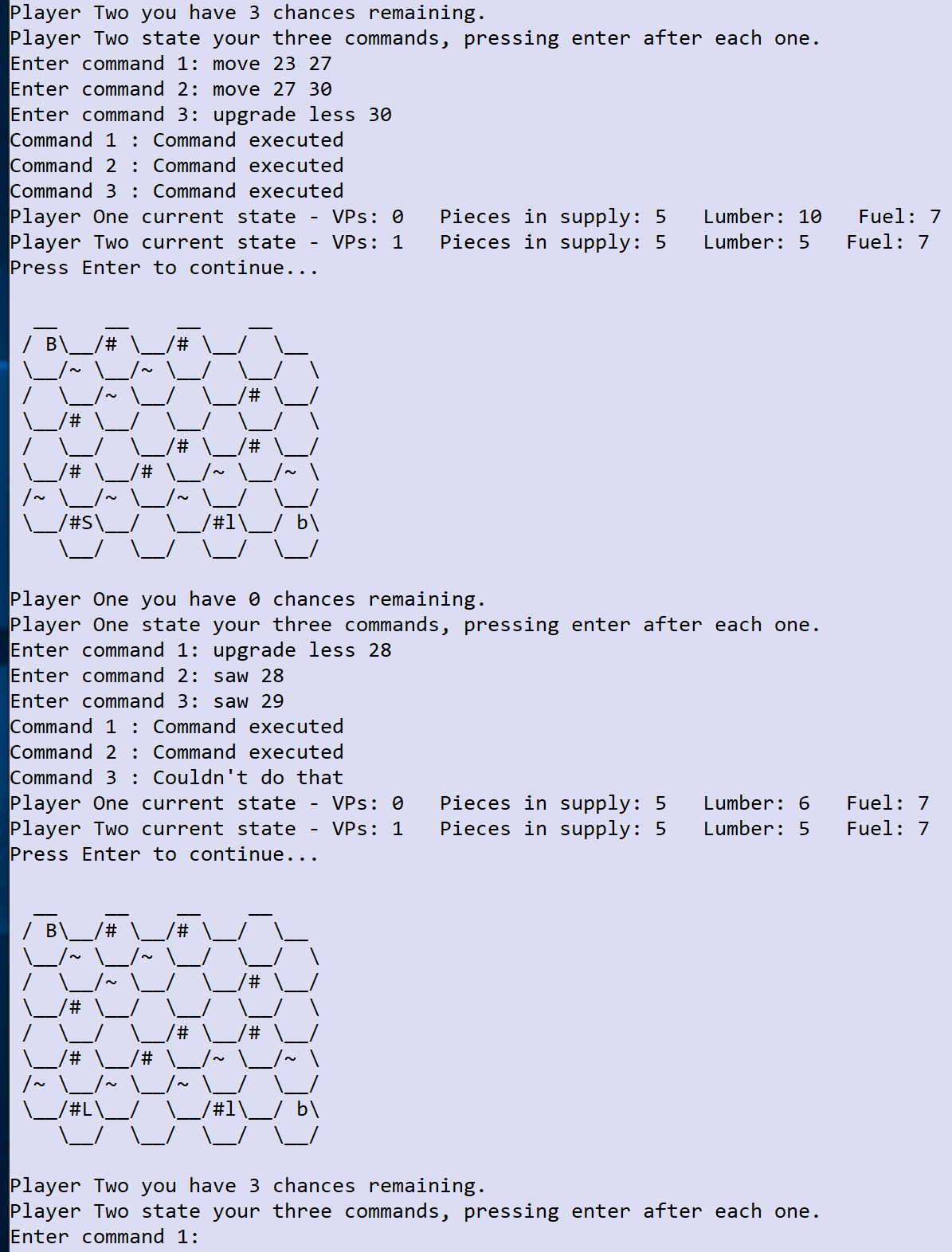
} while (!gameOver || !player1Turn);

Console.writeLine(grid.getGridAsString(player1Turn));

displayEndMessages(player1, player2);

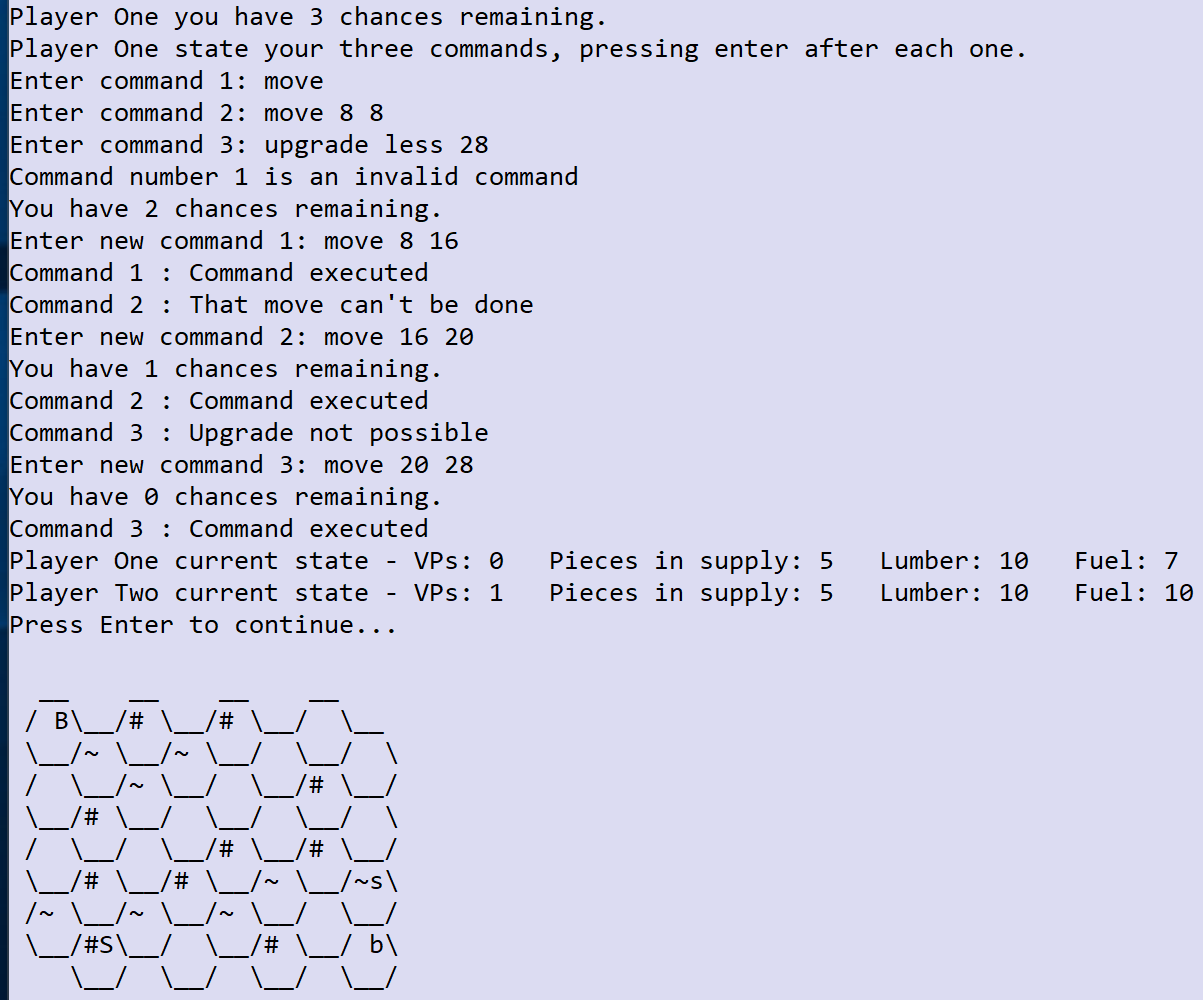
}

1. Solutions which retain the use of player1 and player2 throughout.

**Testing:**

* 1 mark for showing the commands entered correctly and the commands and error messages all being numbered clearly (even if the numbering is incorrect).
* 1 mark for displaying the number of chances correctly each turn and persisting them between turns (e.g. Player One’s second turn shows 0 chances remaining).
* 1 mark for displaying the final error message correctly and not letting Player One have a fourth chance.

For example:



# Task 12 (13 marks)

**Coding:**

* 1 mark for modifying the checkCommandIsValid method to process the new format for the spawn bomb variant of the spawn command.
* 1 mark for correctly checking the format of the new spawn command (ideally using the checkSpawnCommandFormat method they wrote).
* 1 mark for modifying the spawn command in executeSpawnCommand so that it will correctly spawn a bomb and charge 10 lumber (do not award if they don’t check for the availability of the lumber).
* 1 mark for having a new BombPiece class and setting the value of the fuelCostOfMove attribute to 2.
* 1 mark for having a mechanism to count the number of moves until primed and then detecting when the bomb is primed.
* 1 mark for having a method to destroy the bomb when it is killed due to two connections (prior to being primed).
* 1 mark for having a mechanism for exploding the bomb when it is triggered and destroying all the surrounding pieces (award even if this is done at the end of the turn).
* 1 mark for correctly destroying the bomb the moment that it is triggered and not at the end of the turn.
* 1 mark for correctly awarding all of the VPs for the explosion.
* 1 mark for making the bomb display as a Serf piece for the appropriate player.

For example:

Code for modified checkCommandIsValid subroutine:

case "spawn":

return checkSpawnCommandFormat(items);

Code for new checkSpawnCommandFormat subroutine:

boolean checkSpawnCommandFormat(List<String> items) {

int result;

if (items.size() == 2) {

return checkStandardCommandFormat(items);

} else if (items.size() == 3) {

if (!items.get(1).toLowerCase().equals("bomb")) {

return false;

}

try {

result = Integer.parseInt(items.get(2));

} catch (Exception e) {

return false;

}

return true;

}

return false;

}

Code for modified private executeSpawnCommand method:

private int executeSpawnCommand(List<String> items, int lumberAvailable, int piecesInSupply) {

int tileToUse;

int spawnCost;

if (items.size() == 2) {

spawnCost = 3;

tileToUse = Integer.parseInt(items.get(1));

} else {

spawnCost = 10;

tileToUse = Integer.parseInt(items.get(2));

}

if (piecesInSupply < 1 || lumberAvailable < spawnCost || !checkTileIndexIsValid(tileToUse)) {

return -1;

}

Piece thePiece = tiles.get(tileToUse).getPieceInTile();

if (thePiece != null) {

return -1;

}

boolean ownBaronIsNeighbour = false;

List<Tile> listOfNeighbours = new ArrayList<>(tiles.get(tileToUse).getNeighbours());

for (Tile n : listOfNeighbours) {

thePiece = n.getPieceInTile();

if (thePiece != null) {

if (player1Turn && thePiece.getPieceType().equals("B") || !player1Turn && thePiece.getPieceType().equals("b")) {

ownBaronIsNeighbour = true;

break;

}

}

}

if (!ownBaronIsNeighbour) {

return -1;

}

if (items.get(1).equals("bomb")) {

BombPiece newPiece = new BombPiece(player1Turn);

pieces.add(newPiece);

tiles.get(tileToUse).setPiece(newPiece);

} else {

Piece newPiece = new Piece(player1Turn);

pieces.add(newPiece);

tiles.get(tileToUse).setPiece(newPiece);

}

return spawnCost;

}

Code for new BombPiece class:

class BombPiece extends Piece {

private int movesBeforePrimed = 5;

public BombPiece(boolean player1) {

super(player1);

pieceType = "X";

vPValue = 5;

fuelCostOfMove = 2;

}

@Override

public int checkMoveIsValid(int distanceBetweenTiles, String startTerrain, String endTerrain) {

if (distanceBetweenTiles == 1) {

if (movesBeforePrimed > 0) {

movesBeforePrimed -= 1;

return fuelCostOfMove;

}

}

return -1;

}

public void destroyPiece(Tile tileLocation) {

if (!destroyed) {

connectionsToDestroy = 0;

movesBeforePrimed = -1;

destroyed = true;

List<Tile> listOfNeighbours = new ArrayList<>(tileLocation.getNeighbours());

for (Tile neighbour : listOfNeighbours) {

Piece tilePiece = neighbour.getPieceInTile();

if (tilePiece != null) {

tilePiece.explode();

}

}

}

}

public Object[] explode(Tile tileLocation) {

int player1VPs = 0;

int player2VPs = 0;

boolean baronDestroyed = false;

if (!destroyed) {

List<Tile> listOfNeighbours = new ArrayList<>(tileLocation.getNeighbours());

for (Tile neighbour : listOfNeighbours) {

Piece tilePiece = neighbour.getPieceInTile();

if (tilePiece != null) {

if (tilePiece.getBelongsToplayer1()) {

player2VPs += tilePiece.getVPs();

} else {

player1VPs += tilePiece.getVPs();

}

if (tilePiece.getPieceType().toUpperCase().equals("B")) {

baronDestroyed = true;

}

tilePiece.destroyPiece();

neighbour.setPiece(null);

}

}

destroyed = true;

}

return new Object[] {baronDestroyed, player1VPs, player2VPs};

}

public boolean primed() {

return movesBeforePrimed == 0;

}

}

Code for modified destroyPiecesAndCountVPs method:

public Object[] destroyPiecesAndCountVPs(int player1VPs, int player2VPs) {

boolean baronDestroyed = false;

List<Tile> listOfTilesContainingDestroyedPieces = new ArrayList<>();

for (Tile t : tiles) {

if (t.getPieceInTile() != null) {

List<Tile> listOfNeighbours = new ArrayList<>(t.getNeighbours());

int noOfConnections = 0;

for (Tile n : listOfNeighbours) {

if (n.getPieceInTile() != null) {

noOfConnections += 1;

}

}

Piece thePiece = t.getPieceInTile();

if (noOfConnections >= thePiece.getConnectionsNeededToDestroy()) {

if (thePiece.getPieceType().toUpperCase().equals("X")) {

BombPiece bomb = (BombPiece)thePiece;

Object[] returnObjects = bomb.explode(t);

baronDestroyed = (boolean)returnObjects[0];

player1VPs += (int)returnObjects[1];

player2VPs += (int)returnObjects[2];

} else {

thePiece.destroyPiece();

if (thePiece.getPieceType().toUpperCase().equals("B")) {

baronDestroyed = true;

}

}

listOfTilesContainingDestroyedPieces.add(t);

if (thePiece.getBelongsToplayer1()) {

player2VPs += thePiece.getVPs();

} else {

player1VPs += thePiece.getVPs();

}

}

}

}

for (Tile t : listOfTilesContainingDestroyedPieces) {

t.setPiece(null);

}

return new Object[]{baronDestroyed, player1VPs, player2VPs};

}

Code for modified getPieceTypeInTile method:

public String getPieceTypeInTile(int id) {

Piece thePiece = tiles.get(id).getPieceInTile();

if (thePiece == null) {

return " ";

} else {

String pieceType = thePiece.getPieceType();

if (pieceType.equals("X")) {

pieceType = "S";

} else if (pieceType.equals("x")) {

pieceType = "s";

}

return pieceType;

}

}

Code for modified executeMoveCommand method:

private int executeMoveCommand(List<String> items, int fuelAvailable) {

int startID = Integer.parseInt(items.get(1));

int endID = Integer.parseInt(items.get(2));

if (!checkPieceAndTileAreValid(startID) || !checkTileIndexIsValid(endID)) {

return -1;

}

Piece thePiece = tiles.get(startID).getPieceInTile();

if (tiles.get(endID).getPieceInTile() != null) {

return -1;

}

int distance = tiles.get(startID).getDistanceToTileT(tiles.get(endID));

int fuelCost = thePiece.checkMoveIsValid(distance, tiles.get(startID).getTerrain(), tiles.get(endID).getTerrain());

if (fuelCost == -1 || fuelAvailable < fuelCost) {

return -1;

}

movePiece(endID, startID);

List<Tile> listOfNeighbours = new ArrayList<>(tiles.get(endID).getNeighbours());

for (Tile neighbour : listOfNeighbours) {

Piece pieceInTile = neighbour.getPieceInTile();

if (pieceInTile!= null) {

if (pieceInTile.getPieceType().toUpperCase().equals("X")) {

BombPiece bomb = (BombPiece)pieceInTile;

if (bomb.primed()) {

bomb.destroyPiece(neighbour);

}

}

}

}

return fuelCost;

}

Code for modified Piece class:

public void explode() {

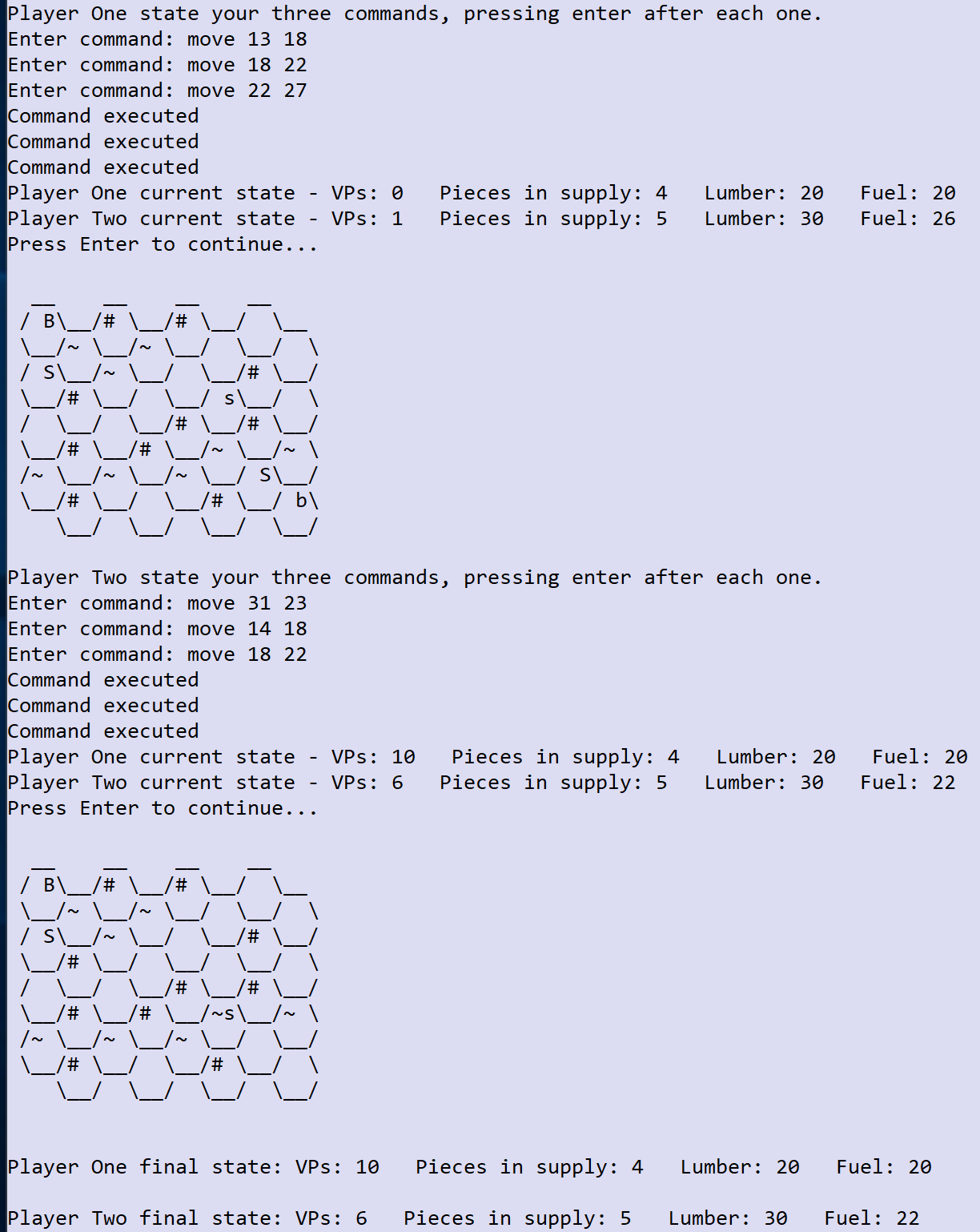
connectionsToDestroy = 0;

}

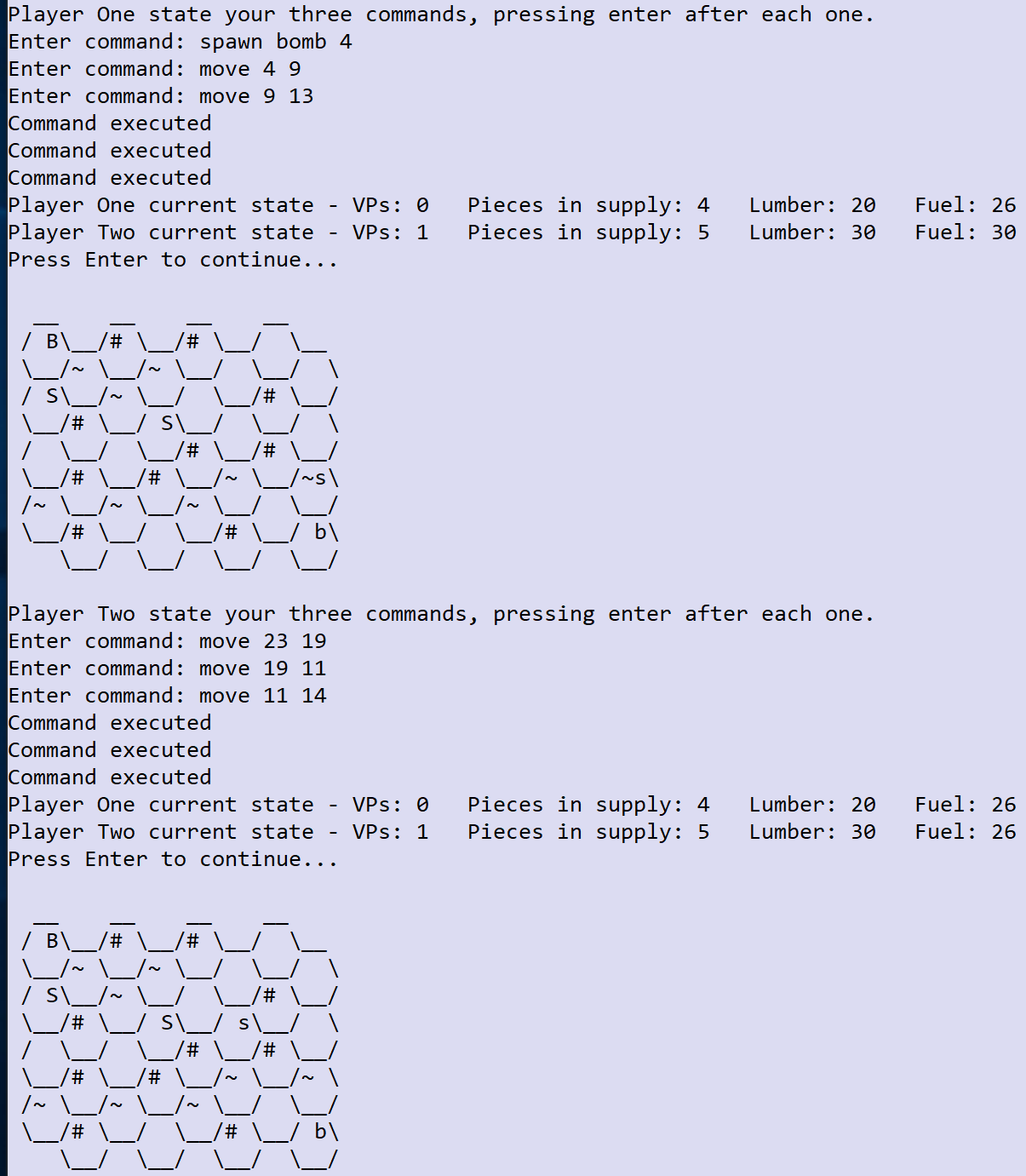
1. Solutions which solve the problem using a different approach and fulfil all of the criteria on the mark scheme, award full marks. When awarding partial marks the criteria can be interpreted slightly loosely if that seems reasonable given the nature of the alternative solution.

R. Solutions that break OOP by accessing private and protected attributes in ugly or unreasonable ways, or solutions that unnecessarily provide direct access to things when they could be solved in a more suitable way using the OOP hierarchy, encapsulation and polymorphism.

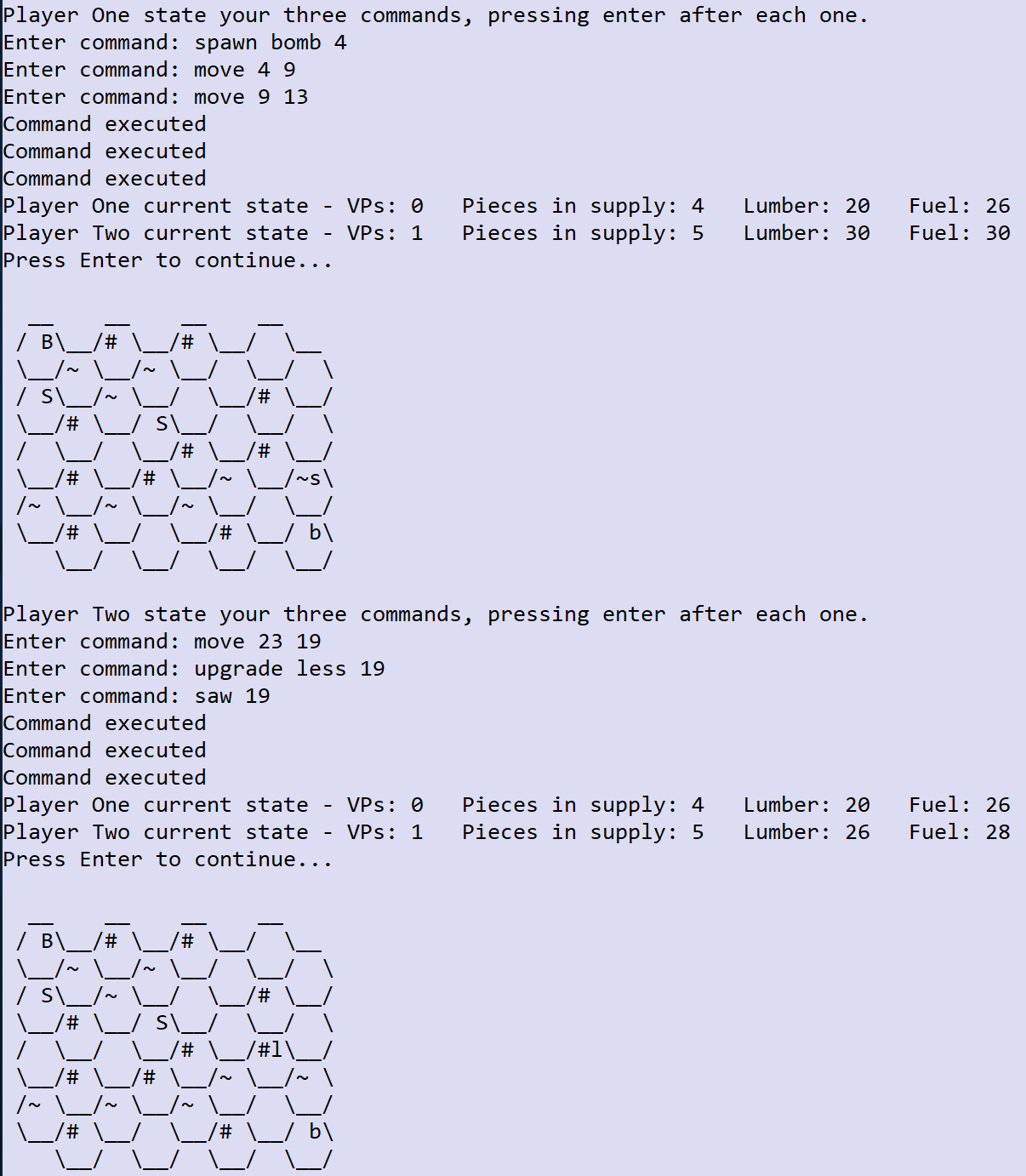
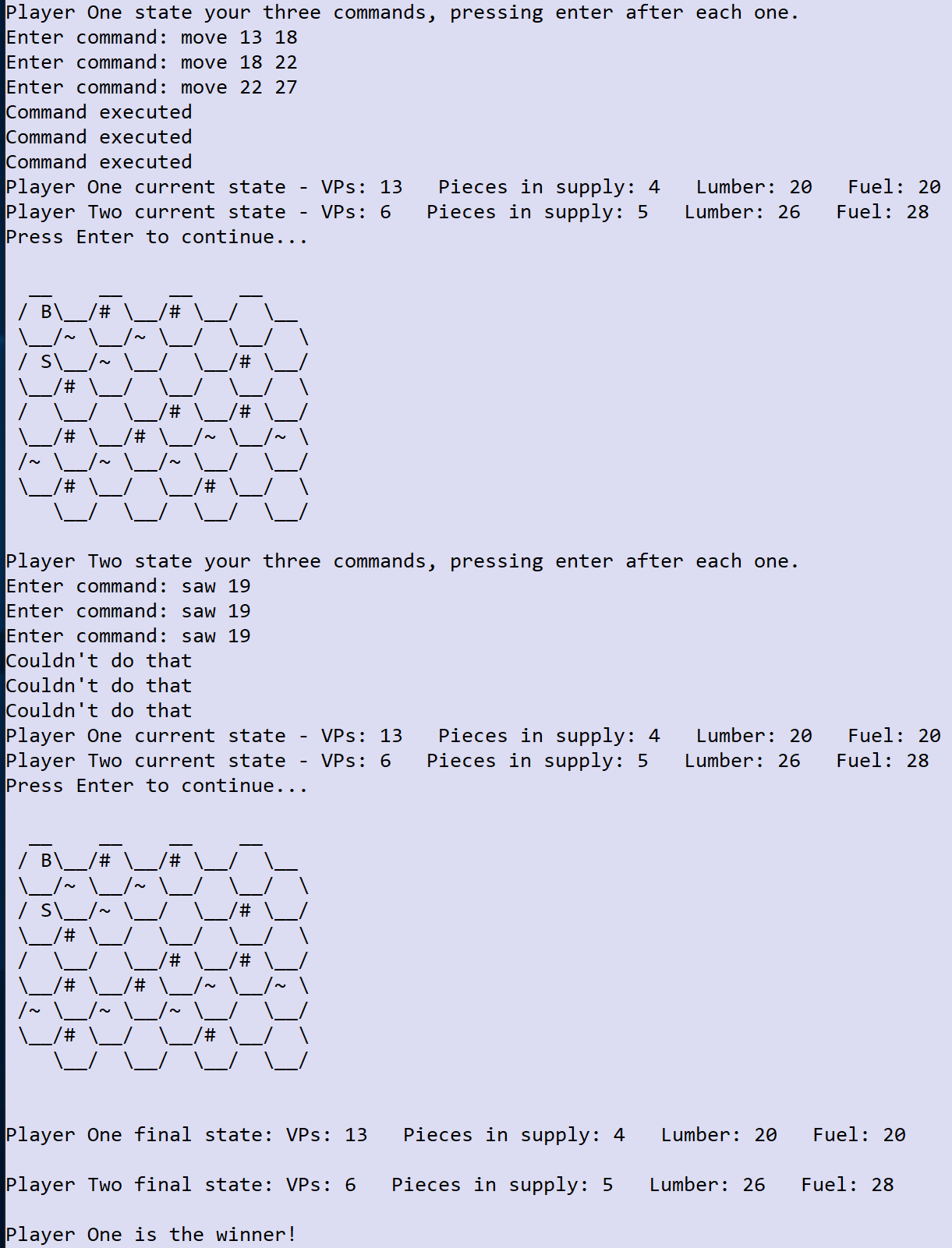
**Testing:**

* 1 mark for showing the commands entered correctly and all of the output from the computer (for both games).
* 1 mark for Player One winning the first game with the correct VP totals.
* 1 mark for Player One winning the second game with the correct VP totals.

For example – Game One:



For example – Game Two:



# Task 13 (10 marks)

**Coding:**

* 1 mark for creating the WizardPiece class with a VP value of 3.
* 1 mark for overriding the checkMoveIsValid method to implement the teleport.
* 1 mark for having a new attribute to track the use of the teleport.
* 1 mark for having a resetTeleport mechanism that is activated at the end or beginning of each turn.
* 1 mark for changing executeSpawnCommand to enable a wizard piece to be spawned.
* 1 mark for changing checkUpgradeCommandFormat to enable the new spawn command to be validated.
* 1 mark for allowing teleport exactly once per turn (only if there is enough fuel).
* 1 mark for getting teleport working for 2 or 3 square distances (even if they can do it multiple times in a turn) and charging 5 fuel but allowing moves of 1 for 1 fuel and blocking move of 4 or more.

For example:

Code for the new WizardPiece class:

class WizardPiece extends Piece {

private boolean teleported;

public WizardPiece(boolean player1) {

super(player1);

pieceType = "W";

vPValue = 3;

resetTeleport();

}

@Override

public int checkMoveIsValid(int distanceBetweenTiles, String startTerrain, String endTerrain) {

if (distanceBetweenTiles == 1) {

return fuelCostOfMove;

} else if (!teleported && (distanceBetweenTiles >= 2) && (distanceBetweenTiles <= 3)) {

teleported = true;

return 5;

}

return -1;

}

public void resetTeleport() {

teleported = false;

}

}

Code for modified checkUpgradeCommandFormat subroutine:

boolean checkUpgradeCommandFormat(List<String> items) {

int result;

if (items.size() == 3) {

if (!items.get(1).toUpperCase().equals("LESS") && !items.get(1).toUpperCase().equals("PBDS") &&

!items.get(1).toUpperCase().equals("WIZ")) {

return false;

}

try {

result = Integer.parseInt(items.get(2));

} catch (Exception e) {

return false;

}

return true;

}

return false;

}

Code for modified executeUpgradeCommand method:

private int executeUpgradeCommand(List<String> items, int lumberAvailable) {

int tileToUse = Integer.parseInt(items.get(2));

if (!checkPieceAndTileAreValid(tileToUse) || lumberAvailable < 5 || !((items.get(1).equals("pbds") ||

items.get(1).equals("less") || items.get(1).equals("wiz")))) {

return -1;

} else {

Piece thePiece = tiles.get(tileToUse).getPieceInTile();

if (!thePiece.getPieceType().toUpperCase().equals("S")) {

return -1;

}

thePiece.destroyPiece();

if (items.get(1).equals("pbds")) {

thePiece = new PBDSPiece(player1Turn);

} else if (items.get(1).equals("less")) {

thePiece = new LESSPiece(player1Turn);

} else {

thePiece = new WizardPiece(player1Turn);

}

pieces.add(thePiece);

tiles.get(tileToUse).setPiece(thePiece);

return 5;

}

}

Code for new method resetWizards in the HexGrid class:

public void resetWizards() {

for (Tile t : tiles) {

Piece thePiece = t.getPieceInTile();

if (thePiece != null) {

if (thePiece.getPieceType().toUpperCase().equals("W")) {

((WizardPiece)thePiece).resetTeleport();

}

}

}

}

Code for new call to resetWizards in the playGame subroutine:

}

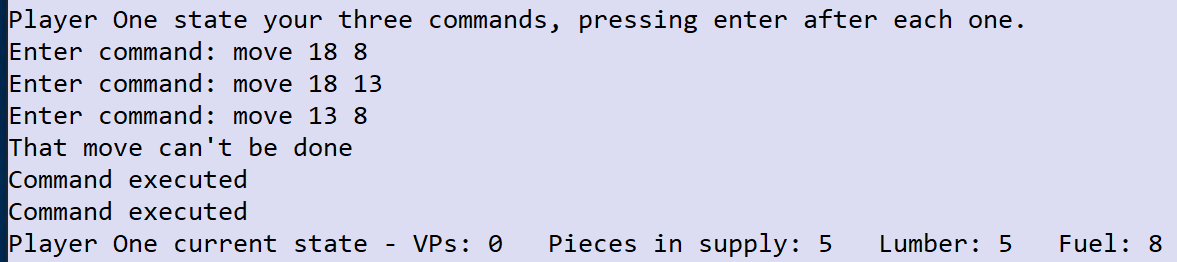
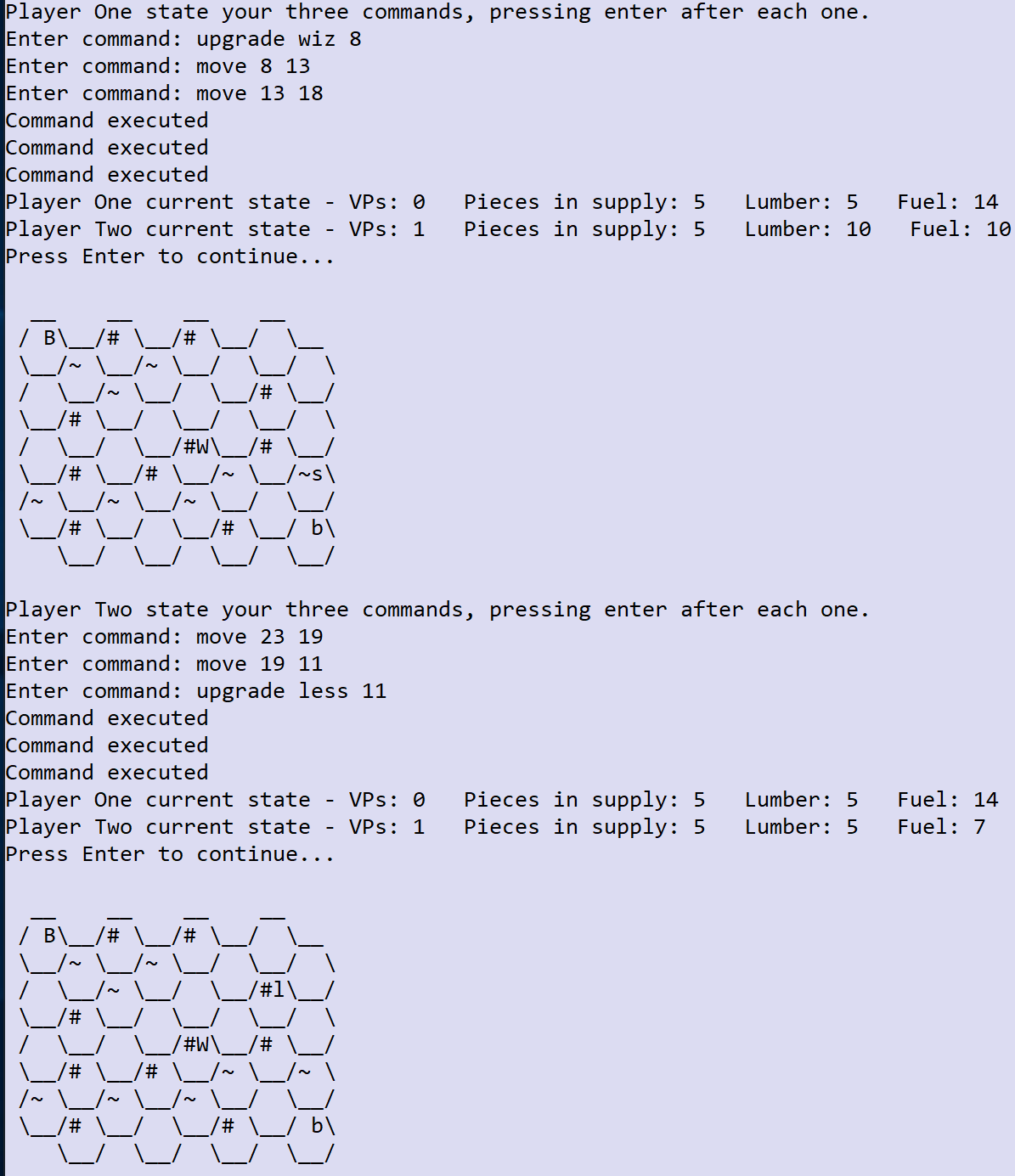
grid.resetWizards();

commands.clear();

**Testing:**

* 1 mark for showing the commands entered correctly and the error message for the move 18 8 command and then correctly executing the following two commands.
* 1 mark for correctly teleporting the piece to location 13 (and then onto 18) in the second board printout.

For example:



# Task 14 (10 marks)

**Coding:**

* 1 mark for creating the new SCFWPiece class and setting the vPValue to 3.
* 1 mark for overriding the checkMoveIsValid method (or any other reasonable way of disabling the move command for this piece).
* 1 mark for implementing the supply command in the SCFWPiece class and checking that there is enough fuel and lumber.
* 1 mark for making it so that the pieces in supply, lumber and fuel are updated correctly when the supply command is run.
* 1 mark for implementing the upgrade scfw command and allowing it to run.
* 1 mark for displaying the new piece as ‘F’ for Player One and charging 3 lumber for the upgrade (and making sure it’s available).
* 1 mark for checking whether the first command entered is a supply command and terminating the loop early if it is.

For example:

Code for new SCFWPiece class:

class SCFWPiece extends Piece {

public SCFWPiece (boolean player1) {

super(player1);

pieceType = "F";

vPValue = 3;

}

@Override

public int checkMoveIsValid(int distanceBetweenTiles, String startTerrain, String endTerrain) {

return -1;

}

public Object[] supply(int fuelAvailable, int lumberAvailable) {

Object[] returnObjects;

if ((fuelAvailable >= 5) && (lumberAvailable >= 5)) {

returnObjects = new Object[]{-1,-5,-5};

} else {

returnObjects = new Object[]{0,0,0};

}

return returnObjects;

}

}

Code for the modified checkCommandIsValid subroutine:

case "move":

return checkMoveCommandFormat(items);

case "supply":

if (!supplyFirst) {

return false;

}

case "dig":

case "saw":

case "spawn":

return checkStandardCommandFormat(items);

Code for the modified checkUpgradeCommandFormat subroutine:

boolean checkUpgradeCommandFormat(List<String> items) {

int result;

if (items.size() == 3) {

if (!items.get(1).toUpperCase().equals("LESS") && !items.get(1).toUpperCase().equals("PBDS") &&

!items.get(1).toUpperCase().equals("SCFW")) {

return false;

}

try {

result = Integer.parseInt(items.get(2));

} catch (Exception e) {

return false;

}

return true;

}

return false;

}

Code for modified method executeUpgradeCommand of the HexGrid class:

if (!checkPieceAndTileAreValid(tileToUse) || lumberAvailable < 5 || !((items.get(1).equals("pbds") ||

items.get(1).equals("less") || items.get(1).equals("scfw")))) {

return -1;

} else {

…

if (items.get(1).equals("pbds")) {

thePiece = new PBDSPiece(player1Turn);

} else if (items.get(1).equals("less")){

thePiece = new LESSPiece(player1Turn);

} else {

thePiece = new SCFWPiece(player1Turn);

}

pieces.add(thePiece);

Code for modified executeCommand method:

case "supply":

Object[] returnObjects = executeSupplyCommand(items, fuelAvailable, lumberAvailable);

execute = (boolean)returnObjects[0];

supplyChange = (int)returnObjects[1];

fuelChange = (int)returnObjects[2];

lumberChange = (int)returnObjects[3];

if (!execute) {

return new Object[] {"Couldn't do that", fuelChange, lumberChange, supplyChange};

}

break;

Code for new executeSupplyCommand method:

private Object[] executeSupplyCommand(List<String> items, int fuelAvailable, int lumberAvailable) {

int tileToUse = Integer.parseInt(items.get(1));

int fuel = 0;

int lumber = 0;

int supply = 0;

if (checkPieceAndTileAreValid(tileToUse) == false) {

return new Object[] {false, supply, fuel, lumber};

}

Piece thePiece = tiles.get(tileToUse).getPieceInTile();

if ((thePiece != null) && (thePiece.getPieceType().toUpperCase().equals("F"))) {

Object[] returnObjects = ((SCFWPiece)thePiece).supply(fuelAvailable, lumberAvailable);

supply = (int)returnObjects[0];

fuel = (int)returnObjects[1];

lumber = (int)returnObjects[2];

if (supply == -1) {

return new Object[] {true, supply, fuel, lumber};

}

}

return new Object[] {false, supply, fuel, lumber};

}

Code for modified playGame subroutine:

void playGame(Player player1, Player player2, HexGrid grid) {

boolean gameOver = false;

boolean player1Turn = true;

boolean validCommand;

boolean supplyFirst = false;

List<String> commands = new ArrayList<>();

Console.writeLine("Player One current state - " + player1.getStateString());

Console.writeLine("Player Two current state - " + player2.getStateString());

do {

Console.writeLine(grid.getGridAsString(player1Turn));

if (player1Turn) {

Console.writeLine(player1.getName() + " state your three commands, pressing enter after each one.");

} else {

Console.writeLine(player2.getName() + " state your three commands, pressing enter after each one.");

}

for (int count = 1; count <= 3; count++) {

String command;

Console.write("Enter command: ");

command = Console.readLine().toLowerCase();

commands.add(command);

List<String> items = Arrays.asList(command.split(" "));

if (items.get(0).toLowerCase().equals("supply") && count == 1) {

supplyFirst = true;

break;

}

}

for (String c : commands) {

List<String> items = Arrays.asList(c.split(" "));

validCommand = checkCommandIsValid(items, supplyFirst);

if (!validCommand) {

Console.writeLine("Invalid command");

} else {

int fuelChange = 0;

int lumberChange = 0;

int supplyChange = 0;

String summaryOfResult;

Object[] returnObjects;

if (player1Turn) {

returnObjects = grid.executeCommand(items, fuelChange, lumberChange, supplyChange, player1.getFuel(),

player1.getLumber(), player1.getPiecesInSupply());

summaryOfResult = returnObjects[0].toString();

fuelChange = (int)returnObjects[1];

lumberChange = (int)returnObjects[2];

supplyChange = (int)returnObjects[3];

player1.updateLumber(lumberChange);

player1.updateFuel(fuelChange);

if (supplyChange == 1) {

player1.removeTileFromSupply();

} else if (supplyChange == -1) {

player1.addPieceToSupplyChain();

}

} else {

returnObjects = grid.executeCommand(items, fuelChange, lumberChange, supplyChange, player2.getFuel(),

player2.getLumber(), player2.getPiecesInSupply());

summaryOfResult = returnObjects[0].toString();

fuelChange = (int)returnObjects[1];

lumberChange = (int)returnObjects[2];

supplyChange = (int)returnObjects[3];

player2.updateLumber(lumberChange);

player2.updateFuel(fuelChange);

if (supplyChange == 1) {

player2.removeTileFromSupply();

} else if (supplyChange == -1) {

player2.addPieceToSupplyChain();

}

}

Console.writeLine(summaryOfResult);

…

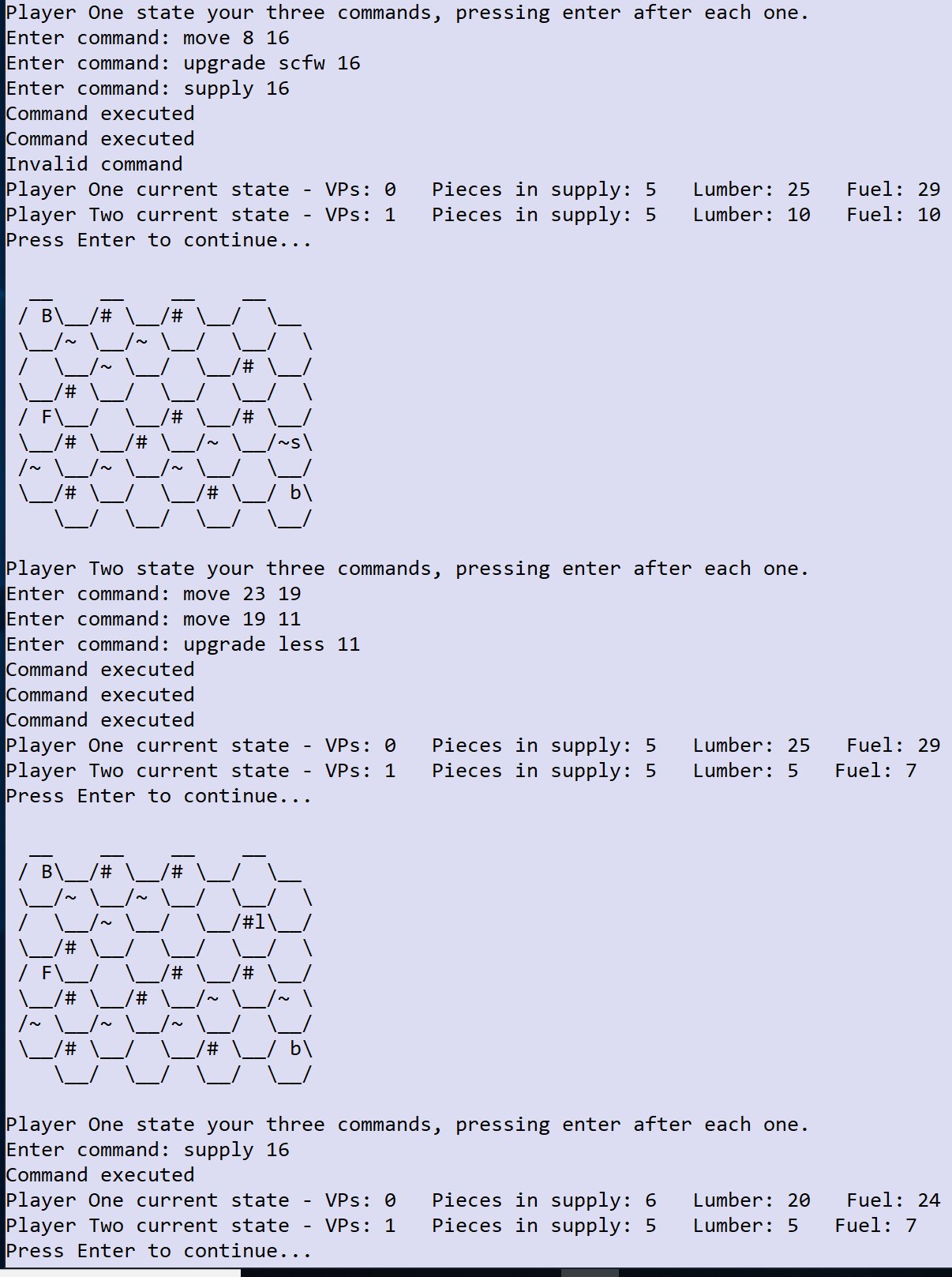
Code for new addPieceToSupplyChain method:

public void addPieceToSupplyChain() {

piecesInSupply += 1;

}

1. Solutions which use an alternative way of making sure that the supply command can only be the first command in the list of commands (as long as they terminate input early when the supply command is typed in first).
2. Solutions which use hasMethod in a similar way to dig and saw instead of checking the piece type and then downcasting the parent object to the child object.

**Testing:**

* 1 mark for showing that the upgrade scfw command worked correctly.
* 1 mark for showing that the first supply 16 command didn’t work, with a suitable error message.
* 1 mark for showing that the second supply 16 command worked successfully and that input was terminated when it was entered.

For example:

# Task 15 (10 marks)

**Coding:**

* 1 mark for modifying the main menu to contain a new option 3 for creating a game.
* 1 mark for changing the loadGame subroutine so that it accepts the fileName as a parameter correctly and doesn’t ask for it to be input inside the subroutine, but asks in the main program prior to calling loadGame.
* 1 mark for correctly returning the success and fileName parameters and using them to ask whether the user would like to start a game and then passing through the appropriate return value as the fileName to loadGame.
* 1 mark for validating the grid size so that the user can only enter 6, 8 or 10.
* 1 mark for printing out the board correctly, showing all the index numbers for each location.
* 1 mark for accepting the terrain input correctly and using it to generate a string to write to the file (even if not printed out to the screen).
* 1 mark for accepting the starting values for each player (VPs, fuel, lumber and supply chain) and writing them to the file.
* 1 mark for accepting more than one piece being entered (in addition to the Baron) for each player.

For example:

displayMainMenu subroutine:

void displayMainMenu() {

Console.writeLine("1. Default game");

Console.writeLine("2. Load game");

Console.writeLine("3. Create game");

Console.writeLine("Q. Quit");

Console.writeLine();

Console.write("Enter your choice: ");

}

Main program (HexBaron):

} else if (choice.equals("2")) {

Console.write("Enter the name of the file to load: ");

fileName = "aqa/hex/baron/"+Console.readLine();

Object[] returnObjects = loadGame(fileName, player1, player2);

fileLoaded = (boolean)returnObjects[1];

if (fileLoaded) {

grid = (HexGrid)returnObjects[0];

playGame(player1, player2, grid);

}

} else if (choice.equals("3")) {

Object[] returnObject = createGame();

boolean success = (boolean)returnObject[0];

fileName = (String)returnObject[1];

if (success) {

Console.write("Would you like to play the game that you've created and saved? ");

String answer = Console.readLine().toLowerCase();

if (answer.equals("yes") || answer.equals("y")) {

returnObject = loadGame("aqa/hex/baron/"+fileName, player1, player2);

fileLoaded = (boolean)returnObject[1];

if (fileLoaded) {

grid = (HexGrid)returnObject[0];

playGame(player1, player2, grid);

}

}

} else {

Console.writeLine("Sorry but the game could not be created.");

}

}

Import statements added to the main program:

import java.io.IOException;

import java.io.FileWriter;

loadGame subroutine:

Object[] loadGame(String fileName, Player player1, Player player2) {

List<String> items;

createGame subroutine:

Object[] createGame() {

List<String> items;

String lineFromFile;

String fileName;

HexGrid grid;

Player player1 = new Player();

Player player2 = new Player();

int gridSize;

List<String> forests;

List<String> peatBogs;

List<String> terrainList = new ArrayList<String>();

List<String> newTerrain = new ArrayList<String>();

List<Integer> p1stats = new ArrayList<Integer>();

List<Integer> p2stats = new ArrayList<Integer>();

List<String> p1pieces = new ArrayList<String>();

List<String> p2pieces = new ArrayList<String>();

List<Integer> p1locs = new ArrayList<Integer>();

List<Integer> p2locs = new ArrayList<Integer>();

String nextPiece = "";

Console.writeLine("Welcome to the Hex Baron Game Creator");

Console.writeLine("=====================================");

Console.writeLine();

Console.write("Please enter the grid size (6, 8 or 10): ");

gridSize = Integer.parseInt(Console.readLine());

while (gridSize != 6 && gridSize != 8 && gridSize != 10) {

Console.write("Invalid grid size, please re-enter (6, 8 or 10): ");

gridSize = Integer.parseInt(Console.readLine());

}

grid = new HexGrid(gridSize);

for (int terrainIndex=0; terrainIndex<(gridSize\*gridSize/2); terrainIndex++) {

terrainList.add(" ");

}

grid.setUpGridTerrain(terrainList);

Console.writeLine(grid.getGridAsIndices());

Console.write("Please enter all the locations of peat bogs separated by commas with no spaces: ");

peatBogs = Arrays.asList(Console.readLine().split(","));

Console.write("Please enter all the locations of forests separated by commas with no spaces: ");

forests = Arrays.asList(Console.readLine().split(","));

for (int terrainIndex=0; terrainIndex<terrainList.size(); terrainIndex++) {

String hex = ""+terrainIndex;

if (peatBogs.contains(hex)) {

newTerrain.add("~");

} else if (forests.contains(hex)) {

newTerrain.add("#");

} else {

newTerrain.add(" ");

}

}

grid.setUpGridTerrain(newTerrain);

Console.write("Please enter the location of the Baron for Player 1: ");

p1locs.add(Integer.parseInt(Console.readLine()));

p1pieces.add("Baron");

grid.addPiece(true,"Baron",p1locs.get(0));

Console.write("Please enter the location of the Baron for Player 2: ");

p2locs.add(Integer.parseInt(Console.readLine()));

p2pieces.add("Baron");

grid.addPiece(false,"Baron",p2locs.get(0));

Console.writeLine(grid.getGridAsString(true));

Console.write("Please enter the starting amount of VPs for Player 1: ");

p1stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of VPs for Player 2: ");

p2stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of fuel for Player 1: ");

p1stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of fuel for Player 2: ");

p2stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of lumber for Player 1: ");

p1stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of lumber for Player 2: ");

p2stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of pieces in the supply chain for Player 1: ");

p1stats.add(Integer.parseInt(Console.readLine()));

Console.write("Please enter the starting amount of pieces in the supply chain for Player 2: ");

p2stats.add(Integer.parseInt(Console.readLine()));

while (!nextPiece.equals("D")) {

Console.write("Enter piece to add for Player One (S-Serf,L-Less,P-PBDS) or D for Done: ");

nextPiece = Console.readLine().toUpperCase();

if (!nextPiece.equals("D")) {

Console.write("Enter the index of the hex where that piece should be placed: ");

p1locs.add(Integer.parseInt(Console.readLine()));

switch (nextPiece) {

case "S":

p1pieces.add("Serf");

break;

case "P":

p1pieces.add("PBDS");

break;

case "L":

p1pieces.add("LESS");

break;

}

grid.addPiece(true,p1pieces.get(p1pieces.size()-1),p1locs.get(p1locs.size()-1));

}

}

nextPiece = "";

while (!nextPiece.equals("D")) {

Console.write("Enter piece to add for Player Two (S-Serf,L-Less,P-PBDS) or D for Done: ");

nextPiece = Console.readLine().toUpperCase();

if (!nextPiece.equals("D")) {

Console.write("Enter the index of the hex where that piece should be placed: ");

p2locs.add(Integer.parseInt(Console.readLine()));

switch (nextPiece) {

case "S":

p2pieces.add("Serf");

break;

case "P":

p2pieces.add("PBDS");

break;

case "L":

p2pieces.add("LESS");

break;

}

grid.addPiece(false,p2pieces.get(p2pieces.size()-1),p2locs.get(p2locs.size()-1));

}

}

Console.writeLine(grid.getGridAsString(true));

try {

String p1string = "Player One";

String p2string = "Player Two";

String gridString = newTerrain.get(0);

Console.write("What name would you like to save the file as (please include the extension): ");

fileName = Console.readLine();

FileWriter out = new FileWriter("aqa/hex/baron/"+fileName);

for (int stat : p1stats) {

p1string += ","+stat;

}

out.write(p1string+"\n");

for (int stat : p2stats) {

p2string += ","+stat;

}

out.write(p2string+"\n");

out.write(""+gridSize+"\n");

for (int terrainIndex=1; terrainIndex<newTerrain.size(); terrainIndex++) {

gridString += ","+newTerrain.get(terrainIndex);

}

out.write(gridString+"\n");

for (int p1Index=0; p1Index<p1pieces.size(); p1Index++) {

out.write("1,"+p1pieces.get(p1Index)+","+p1locs.get(p1Index)+"\n");

}

for (int p2Index=0; p2Index<p2pieces.size(); p2Index++) {

out.write("2,"+p2pieces.get(p2Index)+","+p2locs.get(p2Index)+"\n");

}

out.close();

} catch (IOException e) {

Console.writeLine("File could not be saved.");

return new Object[]{false, null};

}

return new Object[]{true, fileName};

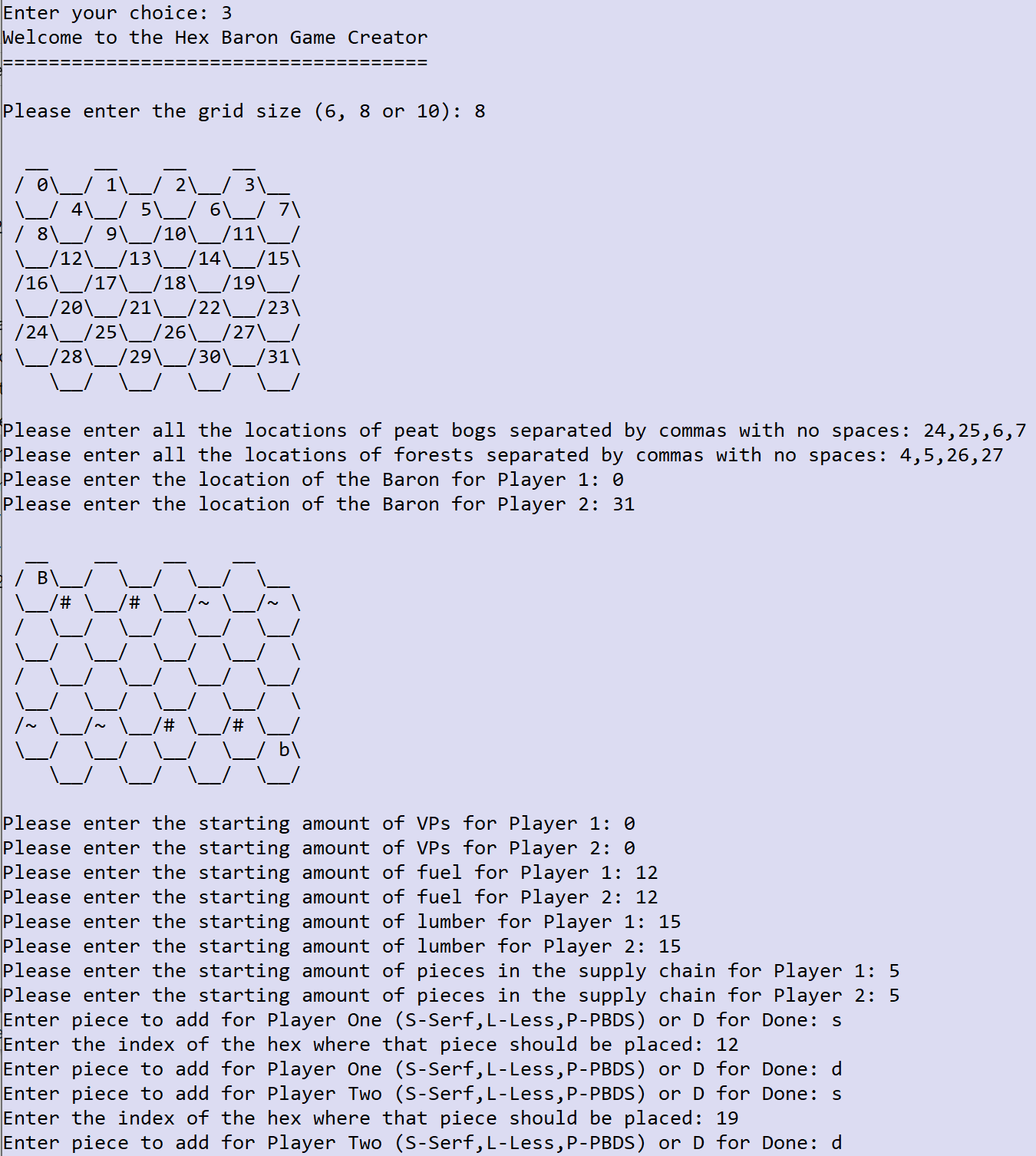
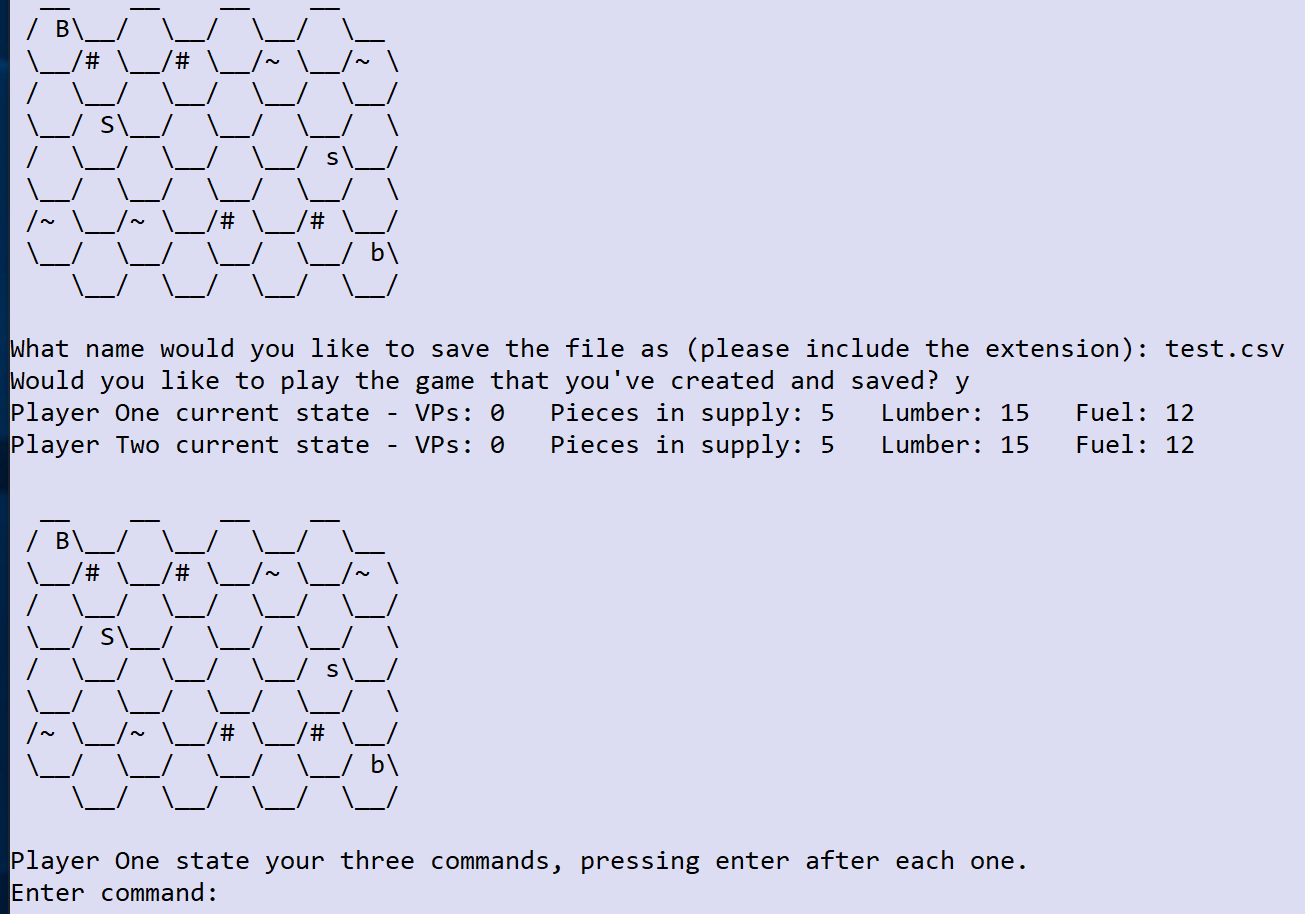
}

1. Solutions which have alternative input mechanisms or file printing mechanisms/structures as long as the code works.
2. Also solutions which handle the file pathing in different ways are perfectly acceptable.

**Testing:**

* 1 mark for showing the commands entered correctly and appropriate prompts, including the board printout showing the hex index numbers and the game starting correctly at the end.
* 1 mark for the test.csv file appearing exactly as shown.

For example:



**test.csv** file:

Player One,0,12,15,5

Player Two,0,12,15,5

8

, , , ,#,#,~,~, , , , , , , , , , , , , , , , ,~,~,#,#, , , ,

1,Baron,0

1,Serf,12

2,Baron,31

2,Serf,19