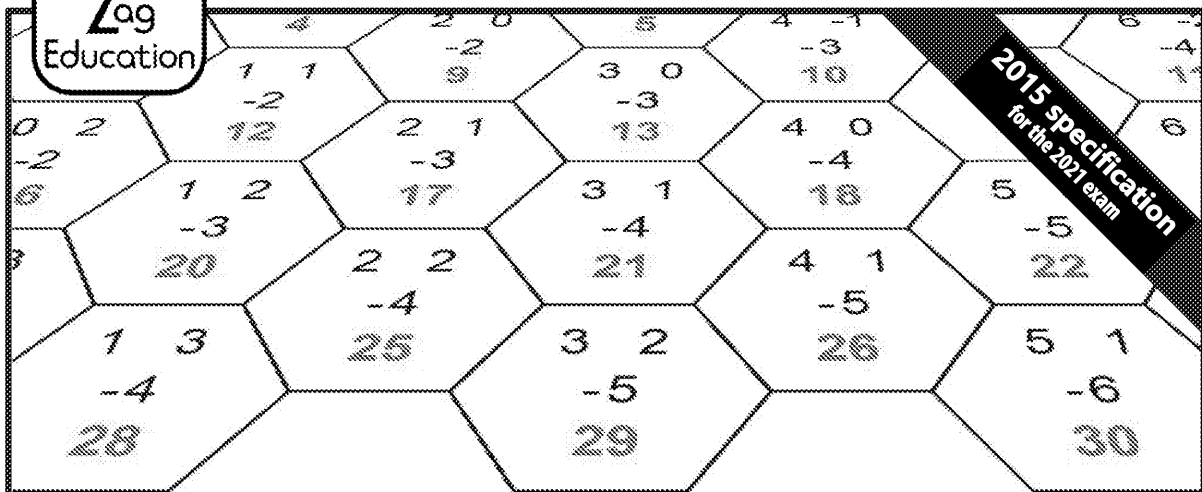




PAPER 1 EXAM RESOURCE PACK 2021

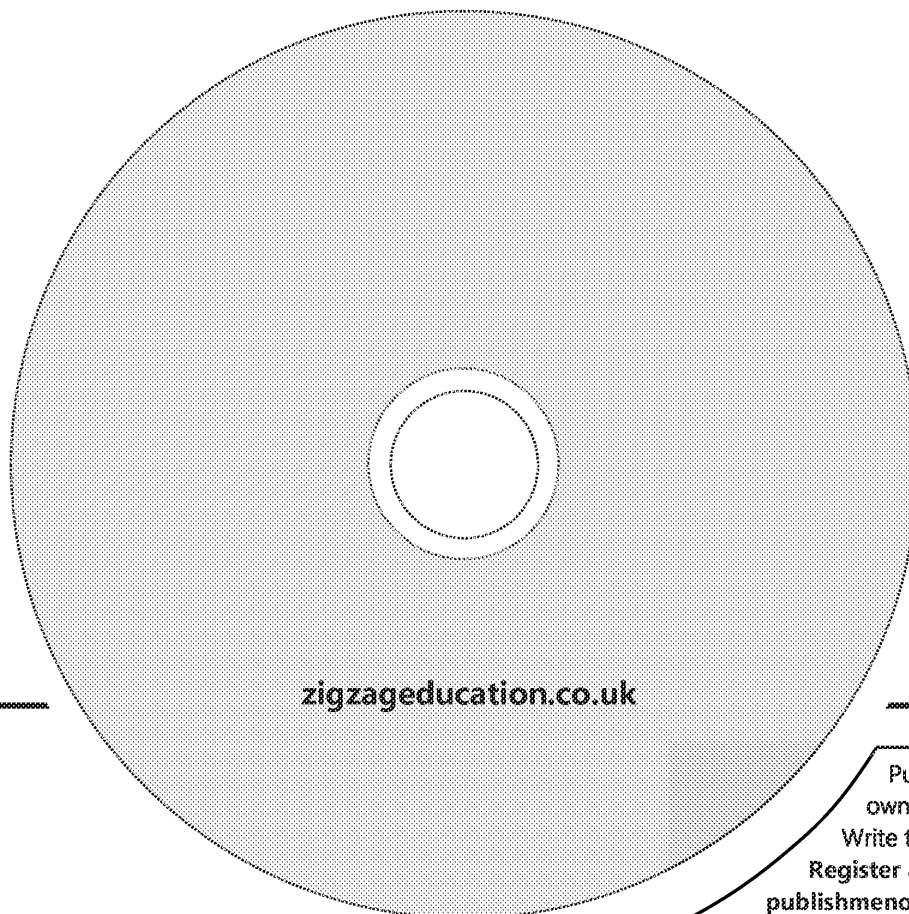
HEX BARON

for A Level AQA Computer Science

JAVA EDITION

DE8/
10585

POD
10585



zigzageducation.co.uk

Publish your
own work...
Write to a brief...
Register at
publishmenow.co.uk

Contents

Product Support from ZigZag Education	ii
Terms and Conditions of Use	iii
Teacher's Introduction	iv

Printouts of CD resources (for reference)

- Commentary (12 pages)
- Class Diagram Tasks (3 pages)
- Theory Questions: Write-on version (5 pages)
- Theory Questions: Non-write-on version (3 page)
- Programming Tasks (17 pages)
- Additional Tasks (Extension) (1 page)
- Class Diagram Tasks: Solutions (3 pages)
- Class Diagram Full (1 pages)
- Theory Questions: Mark Scheme (3 pages)
- Programming Tasks: Mark Scheme (49 pages)
- Electronic Answer Document (4 pages)

Teacher's Introduction

This resource pack is designed to help you support your students taking the A Level Computer Science Paper 1 exam. It is based on the *Hex Baron* preliminary material (Java) – for examination summer 2021.

On the CD, you will find the following:



- | | |
|---|--|
|  HexBaron | this folder contains all of the content (PDF/DOCX) accessible via a HTML interface |
|  Passwords.txt | for teacher use – this file contains all of the passwords for the protected PDFs (also listed below) |

* PRINTED COPIES OF ALL THE MATERIALS IN THIS DIGITAL RESOURCE PACK ARE INCLUDED FOR REFERENCE.

Installation: Copy the entire HexBaron folder onto a network location that is accessible for students, and provide them with a shortcut to the index.html file. All content can be accessed from this page.

Passwords: All of the PDFs accessible via the *Solutions* web page are password-protected, so that students can only access them with your permission. Each password is a four-digit code, as follows:

-  j06a-ClassDiagramTasksMS.pdf
-  j06b-ClassDiagramFull.pdf
-  j07-TheoryQuestionsMS.pdf
-  j08-ProgrammingTasksMS.pdf

The resource pack consists of the following:

- ① **Pre-release Commentary** including a general explanation of how the program works (text and video), plus a more detailed, technical overview* describing each subroutine class and method in turn.

***Note:** although this section is intended to give extra support to teachers and students, it should in no way be seen as a substitute to a student exploring the code for themselves.

- ② **Class Diagram Tasks**

Three class diagram tasks for students to complete while getting to grips with the skeleton program. A mark scheme is provided via the *Solutions* web page as a password-protected PDF.

- ③ **Written Questions**

Theory questions testing students' understanding of the skeleton program, like Section C in the exam. These questions require access to the program, but no modifications need to be made to the program. Write-on (with answer lines) and non-write-on version are available format. Suggested answers are provided via the *Solutions* web page as a password-protected PDF.

- ④ **Programming Tasks**

Fifteen modification exercises put students' programming skills to the test, like Section D in the exam. Example solutions with suggested mark schemes are provided via the *Solutions* web page as a password-protected PDF. Note that these are example solutions and you must use your discretion to award marks accordingly where there are valid alternative solutions.

An **Electronic Answer Document (EAD)** is provided should you wish students to use it for ③ and/or ④ above.

Teacher note (repeated in the Commentary section):

Although the skeleton code is class-based, the HexBaron class is considered to be the main program. It is referred to as such throughout this resource and methods in the HexBaron class are hence referred to as subroutines. This is for two reasons, firstly to keep it in line with the way that the resource is presented in the other languages (which is therefore more likely to match any exam content) and secondly because this is the logical function of the HexBaron class given that all Java programs only contain classes.

This resource is intended to supplement your teaching only. Please read full disclaimer (p. iii) before using it.

HEX BARON -- Commentary

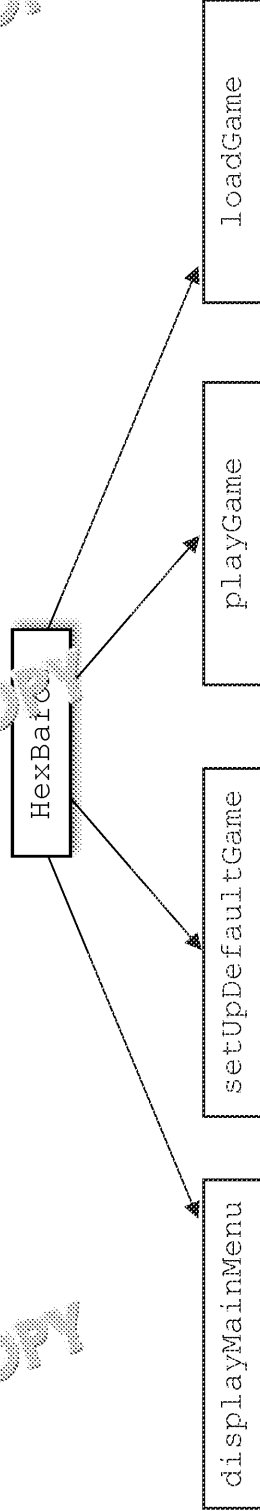
Hex Baron is a two-player game in which the main objective is to gain as many Victory Points (VPs) as possible, normally by killing your opponent's Baron. All adjacent hexes count as a distance of 1 and are legal moves, although for some pieces this will cost 2 fuel. It is best to start by creating a LESS and then you will have enough lumber to be able to spawn another Serf and start digging for fuel. In the game you have to enter three moves before any of them are valid, so if you make a mistake with the first move, you should end up losing your entire turn of all three moves, as it might affect your second and third moves. You thought you'd achieved something that had not failed.

Please watch the video explanation for a better understanding of the game mechanics.

Important: note that although the whole solution is class-based, the HexBaron class is considered to be the main program. It is referred to as such throughout this resource, and methods in the HexBaron class are hence referred to as subroutines. This is for two reasons, firstly to keep it in line with the way that the resource is presented in the other languages (which is therefore more likely to match any exam content) and secondly because this is the logical function of the HexBaron class given that all Java programs only contain classes.

Subroutines

Subroutines (Main Program): Hierarchy Diagram



COPYRIGHT
PROTECTED



INSPECTION COPY

Subroutines (All)

* indicates parameters passed by reference, all other parameters are passed by value

Name	Data	Description
checkCommandIsValid	Parameters: items (list of strings) Return value(s): Boolean	Depending on the first string in the list items, this calls one of the other checkCommandFormat subroutines.
checkMoveCommandFormat	Parameters: items (list of strings) Return value(s): Boolean	Returns true if the following conditions are met: 1) There are three elements in the items list 2) The second and third items are strings containing integers Otherwise it returns false.
checkStandardCommandFormat	Parameters: items (list of strings) Return value(s): Boolean	Returns true if the following conditions are met: 1) There are two elements in the items list 2) The second item is a string containing an integer Otherwise it returns false.
checkUpgrdCommandFormat	Parameters: items (list of strings) Return value(s): Boolean	Returns true if the following conditions are met: 1) There are three elements in the items list 2) The third item is a string containing an integer 3) The second item is either less or greater than (any case) Otherwise it returns false.
displayEndMessages	Parameters: player1, player2 (Player objects) Return value(s): -	Displays the following for both players: 1) Their remaining resources 2) Their VPs

COPYRIGHT
PROTECTED



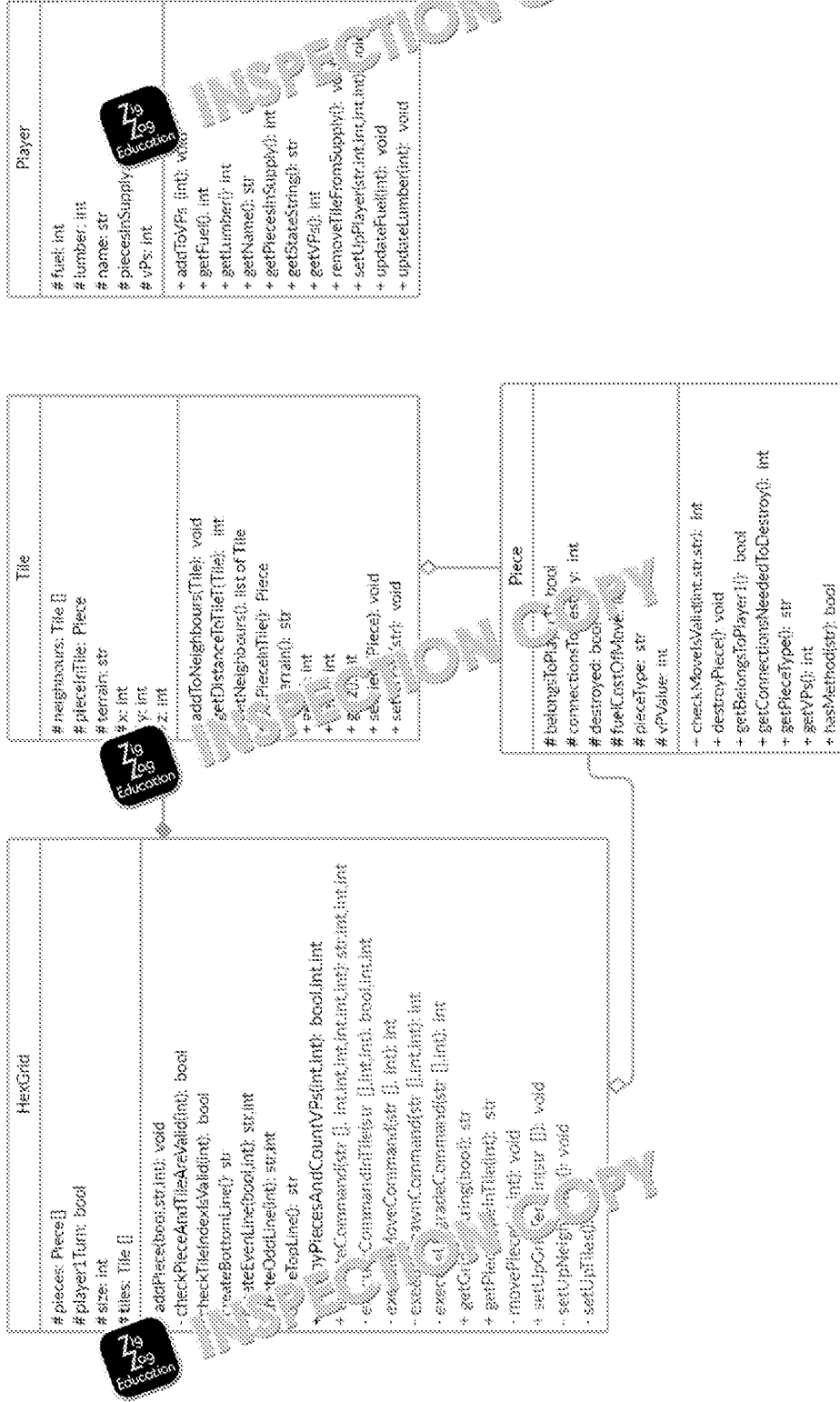
INSPECTION COPY

Name	Data	Description
HexBaron	Parameters: - Return value(s): -	Calls displayMainMenu and processes the result to either quit, load a game by calling loadGame or play the default game by calling setUpDefaultGame and then calling playGame to play the game.
playGame	Parameters: player1 (Player object) player2 (Player object) grid (HexGrid object) Return value(s): -	This alternates between Player 1 and Player 2 and always ensures that Player 2 has an equal number of turns even if Player 1 has just killed their Baron. For each player's turn, the board is printed out and then three commands are read in and validated (by calling checkCommandIsValid) and then executed (by calling executeCommand on the HexGrid object for the game – stored in the variable grid). After each command, the updateLumpier, updateFuel and removeTileFromSupply (if necessary) methods are called on the Player object for that player's turn. Once the commands have been executed, places are destroyed, VPs are assigned and a check is made to see whether the game is over. Before the game moves on to the next player's turn, the status of both players (resources and VPs) is displayed. If the game is over and Player 2 has played, then displayEndMessages is called to display the final scores

COPYRIGHT
PROTECTED



INSPECTION COPY

COPYRIGHT
PROTECTED

INSPECTION COPY

BaronPiece (inherits from Piece)

Name	Data	Description
BaronPiece (constructor)	Parameters: player1 (Boolean) Return value(s): -	Calls the super constructor (Piece) and passes player1 as an argument. Initialises the following protected attributes: • pieceType to "B" • vpValue to 10
checkMoveIsValid (public)	Parameters: distanceBetweenTiles (int) startTerrain (string), endTerrain (string) Return value(s): fuelCostOfMove (int)	If the value of the parameter distanceBetweenTiles is 1 then it returns the value of the protected attribute fuelCostOfMove otherwise it returns -1.

HexGrid

Name	Data	Description
HexGrid (constructor)	Parameters: n (int) Return value(s): -	Initialises the following protected attributes: • size from parameter n • player1Turn to true It calls the private methods setUpTiles and setUpNeighbours.
addPiece (public)	Parameters: player1 (Boolean)	Calls the appropriate constructor to create a new piece of the



COPYRIGHT
PROTECTED

INSPECTION COPY

Name	Data	Description
checkTileIndexIsValid (private)	Parameters: tileToCheck (int) Return value(s): Boolean	Returns true if the parameter tileToCheck is a valid index of the protected attribute tiles.
createBottomLine (private)	Parameters: - Return value(s): Line (string)	Returns a string containing the bottom line of the grid (as displayed at the start of each turn when playing the game).
createEvenLine (private)	Parameters: - Return value(s): Line (string)	Returns a string containing the correct string of an even line of the grid (as displayed at the start of each turn when playing the game).
createOddLine (private)	Parameters: - Return value(s): Line (string)	Returns a string containing the correct string of an odd line of the grid (as displayed at the start of each turn when playing the game).
createTopLine (private)	Parameters: - Return value(s): Line (string)	Returns a string containing the top line of the grid (as displayed at the start of each turn when playing the game).
destroyPiecesAndCountVPS (public)	Parameters: - Return value(s): baronDestroyed (Boolean), player1VPS (int), player2VPS (int)	Loops through every tile in the grid and checks for any pieces that need to be destroyed by counting the number of connections for each piece and comparing it to the number of connections needed to destroy the piece in question. For each piece that is destroyed, track the VPs for the relevant player and also whether their Baron was destroyed or not. Finally, it then removes any pieces from the grid that were destroyed.
executeCommand (public)	Parameters: items (list of strings),	Depending on the first element of items, it either calls executeMoveCommand, executeSpawnCommand,

COPYRIGHT
PROTECTED



INSPECTION COPY

Name	Data	Description
executeCommandInTile (private)	Parameters: items (list of strings) Return value(s): status (Boolean), fuel (int), lumber (int)	Checks whether there is a piece belonging to the player in the tile specified as the second string in items, if not then false, 0 and 0 are returned. It then uses hasMethod to determine whether the piece in the tile has a saw or dig command as specified by the first string in the items list and calls that method. If it was a dig and more than 5 fuel was returned then it sets the terrain of the tile to a field using the setTerrain method. The method then returns true and the fuel or lumber gained from digging or sawing or it returns false and 0,0 if the command could not be executed.
executeMoveCommand (private)	Parameters: items (list of strings), fuelAvailable (int) Return value(s): fuelCost (int)	Checks whether there is a piece belonging to the player in the tile specified as the second string in items and that the tile specified in the third string of items is empty, if not then -1 is returned straight away. Otherwise, it then checks that the move is allowed by calling the checkMoveIsValid method on the piece in the first tile and if there is enough fuel available and the move was valid then it calls movePiece to execute the move and returns fuelCost (normally 1 or 2), otherwise it returns -1.
executeSpawnCommand (private)	Parameters: items (list of strings), lumberAvailable (int)	Checks to see whether the player has at least 1 pieceInSupply and 3 lumber and that the tile specified as the second string in the items list is empty, otherwise it returns

COPYRIGHT
PROTECTED



INSPECTION COPY

Name	Data	Description
getGridAsString (public)	Parameters: p1Turn (Boolean) Return value(s): gridAsString (string)	Uses the private attribute listPositionOfTile to loop through the tiles in the grid calling the private methods createTopLine, createOddLine, createEvenLine and createBottomLine needed to form a string containing the entire grid.
getPieceTypeInTile (public)	Parameters: id (int) Return value(s): pieceType (string)	If there is no piece in the tile specified by id then this returns "" (string with a single space), otherwise it returns the result of the getPieceType method which is called on the piece in the tile.
movePiece (private)	Parameters: newIndex (int), oldIndex (int) Return value(s): -	Sets the piece at newIndex (in the tiles attribute) to the same as the piece (or None) at oldIndex by calling the setPiece method on the tile, then set the piece at the oldIndex to None, again by calling the setPiece method on the tile.
setUpGridTerrain (public)	Parameters: listOfTerrain (list of strings) Return value(s): -	Loops through the listOfTerrain and calls setTerrain for each tile in the protected attribute tiles which is a list of all the tiles on the grid) in the same order.
setUpNeighbours (private)	Parameters: - Return value(s): -	For each tile on the grid, it loops through all the tiles of the grid and adds any tile that is a distance of 1 away from the tile determined by the getDistanceToTileT method in the Tile class) to the list of neighbours by calling the addToNeighbours method on the tile and passing an argument of the tile that is 1 away.
setUpTiles (private)	Parameters: - Return value(s): -	Loops through and creates all of the tiles needed for the grid according to the protected attribute size and adds them to the protected attribute tiles.

COPYRIGHT
PROTECTED



INSPECTION COPY

LESSPiece (inherits from Piece)

Name	Data	Description
LESSPiece (constructor)	Parameters: player1 (Boolean) Return value(s): -	Calls the super constructor (Piece) and passes player1 as an argument. Initialises the following protected attributes: • pieceType to "L" • vpValue to 3
checkMoveIsValid (public)	Parameters: distanceBetweenTiles (int), startTerrain (string), endTerrain (string) Return value(s): fuelCostOfMove (int)	If the value of the parameter distanceBetweenTiles is 1 and startTerrain is not a forest then if the move begins or ends in a peat bog it returns fuelCostOfMove x2 and if the move doesn't begin or end in a peat bog then it returns fuelCostOfMove otherwise it returns 1.
saw (public)	Parameters: terrain (string) Return value(s): lumber (int)	If terrain is a forest it returns 1, otherwise it returns 0.

PBDSPiece (inherits from Piece)

Name	Data	Description
PBDSPiece (constructor)	Parameters: player1 (Boolean) Return value(s): -	Calls the super constructor (Piece) and passes player1 as an argument. Initialises the following protected attributes: • fuelCostOfMove to 2

COPYRIGHT
PROTECTED



INSPECTION COPY

Piece

Name	Data	Description
Piece (constructor)	Parameters: player (Boolean) Return value(s): -	Initialises the following protected attributes: • belongsToPlayer1 to Player1 • connectionsToDestroy to 2 • destroyed to false • fuelCostOfMove to 1 • pieceType to "S" • vpValue to 1
checkMoveIsValid (public)	Parameters: distanceBetweenTiles (int), startTerrain (string), endTerrain (string) Return value(s): fuelCostOfMove (int)	If the value of the parameter distanceBetweenTiles is 1 then if the move begins or ends in a peat bog it returns fuelCostOfMove x2 and if the move doesn't begin or end in a peat bog then it returns fuelCostOfMove otherwise it returns -1.
destroyPiece (public)	Parameters: - Return value(s): -	Sets the value of the protected attribute destroyed to true.
getBelongsToPlayer1 (public)	Parameters: - Return value(s): belongsToPlayer1 (Boolean)	Returns the value of the protected attribute belongsToPlayer1.
getConnectionsNeededToDestroy (public)	Parameters: - Return value(s): -	Returns the value of the protected attribute connectionsToDestroy.

COPYRIGHT
PROTECTED



INSPECTION COPY

Player

Name	Data	Description
Player (constructor)	Parameters: - Return value(s): -	This is an empty constructor that simply creates an empty object, later on it will be populated using setUpPlayer().
addToVPs	Parameters: n (int) Return value(s): -	Increments the attribute vPs.
getFuel (public)	Parameters: - Return value(s): fuel (int)	Returns the value of the attribute fuel.
getLumber (public)	Parameters: - Return value(s): lumber (int)	Returns the value of the attribute lumber.
getName (public)	Parameters: - Return value(s): name (string)	Returns the value of the attribute name.
getPiecesInSupply (public)	Parameters: - Return value(s): piecesInSupply (int)	Returns the value of the attribute piecesInSupply.
getStateString (public)	Parameters: - Return value(s): stateString (string)	Returns a string containing the vPs, the piecesInSupply, the lumber and the fuel.
getVPs (public)	Parameters: - Return value(s): vPs (int)	Returns the value of the attribute vPs.
removeTileFromSupply (public)	Parameters: - Return value(s): -	Decrements the piecesInSupply attribute by 1.
setUpPlayer (public)	Parameters: N (string), V (int), F (int), L (int), T (int)	Initialises the following protected attributes: - name from parameter N - vPs from parameter V

COPYRIGHT
PROTECTED



INSPECTION COPY

Tile

Name	Data	Description
Tile (constructor)	Parameters: xCoord (int), yCoord (int), zCoord (int) Return value(s): -	Initialises the following protected attributes: • x from parameter xCoord • y from parameter yCoord • z from parameter zCoord • terrain to "" (a string containing a single space) • pieceInTile to None
addToNeighbours (public)	Parameters: N (Tile object) Return value(s): -	Adds the tile N to the end of the list of protected attribute neighbours.
getDistanceToTile (public)	Parameters: t (Tile object) Return value(s): distance (int)	Returns the maximum of the absolute difference in x, y and z between the current tile and t.
getNeighbours (public)	Parameters: - Return value(s): neighbours (list of Tiles)	Returns the value of the protected attribute neighbours.
getPieceInTile (public)	Parameters: - Return value(s): pieceInTile (Piece object or None)	Returns the value of the protected attribute pieceInTile.
getTerrain (public)	Parameters: - Return value(s): terrain (string)	Returns the value of the protected attribute terrain.
getX (public)	Parameters: - Return value(s): x (int)	Returns the value of the protected attribute x.
getY (public)	Parameters: -	Returns the value of the protected attribute y.

COPYRIGHT
PROTECTED



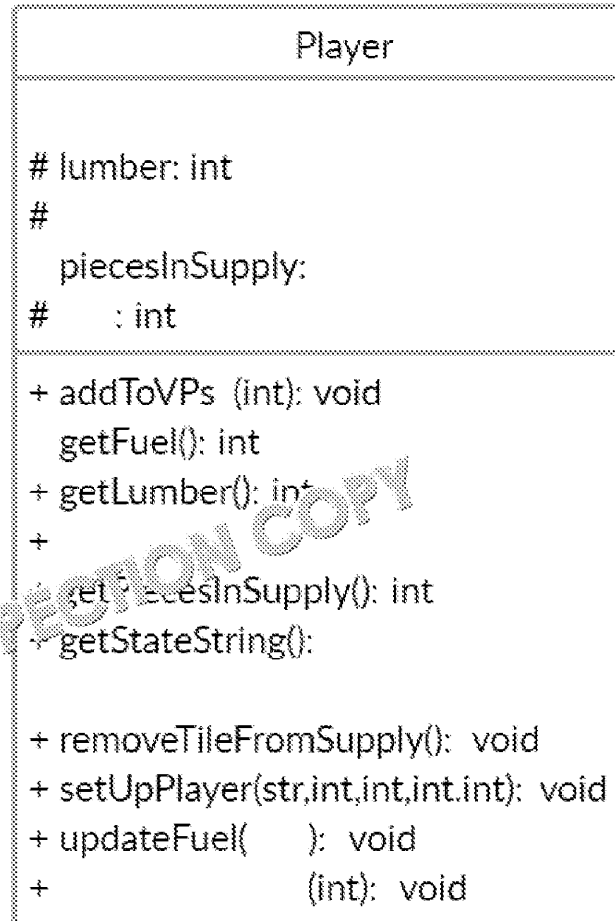
INSPECTION COPY

HEX BARON – Class Diagram Tasks

Task 1

A partially-complete UML class diagram for the `Player` class is shown below.

Fill in the missing details.



INSPECTION COPY

COPYRIGHT
PROTECTED

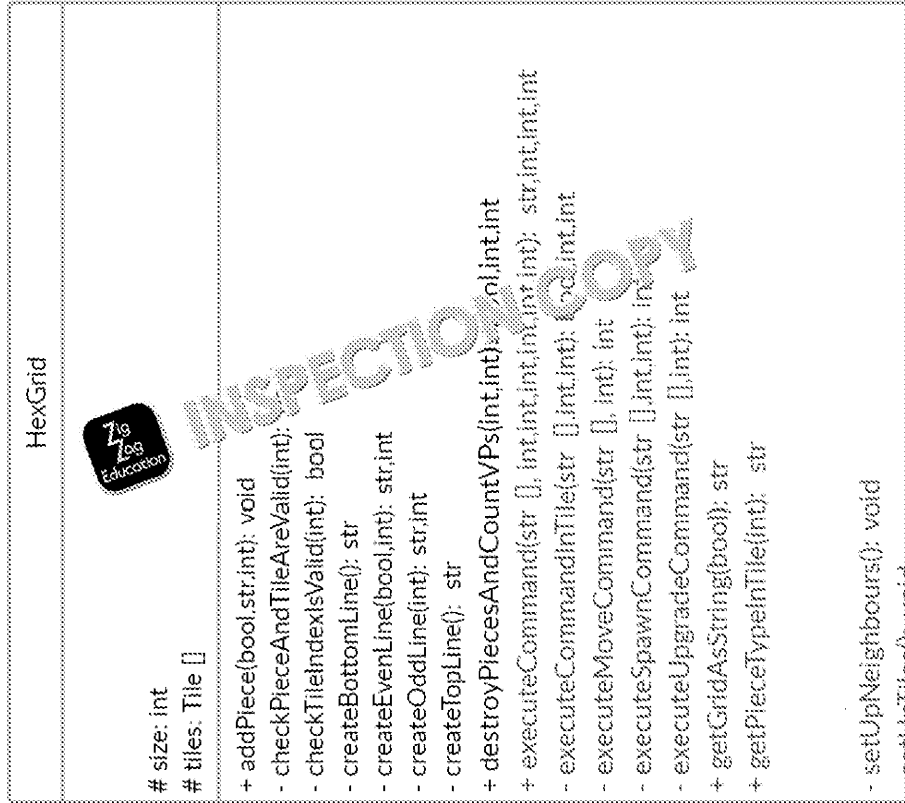


Task 2

(5 marks)

A partially-complete UML class diagram is shown.

Fill in the missing details, including any relationships between the classes shown.



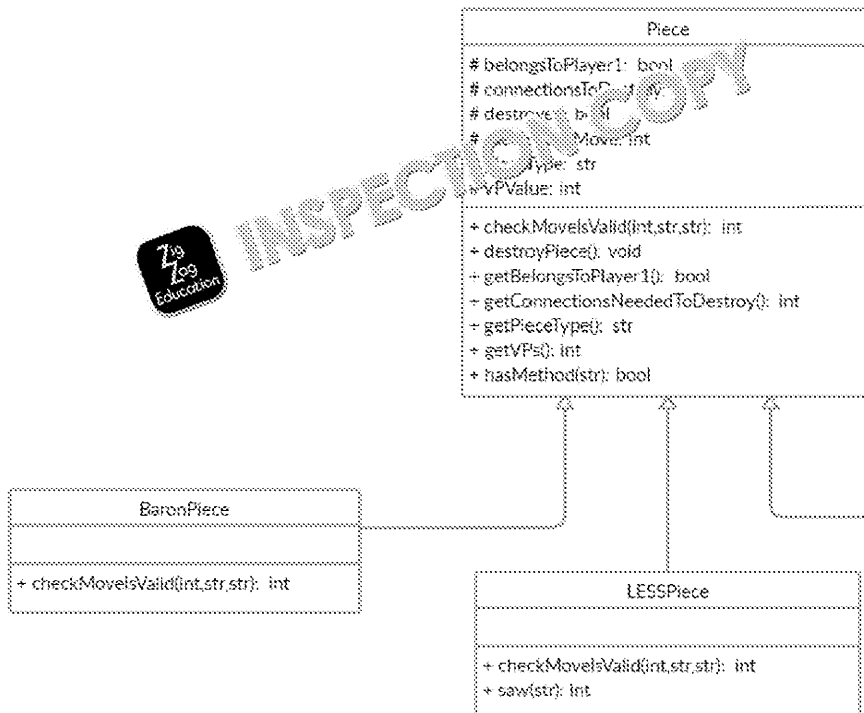
COPYRIGHT
PROTECTED



INSPECTION COPY

Task 3

There are two pieces of information missing from the UML class diagram below.



Explain what are they and why they are normally omitted?

INSPECTION COPY

COPYRIGHT
PROTECTED

HEX BARON – Theory Questions

These questions refer to the preliminary material and require you to look at the code but do not require any additional programming.

1 This question refers to the `playGame` subroutine in the main program. The subroutine does not currently print out the actual player names when at the start of each turn; it just prints out 'Player One' and 'Player Two'.

a) How could this be changed so that it would print out the names using methods to get the selected attribute name from the `Player` class?



b) This is an example of using an accessor method. Why are accessors used in object-oriented programming?

2 This question refers to the `playGame` subroutine in the main program. In the subroutine, the variable `player1Turn` is set to true to indicate Player One's turn. How does the game process Player Two's turn?

3 This question refers to the `hasMethod` method within the `Piece` class. Explain how `HasMethod` is used in the `executeCommandInTile` method in the `HexGrid` class.



Explain how `HasMethod` is used in the `executeCommandInTile` method in the `HexGrid` class.

4 This question refers to the `getDistanceToTileT` method in the `Tile` class. `getDistanceToTileT` is used to calculate the distance from the current tile to the target tile.

a) Explain how this calculation is performed.



INSPECTION COPY

COPYRIGHT
PROTECTED



- 4 b) Give a worked example of this calculation if the current tile has coordinates (4,4,0) and the destination tile has coordinates (4,-4,0). All coordinates are

.....

.....

.....

.....

- 5 The BaronPiece, LESSPiece and PBDSPiece classes all inherit from the Piece class.

- a) Explain what is meant by 'inheritance'.

.....

.....

- b) Why isn't there a 'SerfPiece' class?

.....

.....

- 6 This question refers to the BaronPiece class and the Piece class. The BaronPiece class overrides the public method checkMoveIsLegal.

- a) Explain what is meant by 'overriding'.

.....

.....

.....

- b) Explain why the method was overridden in this case.

.....

.....

.....

- 7 Big-O notation is used to measure the time complexity of algorithms.

- a) Examine the getNeighbours method in the HexGrid class.

.....

- b) Which other method is used as a measure of the complexity of algorithms?

.....

**COPYRIGHT
PROTECTED**

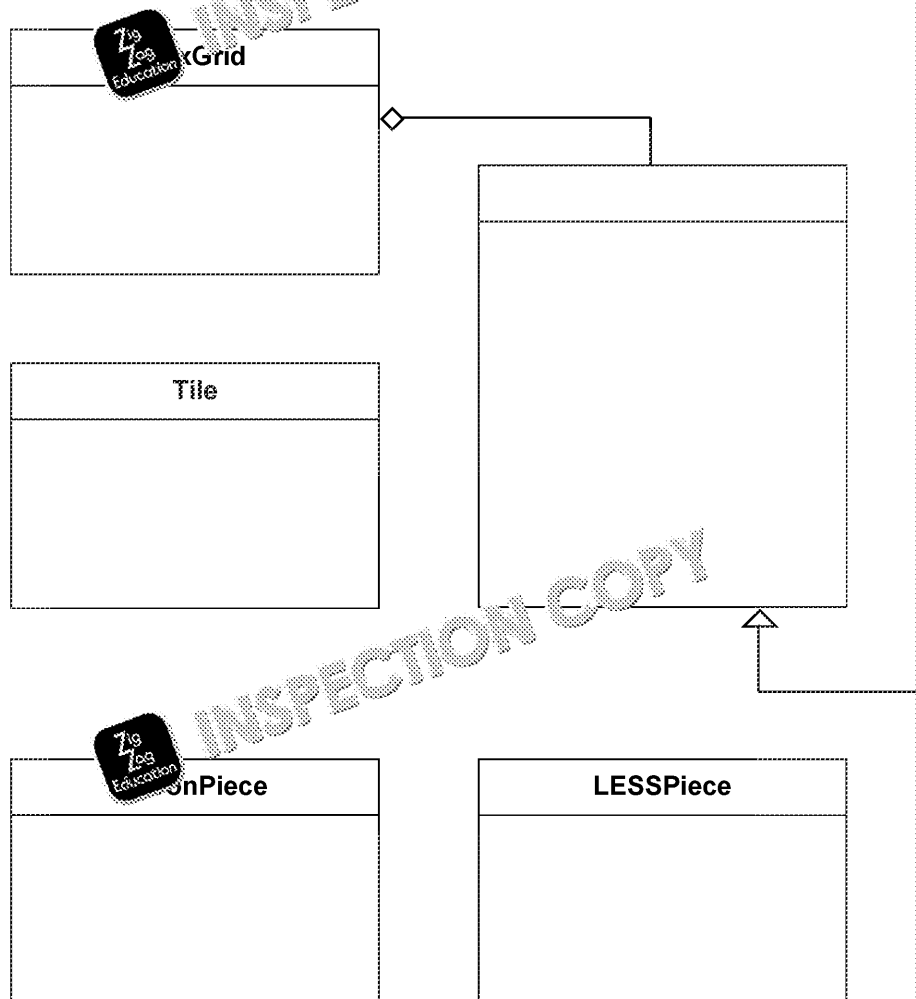


- 8 The skeleton program uses a Comma Separated Values (CSV) file as saved game. Explain why a CSV file is suitable for cross-platform appl

.....

.....

- 9 Class diagrams can be represented using Unified Modeling Language. Add the missing class names and relationships to the diagram below:



- 10 This question refers to the executeUpgradeCommand method of the HexBaron class.
- a) What does a return value of -1 from this method mean?

.....

.....

- b) What does a return value of 5 from this method mean?

.....

.....

**COPYRIGHT
PROTECTED**



- 11 This question refers to the `dig` method of the `PBDSPiece` class and the `executeCommandInTile` method of the `HexGrid` class.

When a 'dig' command is issued by a player, it is possible that the tile is changed from a peat bog to a field and that they will receive 5 fuel instead of the

Explain how this eventuality is processed in the code with specific references to the methods used, function calls made and return values



- 12 This question refers to the `destroyPiecesAndCountVPs` method of the `Piece` class.

For a piece to be destroyed in the game, it needs to have two connected neighbours (i.e. pieces in two of the tiles that share a side with it).

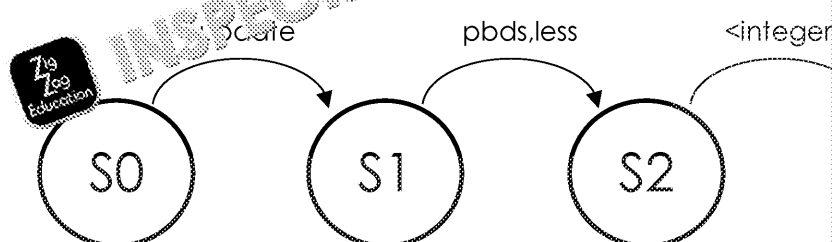
- a) Explain how the `destroyPiecesAndCountVPs` method uses the protected attribute `connections` to destroy the `Piece`



- b) Why is polymorphism not possible for private attributes?

- 13 This question refers to the `checkUpgradeCommandFormat` subroutine

Currently the validation performed by this subroutine could be represented by a state machine diagram (note that the 0 or higher condition is not currently checked in the code answer).



**COPYRIGHT
PROTECTED**



13 Where <integer> is any valid (0 or higher) integer.

a) Write down the regular expression that is the equivalent of this FSM

.....

b) Improve your regular expression so that it only accepts integers from

.....

c) What is the definition of a regular language?

.....



14 This question refers to the `getGridAsString` method of the `HexG`

a) State the identifier for a local variable used in this method.

.....

b) What are the advantages of using local variables?

.....

.....

.....

c) This method uses one private and two protected attributes. What is the difference between a private and a protected attribute?

.....



.....

.....

15 This question refers to the `executeSpawnCommand` method of the

Explain how the method works, including reference to how it meets the requirements for spawning a new Serf (it must be on an empty square adjacent to that of the existing Serf)

.....

.....

.....

.....



.....

.....

.....

**COPYRIGHT
PROTECTED**



HEX BARON – Theory Questions

These questions refer to the preliminary material and require you to look at the code but do not require any additional programming.

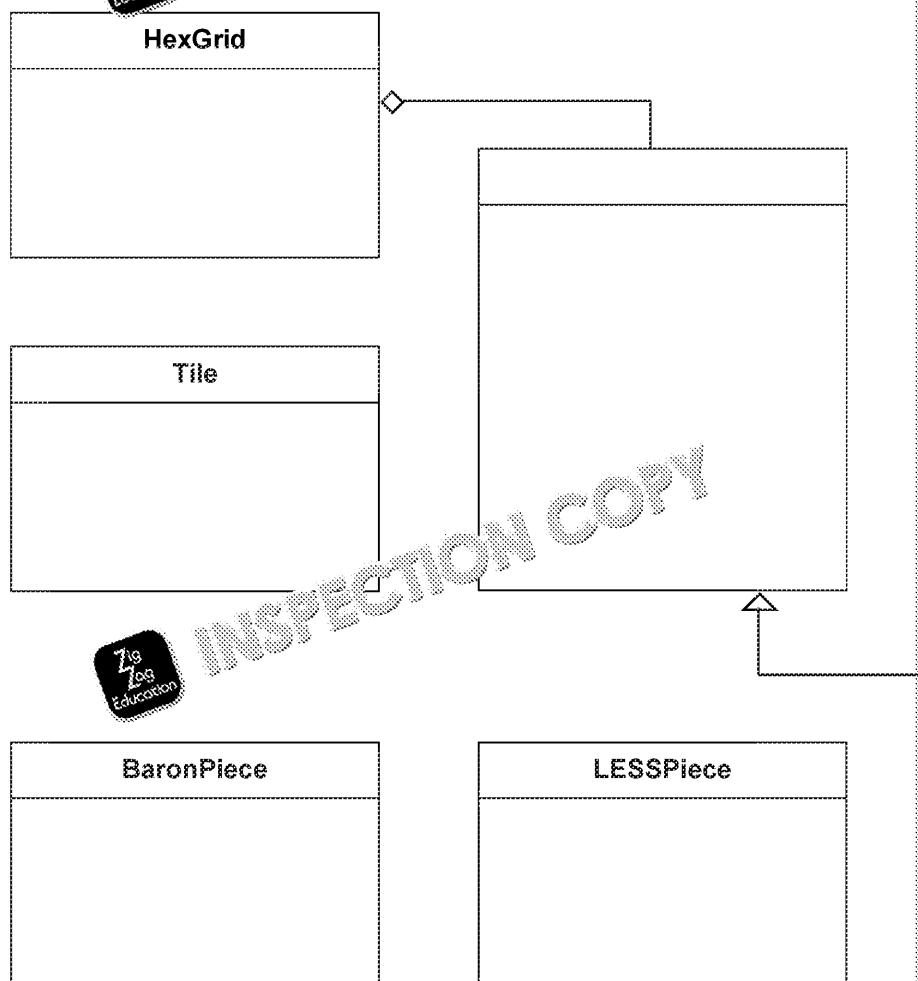
- 1 This question refers to the `playGame` subroutine in the main program.
The subroutine does not currently print out the actual player names who are at the start of each turn. It just prints out 'Player One' and 'Player Two'.
 - a) How would this be corrected so that it would print out the names using the protected attribute name from the `Player` class?
 - b) This is an example of using an accessor method. Why are accessors used in object-oriented programming?
- 2 This question refers to the `playGame` subroutine in the main program.
In the subroutine, the variable `player1Turn` is set to true to indicate Player One's turn. How does the game process Player Two's turn?
- 3 This question refers to the `hasMethod` method within the `Piece` class and the `executeCommandInTile` method in the `HexCell` class.
Explain how `hasMethod` is used in the `executeCommandInTile` method.
- 4 This question refers to the `getDistanceToTileT` method in the `Tile` class. The `getDistanceToTileT` is used to calculate the distance from the current tile to the destination tile.
 - a) Explain how this calculation is performed.
 - b) Give a worked example of this calculation if the current tile has coordinates (0,0,0) and the destination tile has coordinates (4,-4,0). All coordinates are in (x,y,z) form.
- 5 The `BaronPiece`, `LESSPiece` and `PBDSPiece` classes all inherit from the `Piece` class.
 - a) Explain what is meant by 'inheritance'.
 - b) Why isn't there a 'SerfPiece' class?
- 6 This question refers to the `BaronPiece` class and the `Piece` class.
The `BaronPiece` class overrides the public method `checkMoveIsValid`.
 - a) Explain what is meant by 'overriding'.
 - b) Explain why the method was overridden in this case.

INSPECTION COPY

COPYRIGHT
PROTECTED



- 7 Big-O notation is used to measure the time complexity of algorithms.
- Examine the `setUpNeighbours` method in the `HexGrid` class
 - Which other method is used as a measure of the complexity of algo
- 8 The skeleton program uses a Comma Separated Values (CSV) file as a saved game. Explain why a CSV file is suitable for a cross-platform applica
- 9 Class diagrams can be represented using Unified Modeling Language. Add the missing names and relationships to the diagram below:



- 10 This question refers to the `executeUpgradeCommand` method of the
- What does a return value of -1 from this method mean?
 - What does a return value of 5 from this method mean?
- 11 This question refers to the `log` method of the `PBDSPiece` class and the `executeCommandInTile` method of the `HexGrid` class.
- When a logging command is issued by a player, it is possible that the tile will move from a peat bog to a field and that they will receive 5 fuel instead of the
- Explain how this eventuality is processed in the code with specific references to methods used, function calls made and return values.

**COPYRIGHT
PROTECTED**

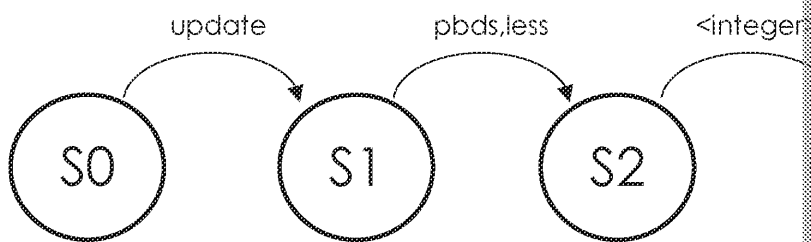


- 12 This question refers to the `destroyPiecesAndCountVPs` method of the `Piece` class.

For a piece to be destroyed in the game, it needs to have two connecting neighbours (i.e. pieces in two of the tiles that share a side with it).

- Explain how the `destroyPiecesAndCountVPs` method uses the protected attribute `connectionsToNeighbours` of the `Piece` class.
- Why is polymorphism not possible for private attributes?

- 13 This question refers to the `checkUpgradeCommandFormat` subroutine. Currently, the validation performed by this subroutine could be represented by the following FSM (note that the 0 or higher condition is not currently checked in the code answer).



Where `<integer>` is any valid (0 or higher) integer.

- Write down the regular expression that is equivalent of this FSM.
- Improve your regular expression so that it only accepts integers from 0 or higher.
- What is the definition of a regular language?

- 14 This question refers to the `getGridAsString` method of the `HexGrid` class.

- State the identifier for a local variable used in this method.
- What are the advantages of using local variables?
- This method uses one private and two protected attributes. What is the difference between a private and a protected attribute?

- 15 This question refers to the `executeSpawnCommand` method of the `HexGrid` class.

Explain how the method works, including reference to how it meets the requirement of spawning a new Serf (it must be on an empty square adjacent to that piece).

**COPYRIGHT
PROTECTED**



HEX BARON – Programming Tasks

Task 1

This question refers to the `playGame` subroutine in the main program.

The commands input by the player are already converted to lower case, which is case insensitive. However, sometimes a player may additionally or accidentally enter a space at the start or end of the command – currently this often causes an invalid command.

- Create a new subroutine called `strip` to remove any leading and trailing spaces from the command.
- Change the existing line of code that converts the command to lower case to use the new `strip` subroutine to also remove the unnecessary spaces.

Test that the change you have made works:

- Choose menu option 1 to run the default game and then enter the following commands (they are in quotes as the spaces are important to type):
 - "move 8 16 "
 - "move 8 16"
 - " upgrade pbds 16 "
- Show a screen copy of entering the commands and the response from the program. Highlight the Player One current state line.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `playGame` and the new `strip` subroutine.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 2

This question refers to the `setUpDefaultGame` subroutine in the main

At the moment, the game sets the default names of the players to 'Player One' and 'Player Two' in the `setUpDefaultGame` subroutine. Change this subroutine to prompt for names and then use the values entered to construct the players.

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter "Tom" and "Jerry" as the name for Player Two.
- Show a screen copy of entering the commands and the responses including the prompt for the first player to enter their commands.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `setUpDefaultGame`.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 3

This question refers to the `playGame` and `setUpDefaultGame` subroutines of the `Player` class and to the `destroyPiecesAndCountVPs` method of the `HexGrid` class.

Normally when a piece is destroyed it depletes your supply chain and moves you closer to losing the game as the number of pieces that you can make is limited. This means that if a piece is destroyed then you can't make a new piece in your supply chain.

- Add a method to the `Player` class that will allow the pieces in the supply chain to be replenished.
- Modify the `destroyPiecesAndCountVPs` method to take player and grid as parameters and pass them in from `playGame` in both places that it is called.
- Further modify the `destroyPiecesAndCountVPs` method so that it calls the `addPiecesInSupply` method that you created for the appropriate player to add an additional piece in their supply every time one of their pieces is destroyed.
- Modify the `setUpDefaultGame` subroutine so that the existing pieces in the grid are replaced with the following six:

```
grid.addPiece(true, "Baron", 0);
grid.addPiece(true, "Serf", 7);
grid.addPiece(true, "Serf", 10);
grid.addPiece(false, "Baron", 31);
grid.addPiece(false, "Serf", 15);
grid.addPiece(true, "Serf", 23);
```

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter the following commands:
 - move 13 10
 - move 10 14
 - move 14 19
- Show a screen copy of entering the commands and the responses from the game, including the prompt for the second player to enter their commands.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `playGame` and the `destroyPiecesAndCountVPs` method of the `HexGrid` class.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 4

This question refers to the `setUpDefaultGame` subroutine in the main `executeSpawnCommand` method in the `HexGrid` class.

Change the rules of the game so that each player can have a maximum of 4 pieces. There should be an additional output message generated saying that they have exceeded max pieces.

- Modify the private method `executeSpawnCommand` of the `HexGrid` class.
- Modify the `setUpDefaultGame` subroutine so that the existing pieces are replaced with the following eight:

```
grid.addPiece(true, "Baron", 0);
grid.addPiece(true, "Serf", 8);
grid.addPiece(true, "Serf", 24);
grid.addPiece(true, "Serf", 5);
grid.addPiece(true, "Serf", 2);
grid.addPiece(true, "Serf", 3);
grid.addPiece(false, "Baron", 31);
grid.addPiece(false, "Serf", 23);
```

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter the following commands:
 - spawn 4
 - move 4 0
- Show a screen copy of entering the commands and the responses. Point at which it prints out the Player One current state line.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended `executeSpawnCommand` method in the `HexGrid` class.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 5

This question refers to the `playGame` subroutine in the main program, and `getGridAsIndices` in the `HexGrid` class.

Add an additional command to the game which will print out the hex grid as a hex; these should be in the top half of each hex and the board should be the display. The new command, "hexes" should not count as one of the three; the results should be displayed immediately upon entry, i.e. the grid should be numbers displayed as soon as Enter is pressed. The player should be able to enter their command and then have three normal commands.

- Create a new method for the `HexGrid` class called `getGridAsString` which returns a string containing the grid with all of the index numbers in the correct order. This should be on the current `getGridAsString` method. You may wish to duplicate the `createOddLine` and `createEvenLine` methods.
- Modify the `playGame` subroutine to implement a call to the new method when the user enters the command "hexes" and then request the normal command.

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter the following commands:
 - move 8 16
 - hexes
 - move 16 24
 - hexes
 - hexes
- Show a screen copy of entering the commands and the responses including the prompt for the second player to enter their commands. The hex grid with the numbers and the printout of the board after Player 1 move.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `playGame` and the `getGridAsIndices` method of the `HexGrid` class and any new versions of `createOddLine` and `createEvenLine` added to the class.
- SCREEN CAPTURE(S) showing the completed test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 6

This question refers to the `playGame` and `setUpDefaultGame` subroutines.

Change the move command so that if you move the same piece three times in cost of 1 fuel (in total). For example, if you moved a 6erf to a field (from 0 to 4) and then to a field, it would only cost 4 fuel instead of 5. If you moved a B to a field, it would only cost 2 fuel instead of 3. They should be able to do the triple move even if they are out of fuel on 0.

- Modify the `playGame` subroutine to check whether three valid moves exist and if so, refund them one fuel.
- Modify the `setUpDefaultGame` subroutine so that Player One's initial state is:

```
player1.setUpPlayer("Player One", 0, 2, 10, 5)
```

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter the following commands:
 - move 0 4
 - move 4 9
 - move 9 17
- Show a screen copy of entering the commands and the responses and point at which it prints out the Player One current state line.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `playGame`.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 7

This question refers to the `playGame` subroutine in the main program and `executeCommandInTile` method and a new method, `makeField`.

Change the saw and dig commands so that if you saw or dig the same location the same turn then you get 5 of that resource and the terrain reverts to a field. A chance of getting 5 fuel when you dig so that you always get only 1.

- Modify the `executeCommandInTile` to remove the possibility of creating a field.
- Modify the `dig` method of the `PBDSPiece` to remove the possibility of generating 5 fuel.
- Add a method to the `HexGrid` class called `makeField` which takes the tile which is to be made into a field.
- Modify the `playGame` subroutine to check whether three dig or saw commands are executed in a single turn by either player and give them the extra resource by calling the newly added `makeField` method in the `HexGrid` class.

Test that the changes you have made work:

- a) Choose menu option 1 to run the default game and then enter the following commands:

Player One	• move 8 16 • move 16 24 • upgrade phase 1
Player Two	• move 23 27 • move 27 30 • upgrade less 30
Player One	• dig 24 • dig 24 • dig 24
Player Two	• saw 30 • saw 30 • saw 30

- b) Show a screen copy of the final board after all of the commands have been entered. Also show the Player One and Player Two current states that are shown immediately after the commands are entered.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `playGame`, `executeCommandInTile` and `makeField` methods of the `HexGrid` class and the modified `dig` method of the `PBDSPiece` class.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 8

This question refers to the `checkCommandIsValid` and `setUpDefaultGame` in the main program, the creation of a new subroutine `checkDowngradeCommandFormat`, the `executeCommand` method, `executeDowngradeCommand` as well as the `executeCommand` method of the `HexGrid` class.

Introduce a new downgrade command that a PDBS or LESS can be downgraded by 1 lumber. The syntax for the command is "downgrade NUM" where NUM is the number of lumber to be downgraded.

- Modify the `checkCommandIsValid` subroutine to allow for and handle the downgrade command.
- Create a new subroutine `checkDowngradeCommandFormat` within the `checkCommandIsValid` subroutine.
- Modify the `executeCommand` method of `HexGrid` to call a new `executeDowngradeCommand` method.
- Create a new private method in `HexGrid` called `executeDowngradeCommand`.
- Modify the `setUpDefaultGame` subroutine so that `player1` starts with 16 lumber.

```
grid.addPiece(true, "Serf", 16);
```

Test that the changes you have made work by:

- Choose menu option 1 to run the default game and then enter the command: `downgrade 24`
The response should be:

```
downgrade pdds 24
```

 - `downgrade 24`
- Show a screen copy of entering the commands and the responses at the end of the game point after it prints out the updated board.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `checkCommandIsValid`, the new subroutine `checkDowngradeCommandFormat`, the `executeCommand` method of the `HexGrid` class and the new code for the `executeDowngradeCommand` method of the `HexGrid` class.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 9

This question refers to the `setUpDefaultGame` subroutine in the main `BaronPiece` class.

Change the rules for the Baron so that it needs three connections to kill it.

- Modify the `BaronPiece` class to make it so that it can only be destroyed when it has three connections.
- Modify the `setUpDefaultGame` subroutine so that the four start the game with seven:

```
grid.addPiece(true, "Baron", 0);
grid.addPiece(true, "Serf", 15);
grid.addPiece(true, "Serf", 27);
grid.addPiece(true, "Serf", 25);
grid.addPiece(false, "Baron", 19);
grid.addPiece(false, "Serf", 8);
grid.addPiece(false, "Serf", 2);
```

Test that the changes you have made work:

- a) Choose menu option 1 to run the default game and then enter the following commands:

Player One	• move 25 21 • move 21 18 • move 19 25
Player Two	• move 5 1 • move 5 1 • move 1 4

- b) Show a screen copy of entering the commands for the entire game.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended `BaronPiece` class.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 10

This question refers to the `checkCommandIsValid` subroutine in the `HexGrid` class, the `executeCommand` method and a new `executeSalvageCommand` method.

Develop the salvage command. Any of your own Serf, 7 BDS or LESS piece. Salvaging a piece will add 1 to your supply chain giving you 5 lumber and re-board. The format of the command is `salvage NUM`, where NUM is the location of the piece to salvage.

- Modify the `checkCommandIsValid` subroutine to allow for and execute the `salvage` command.
- Modify the `executeCommand` method of `HexGrid` to call a new `executeSalvageCommand` method.
- Create a new private method in `HexGrid` called `executeSalvageCommand`.

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter the following commands:
 - `salvage 31`
 - `salvage 4`
 - `salvage 8`
- Show a screen copy of entering the commands and the response from the program. Point at which it prints out the updated board.

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended subroutine `checkCommandIsValid`, the `executeCommand` and `executeSalvageCommand` methods.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 11

This question refers to the `playGame` subroutine in the main program and to the

Introduce three chances for each player so that if they enter an invalid move or command executed for some reason, they can correct it, but only three times per game. Allow them to re-enter the invalid move and deduct 1 from their chances. The remainder of each time a player uses up a chance and at the start of each player's turn.

Also, as the commands and replacements could now be confusing, make sure that for a command or a replacement command they are told which command number

- Modify the `Player` class to create an attribute called `chances` for the
- Modify the `playGame` subroutine so that it allows the player to enter a command that is invalid or could not be executed for some reason.
- Create three new methods for the `Player` class to control access to the `deductChance`, `getChances` and `resetChances`.
- By way of example and clarity, the testing output for Player One's first

```
Enter command 1: move
Enter command 2: move 8 8
Enter command 3: upgrade less 28
Command number 1 is an invalid command
You have 2 chances remaining.
Enter new command 1: move 8 16
Command 1: Command executed
Command 2: That move can't be done
Enter new command 2: move 16 20
```

Test that the changes you have made work by:

- Choose menu option 1 to run the default game and then enter the following commands between each turn:

Player One	• move • move 8 8 • upgrade less 28 • move 8 16 • move 16 20 • move 20 28 • Enter (for Player Two's turn)
Player Two	• move 23 27 • move 27 30 • upgrade less 30 • Enter (for Player One's turn)
Player One	• upgrade less 28 • saw 28 • saw 29 • Enter (for Player Two's turn)

- Show a screenshot of entering the commands and all the responses

Evidence that you need to provide:

- Your PROGRAM SOURCE CODE for the amended `playGame` subroutine.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 12

This question refers to the `checkCommandIsValid` and `setUpDefault` new `checkSpawnCommandFormat` subroutine in the main program, to the `Piece` class and to the `executeSpawnCommand`, `destroyPiece`, `getPieceTypeInTile` and `executeMoveCommand` methods in the

Develop a Bomb piece. The bomb can only be spawned by the Baron but it costs 10 lumber to spawn and 2 lumber for each space that it moves regardless of whether it is moved by the Baron or the player.

When the bomb explodes, it will destroy all pieces in the hexes adjacent to the bomb. It will also destroy two pieces are adjacent to it (as per normal attacks ability).

Once the bomb has moved five spaces it is immobilised and becomes 'primed' (even in the middle of a move) one of the hexes surrounding the bomb will not explode if it is primed next to an existing piece (but of course existing pieces), only if another piece enters a hex next to it, kind of like a

The command is a modification of the spawn command, so now spawn can take an argument of bomb, e.g. spawn bomb NUM would spawn a bomb at NUM in Baron, and spawn NUM would operate as normal. The bomb has a VP value of 5, so when it explodes, the opposing player receives 5 VPs.

- Modify the `checkCommandIsValid` subroutine to make the new one correctly.
- Create a new subroutine called `checkValidCommandFormat` to make `checkCommandIsValid` format, to make sure that the new spawn bomb is correctly formatted.
- Modify the private method `executeSpawnCommand` so that it will check for the number of pieces or report an error if they don't have enough pieces.
- Create a new `BombPiece` class for the bomb.
- Modify the method `destroyPiecesAndCountVPs` so that if a bomb explodes and destroys the surrounding pieces too.
- Modify the method `getPieceTypeInTile` for `HexGrid` so that it can find a bomb when the board is displayed.
- Modify the private method `executeMoveCommand` so that it checks for a primed bomb next to a primed bomb, and explodes the bomb if it did.
- Modify the `Piece` class to add an `explode` method so that it can explode the turn in case a bomb was set off during the turn.
- Modify the `setUpDefaultGame` subroutine so that both players are correctly set up.

```
player1.setUpPlayer("Player One", 0, 30, 30,  
player2.setUpPlayer("Player Two", 1, 30, 30,
```

TASK CONTINUES ON THE NEXT PAGE

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Test that the changes you have made work:

- a) Choose menu option 1 to run the default game and then enter the following commands:

Player One	• spawn bomb 4 • move 4 9 • move 9 13
Player Two	• move 23 19 • move 10 11 • move 11 14
Player One	• move 13 18 • move 18 22 • move 22 27
Player Two	• move 31 23 • move 14 18 • move 18 22

- b) Show a screen copy of entering the commands and the responses at the point at which it prints out that Player One is the winner.

- c) Choose menu option 1 to run the default game and then enter the following commands:

Player One	• spawn bomb 4 • move 4 9 • move 9 13
Player Two	• move 23 19 • upgrade to 10 • move 10 11
Player One	• move 13 18 • move 18 22 • move 22 27
Player Two	• saw 19 • saw 19 • saw 19

- d) Show a screen copy of entering the commands and the responses at the point at which it prints out that Player One is the winner.

Evidence that you need to provide:

- a. Your PROGRAM SOURCE CODE for the modified `checkCommand`, `checkSpawnCommandFormat` subroutines, the new `BombPiece` class and the modified methods `executeSpawnCommand`, `destroyPiece`, `AccountVPs`, `getPieceTypeInTile` and of the `TextIO` class.
- b. SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 13

This question refers to the `playGame`, `checkUpgradeCommandForm`, `setUpDefaultGame` subroutines in the main program, to the new `Wizard` class, to the new `executeUpgradeCommand` and new `resetWizards` methods in the

Introduce a new piece, a Wizard which can teleport a distance of three squares, this costs 5 fuel. It can also move one square for a cost of 1 fuel. The piece is upgraded from a Serf using the new command `upgradewiz NUM`, where NUM is containing a Serf. To move the wizard, you simply use the move command. The cost is two or three squares, when it teleports, if it is one square then it moves one square, if it is two or three squares then it teleports, if it is an invalid move. The wizard should be displayed with a picture of a wizard and should have a VP value of 3.

- Create a new `WizardPiece` class.
- Modify the `checkUpgradeCommandFormat` subroutine to allow the new command.
- Modify the method `executeUpgradeCommand` to create the new piece.
- Add a new method to the `HexGrid` class called `resetWizards` to reset the wizards' commands at the end of each turn.
- Modify the `playGame` subroutine to call the new `resetWizards` method (for both sides either side and the new line).
- Modify the `setUpDefaultGame` subroutine so that Player One starts with a wizard.
`player1.setUpPlayer("Player One", 0, 20, 10, 10)`

Test that the changes you made work:

- a) Choose menu option 1 to run the default game and then enter the

Player One	• upgrade wiz 8 • move 8 13 • move 13 18
Player Two	• move 23 19 • move 19 11 • upgrade less 11
Player One	• move 18 8 • move 18 13 • move 13 8

- b) Show a screen copy of entering the command and the responses. Indicate the point at which it prints out the Player One current state line.

Evidence that you need to provide:

- a. You need to provide the PROGRAM SOURCE CODE for the amended subroutines `printHexGrid`, `checkUpgradeCommandFormat`, the `executeUpgradeCommand` and `resetWizards` methods of the `HexGrid` class and the new `Wizard` class.
- b. SCREEN CAPTURE(S) showing the required test.

Task 14

This question refers to the `playGame`, `checkCommandIsValid`, `checkUpgradeCommandFormat` and `setUpDefaultGame` subroutines in the main program, to the new `SCFWPiece` class and to the `executeUpgradeCommand`, `executeCommand` and new `executeSupplyCommand` methods in the `HexGrid` class.

Introduce a new piece, a supply chain factory worker, with the same cost of 5 but the syntax for the upgrade is "upgrade scfw NUM" where NUM is the location of the created, the piece is displayed as F for Player One and f for Player Two. A new "supply NUM", which will take 3 minutes of your moves (so it can only be the first and 5 fuel and will add 1 to the supply chain; NUM is the location of the factory is 3. The factory can be moved once created.

- Create a new `SCFWPiece` class.
- Modify the `checkCommandIsValid` subroutine to validate the supply command.
- Modify the `CheckUpgradeCommandFormat` subroutine so that it can handle the supply command.
- Modify the private method `ExecuteUpgradeCommand` of the `HexGrid` class to allow the `SCFWPiece` to be created.
- Modify the method `ExecuteCommand` of the `HexGrid` class to allow the supply work by calling a new private method, `ExecuteSupplyCommand`.
- Create the new private method `ExecuteSupplyCommand` in the `HexGrid` class to handle the new supply command.
- Modify the `PlayGame` subroutine to only allow the supply command to be entered three times and terminate input for the other two commands if received as the third to call the `AddPieceToSupplyChain` method.
- Create a new `AddPieceToSupplyChain` method in the `Player` class to add the piece to the supply chain.
- Modify the `SetUpDefaultGame` subroutine so that Player One starts with 10 fuel.

```
pl1.setUpPlayer("Player One", 0, 30, 30, 10)
```

Test that the changes you have made work:

- Choose menu option 1 to run the default game and then enter the following commands:

Player One	• move 8 16 • upgrade scfw 16 • supply 16
Player Two	• move 23 19 • move 19 11 • upgrade less 11
Player One	• supply 16

- Show a screen copy of entering the commands and all the responses.

Evidence that you need to provide

- Your SOURCE CODE for the amended subroutines `playGame`, `checkCommandIsValid` and `checkUpgradeCommandFormat`, `executeUpgradeCommand`, `executeCommand` and new `executeSupplyCommand` methods of the `HexGrid` class, the new `SCFWPiece` class and the `addPieceToSupplyChain` method in the `Player` class.
- SCREEN CAPTURE(S) showing the required test.

INSPECTION COPY

COPYRIGHT
PROTECTED



Task 15

This question refers to the `displayMainMenu`, `loadGame` and new code in the main program. The question also relies on using the `getGridAsIndex` including any methods called by that method. Therefore it is best to begin with the codebase that you ended Task 5 with.

Introduce a new command to the main menu that will allow a custom game to be able to enter the grid size as a choice of 6, 8 (the current size) or 10 displayed for them to check – this is a reuse of the code from Task 5 and Task 5 will be the starting point for this task), the location of a Baron for each player, the location of a piece that they wish to add. They should be able to enter the starting amount of VP, lumber and fuel and define the starting resources (including supply chain) for each player. There is no need to perform any validation on the data entry except for the grid size.

The prompts that should be given in this new subroutine (in order) are:

- "Please enter the grid size (6, 8 or 10): "
- "Please enter all the locations of peat bogs separated by commands with a space: "
- "Please enter all the locations of forests separated by commands with a space: "
- "Please enter the location of the Baron for Player 1: "
- "Please enter the location of the Baron for Player 2: "
- "Please enter the starting amount of VPs for Player 1: "
- "Please enter the starting amount of VPs for Player 2: "
- "Please enter the starting amount of fuel for Player 1: "
- "Please enter the starting amount of fuel for Player 2: "
- "Please enter the starting amount of lumber for Player 1: "
- "Please enter the starting amount of lumber for Player 2: "
- "Please enter the starting amount of pieces for Player 1: "
- "Please enter the starting amount of pieces for Player 2: "
- "Enter the piece to add for Player One (S=Serf, L=LESS, P=PBDS) or D=Dragon: "
- "Enter the piece to add for Player Two (S=Serf, L=LESS, P=PBDS) or D=Dragon: "
- "What name would you like to save the file as (please include the extension): "
- "Would you like to play the game that you've created and saved? "

If a piece is chosen by either player, the following prompt should be given:

- "Enter the index of the hex where that piece should be placed: "

The game should automatically save in a file (it should prompt for the name of the file). The players should have the opportunity to play it immediately if they wish with the same name. For reference, see the `game1.txt` file supplied with the pre-release code. The file should be saved as follows (it is all comma-separated values):

Line 1: text for Player One's name, Player One's VPs, Player One's fuel, Player One's supply chain
Line 2: text for Player Two's name, Player Two's VPs, Player Two's fuel, Player Two's supply chain
Line 3: board size
Line 4: list all separated by commas in index order
Line 5+: one line for each piece on the board with the format: 1 or 2 to indicate player, piece name (case sensitive), index (board location)

TASK CONTINUES ON THE NEXT PAGE

INSPECTION COPY

COPYRIGHT
PROTECTED



- Modify the `displayMainMenu` subroutine to add option 3. Create
- Modify the main subroutine so that it calls a new subroutine `createGame` if they would like to load the game and then loads the game. the call `loadGame` so that it accepts the file name as a parameter
- Create a new subroutine called `createGame` which will create the file and return two values, `success` (as a `Boolean`) and `fileName` (the
- Modify the `loadGame` subroutine to correct the bug as described above

Test that the changes you have made work:

- Choose menu option 3 to create a game and then enter the following prompt:
 - 8
 - 24,25,6,7
 - 4,5,26,27
 - 0
 - 31
 - 0
 - 0
 - 12
 - 12
 - 15
 - 15
 - 5
 - 5
 - D
 - S
 - 19
 - D
 - test.csv
 - y
- Show a screen copy of entering the commands and the responses point at which it prints out the Player One current state line.
- Show the contents of the file **test.csv** that was created.

Evidence that you need to provide

- Your **SOURCE CODE** for the amended subroutines `displayMainMenu` and the new subroutine `createGame` in the file. Also include any import statements that you added to the main program.
- SCREEN CAPTURE(S)** showing the required test.

**COPYRIGHT
PROTECTED**



HEX BARON – Extension Tasks

These challenges are presented without solutions, and offer to further explore the skeleton program.

1. Introduce a GOD mode in which resources are infinite for both sides exited (using the command "godmode on" or "godmode off") then the rules and the resources revert to their values before godmode was used.
2. Create a computer controlled player
3. Make the game more user-friendly so that each time any command is given, e.g. 5 lumber deducted for upgrading PBDS in hex 12, connections from hexes 12 & 17 and was destroyed in hex 20.
4. Introduce a new level of upgrade so that pbds and less pieces can be upgraded for a cost of 10 lumber but will now generate 2 resources per turn command.
5. Introduce a new rule so that connections from your own pieces don't count.
6. Introduce an assassin piece that can use a kill command once per turn. If used, the kill command will kill the Baron if it is next to it.
7. Introduce a poison piece command so that each player can poison a piece. If a poisoned piece is killed the victim gains VPs instead of the killer.
8. Introduce a random game command from the main menu where the resources are randomised (but fairly with suitable rules for distribution).
9. Introduce a new blocking/wall piece that requires three connections to be destroyed (invincible for three turns).
10. Change the rules so that the connection check for killing a piece is done at the end of the turn.
11. Add the ability to modify the terrain (for a cost), or have a new piece that can modify the terrain.
12. Introduce ranged units that have connections only two edges away from their base.
13. Introduce a mini serf (or pleb) piece that only costs 1 but cannot be automatically killed if it ends the turn next to an enemy. It also does not supply chain to spawn.
14. Have an advanced game mode whereby there is one trap square on the map. A piece landing in it will be destroyed. Alternatively, give each player a trap on any field not adjacent to the opponent's Baron, once per game.
15. Change the rules so that you can choose to have an immovable Baron for a cost of 10 lumber and 1 extra fuel every turn.
16. Introduce a game timer so that each player has 180 seconds (or any constant or inputted variable) to enter their moves. Failure to complete a move in the player forfeiting the remainder of their moves; whatever they have executed (if possible), after which it will be the other player's turn.
17. Add hit points for pieces to add in the mechanic of when/how pieces are destroyed.
18. Change the game so that the coordinates system is removed and replaced by the number of hexes that you need to move through. The IDs should be used for working out / maintaining which tiles are neighbours.
19. Change the game rules so that the game is terminated immediately if a player destroys the other player's Baron. At the moment if Player 1 destroys Player 2's Baron, they still get to move.
20. Change the victory rules at the end of the game so that once the game ends, each player gets VPs for all of their remaining pieces according to the VP value for the piece.

INSPECTION COPY

**COPYRIGHT
PROTECTED**

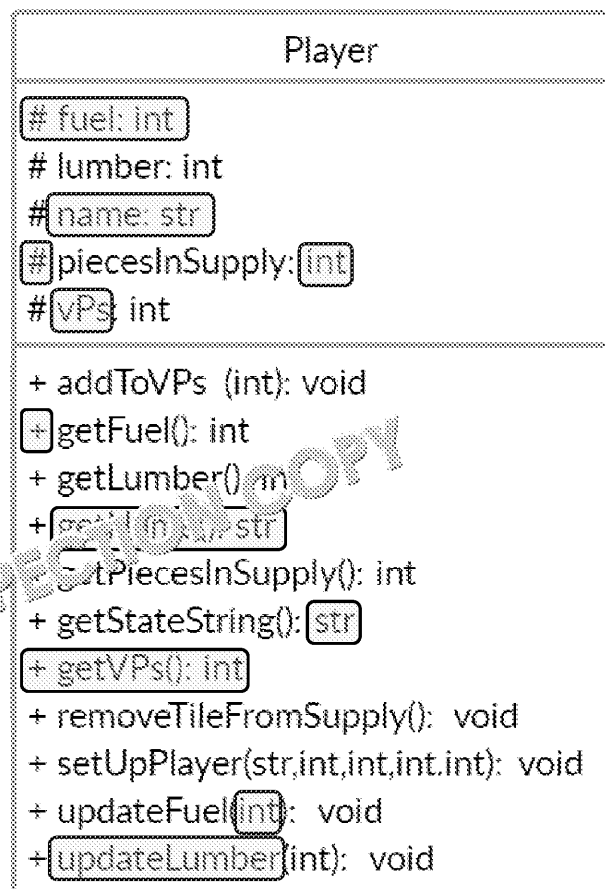


HEX BARON – Class Diagram Tasks (MS)

Task 1

- 1 mark for completing the attributes correctly (as highlighted)
- 1 mark for completing the methods correctly (as highlighted)

A. Any order (it does not need to be alphabetical)



INSPECTION COPY

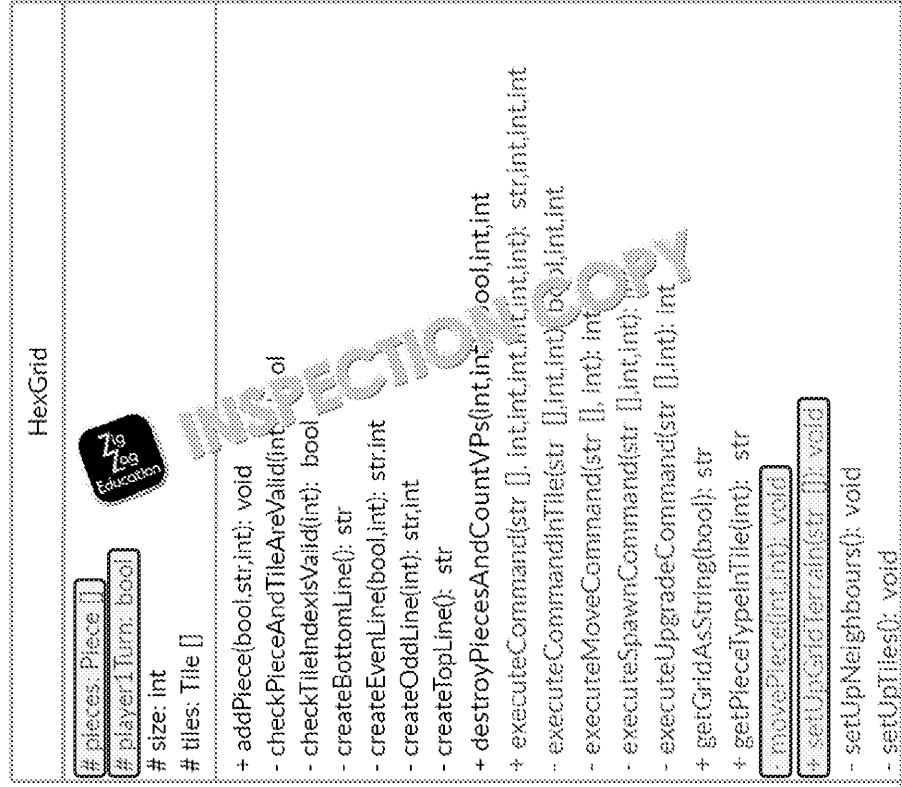
COPYRIGHT
PROTECTED



Task 2

(5 marks)

- 1 mark each for correctly completing the HexGrid and Tile classes
- 2 marks for correctly completing the Piece class (deduct 1 mark for a mostly correct attempt)
- 1 mark for adding the missing relationships



COPYRIGHT
PROTECTED



INSPECTION COPY

Task 3

- The constructors; because the convention is that they are not included (include them for automatic code generation).

1 mark for naming constructors and 1 mark for the explanation

- Methods that override methods in their parent class; because the parent class has the same name in the child class, it is clear that it has overridden the method (although some UML diagram tools use a <<override>> stereotype on the method).

1 mark for naming override (or overridden methods) and 1 mark for the explanation

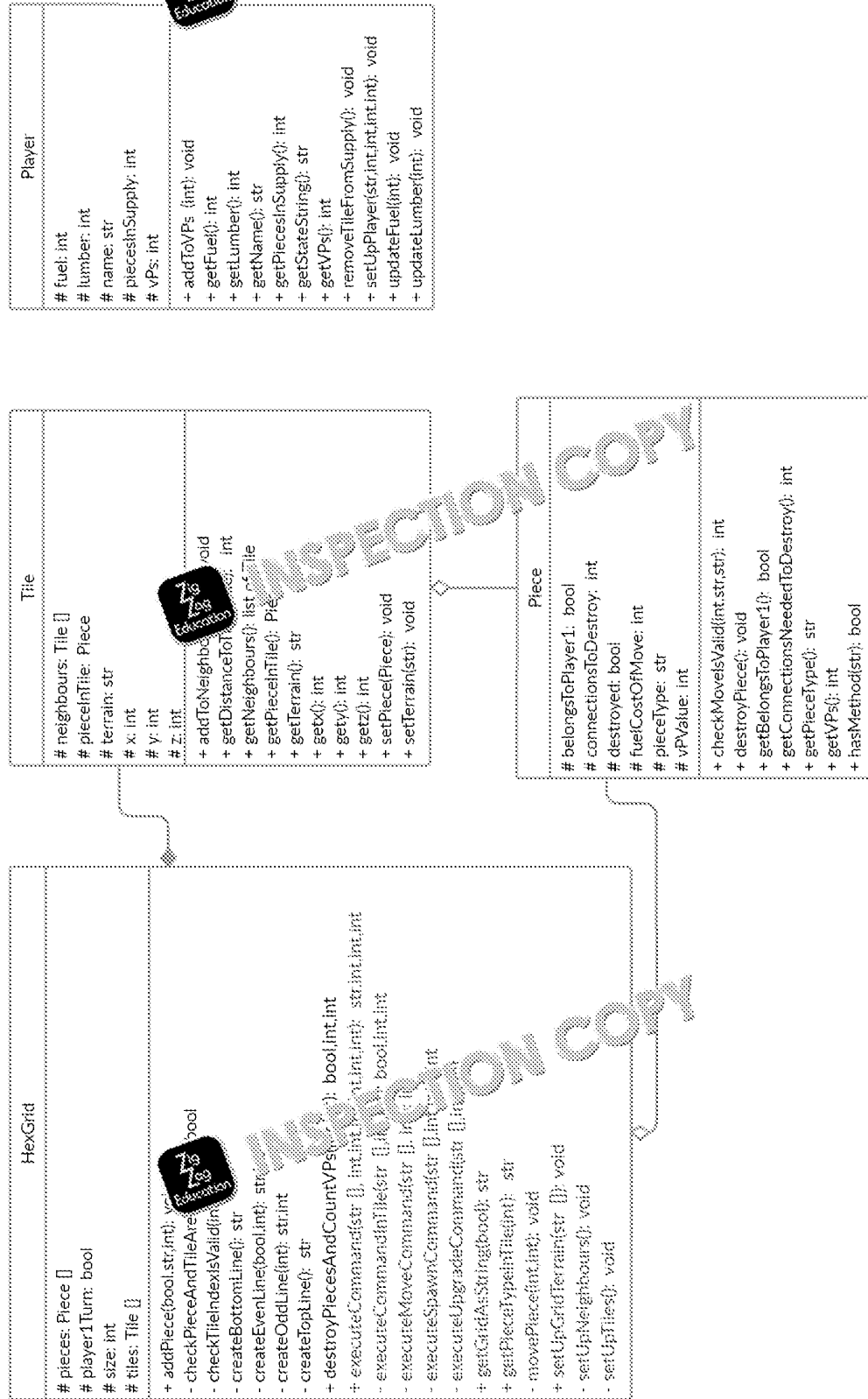


**COPYRIGHT
PROTECTED**



HEX BARON

Class Diagram



COPYRIGHT
PROTECTED



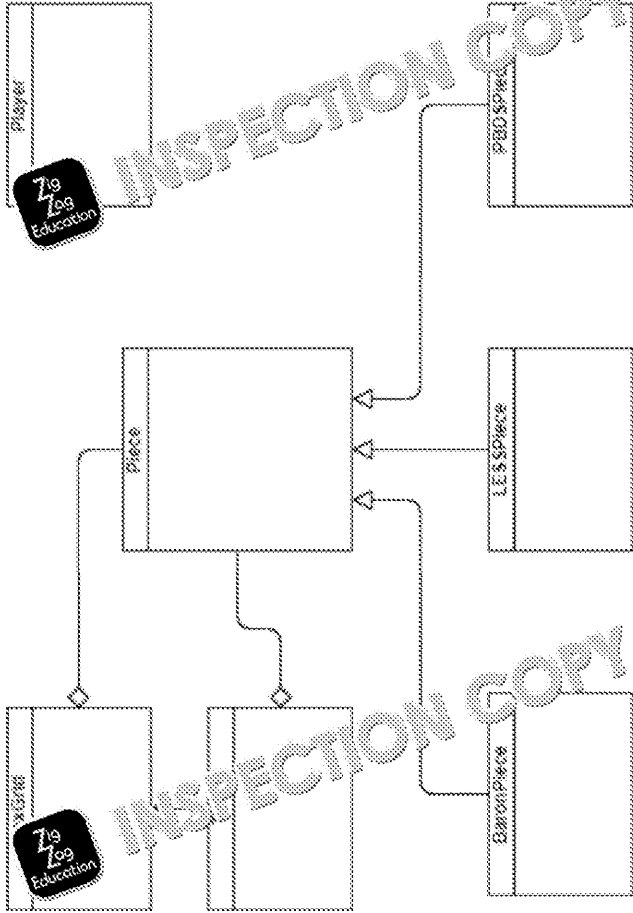
INSPECTION COPY

Question	Suggested Solution	Marks	Guidance
1			
a	 Player1.getName() and player2.getName()	1 mark	A. any sensible mention of getName()
b	 To provide an interface to private or protected attributes	1 mark	A. either private or protected A. access or another similar word for interface
2	The variable player1Turn is toggled using not player1Turn; Player One always gets their turn if Player One ends the game as the game can only end when player1Turn is true	2 marks	1 mark for each correct answer
3	hasMethod is used to check whether the command that the player typed in is available in the class for the piece in the tile	2 marks	A. to retrieve the method in the Piece class if it exists
4	It returns the maximum of three possible numbers: The absolute difference between the x coordinate of the tile t and itself; The absolute difference between the y coordinate of the tile t and itself; The absolute difference between the z coordinate of the tile t and itself	3 marks	1 mark for the first point 1 mark for any of the other three points or 2 marks for all of the last three points - MAX 2 if no mention of absolute (or equivalent)
b	First, $\max(\text{abs}(x-4), \text{abs}(-3-(-4)))$ which is $\max(2, 1)$ then $\max(2, \text{abs}(y-0))$ which is $\max(2, 0) = 0$	2 marks	1 mark for each max calculation if correct - MAX 1 mark if just $\max(2, 1, 0)$ or similar short answer
5			
a	Inheritance is where you create a new class that gains all the attributes and methods of its parent	1 mark	A. similar words or meanings, e.g. behaviours
b	Because the implementation for this is already in the Piece class	1 mark	A. answers that refer to Piece as a concrete class or talk about avoiding it being an abstract class

**COPYRIGHT
PROTECTED**



INSPECTION COPY

Question	Suggested Solution	Marks	Guidance
8	Because it is a standard format; That can be used/read easily on all platforms	1 mark	• MAX 1 mark, accept any reasonable point
9		4 marks	• 1 mark for both missing class names (Piece and PBDSPiece) • 1 mark for the composition from HexGrid to Tile • 1 mark for the aggregation from Tile to Piece • 1 mark for the two inheritance arrows from BaronPiece and LESSPiece to Piece • R. incorrect arrowhead or arrowheads on the wrong end
10 a	The upgrade was unsuccessful	1 mark	• A. any similar statement
b	The upgrade was successful AND 5 lumber should be deducted from the player	1 mark	• Only accept answers that make both points
11	In the die method a random number is generated	4 marks	• A. blank or space for field

COPYRIGHT
PROTECTED



INSPECTION COPY

Question	Suggested Solution	Marks	Guidance
13 a	upgrade (pbds less) (0 [1-9][0-9]*) 	3 marks	1 mark for upgrade and (pbds less) 1 mark for integers including ones starting with 0, such as 01, which are not strictly valid (e.g. [0-9]*) or 2 for only strictly positive integers including 0
b	Upgrade (pbds less) (0 [1-9][0-9]*)? 	1 mark	1 mark for any expression that only allows integers from 0–9 - DPT problem with integers starting with 0, e.g. 01 from part a
c	One that can be represented by a FSM (with no outputs)	1 mark	A. one that can be represented by a regular expression
14 a	gridAsStarting listPositionOfTile returnObjects	1 mark	Any one of the variables listed for 1 mark MAX
b	Their scope is limited to the current subroutine; So that means that you can test the subroutine in isolation; And, therefore, allow people to use the subroutine through its interface only so they do not need to concern themselves with the implementation	3 marks	A. any valid point about scope being the subroutine or a related advantage (e.g. no need to worry that there might be changes elsewhere) A. equivalent ideas A. method or function for subroutine
c	A private attribute is only available within the class where it is defined; Whereas a protected attribute can also be accessed by sub-classes	2 marks	A. children for sub-classes A. equivalent points presented from a different perspective
15	GROUP 1 Marks	6 marks	MAX 3 from Group 1 Marks

COPYRIGHT
PROTECTED



INSPECTION COPY

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

COPYRIGHT
PROTECTED



INSPECTION COPY

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 playerName1;
String playerName2;
HexGrid grid = new HexGrid(gridSize);
Console.WriteLine("Player One, please enter your name: ");
player1.name = Console.ReadLine();
Console.WriteLine("Player Two, please enter your name: ");
player2.name = Console.ReadLine();
player1.setUpPlayer(player1.name, 0, 10, 10, 5);
player2.setUpPlayer(player2.name, 1, 10, 10, 5);
```

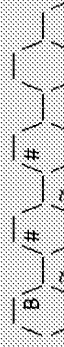
- A. Solutions which store the inputs in two variables and then pass those through to the **setUpPlayer** method when called.

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:

```
Enter your name: 1
Player One, please enter your name: Tom
Player Two, please enter your name: Vicky
Player One current state - VPs: 0   Pieces in supply: 5   Lumber: 10   Fuel: 10
Player Two current state - VPs: 1   Pieces in supply: 5   Lumber: 10   Fuel: 10
```



COPYRIGHT
PROTECTED



INSPECTION COPY

Task 3

(4 marks)

Coding:

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

For example:

addPiecesInSupply method added to the Player class:

```
public void addPiecesInSupply(int n) {  
    piecesInSupply += n;  
}
```

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);  
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

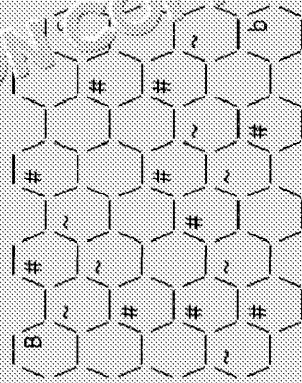
Testing:

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

For example:



Player One your three commands, pressing enter after each one.
Enter command: move 13 10
Enter command: move 10 14
Enter command: move 14 19
Command executed.
Command executed.
Command executed.
Player One current state - vps: 2 Pieces in supply: 6 Lumber: 10 Fuel: 7
Player Two current state - vps: 2 Pieces in supply: 7 Lumber: 10 Fuel: 10
Press Enter to continue...



Player Two state your three commands, pressing enter after each one.

COPYRIGHT
PROTECTED



INSPECTION COPY

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 6 pieces' when the number of pieces for the player is 6 (award even if the first mark calculate 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 (where 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> listNeighbours = new ArrayList<>(tiles.get(tileToUse).getNeighbours());
for (Tile n : listNeighbours) {
    thePiece = n.getPieceInTile();
    if (thePiece != null) {
        if (player1Turn % thePiece.getPieceType().equals("g") || !player1Turn % thePiece.getPieceType().equals("b")) {
            ownBaronIsNeighbour = true;
            break;
        }
    }
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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:

```
Player One your three commands, pressing enter after each one.  
Enter command: spawn 4  
Enter command: move 4 9  
Enter command: move 9 17  
Spawn attempted, exceed max pieces.  
Spawning did not occur.  
That move can't be done  
That move can't be done  
Player One current state - VPs: 0 Pieces in supply: 5 Number: 10 Fuel: 10
```



COPYRIGHT
PROTECTED



INSPECTION COPY

Task 5

(9 marks)

Coding:

- 1 mark for calling the new method when the 'hexes' command is entered.
- 1 mark for ensuring sure that another command can be entered after the 'hexes' command did not count as one of the three commands (no matter how many times the user enters 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 (checking 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.WriteLine("Enter command: ");  
    command = Console.ReadLine().ToLowerCase();  
    while (command != "hexes") {  
        grid.GetGridAsIndices();  
        Console.WriteLine("Enter command: ");  
        command = Console.ReadLine().ToLowerCase();  
    }  
    commands.add(command);  
}
```

Code for new GetGridAsIndices method:

```
public string GetGridAsIndices() {  
    int positionGrid = 0;  
    while (positionGrid < 9)
```

COPYRIGHT
PROTECTED



INSPECTION COPY



Code for the new create_dLineIndices method:

**COPYRIGHT
PROTECTED**



INSPECTION COPY

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);  
    }  
}
```

- A. 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.

**COPYRIGHT
PROTECTED**



INSPECTION COPY

- 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.



0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29
30	31			



```

layer One current state -
layer Two current state -
Pr ss Enter to continue..

```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

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 using the correction criteria for a selection structure to check that the player has made three valid move commands.
- 1 mark for Player One additional fuel before the third move (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 = 0; count <= 3; count++) {
        Console.WriteLine("Enter command: ");
        commands.add(Console.ReadLine().ToLower());
    }
    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;
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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:



```
Player One: your three commands, pressing enter after each one.
Enter command: move 0 4
Enter command: move 4 9
Enter command: move 9 17
Command executed
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 5 Number: 10 Fuel: 0
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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 location).
- 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 digs or 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 (tile.get(0).equals("dig")) {  
        fuel += (invokeMethod.invoke(thePiece, parameters));  
    }  
}
```

Code for the modified playGame subroutine:

```
int validDigCommands = 0;  
int validSawCommands = 0;  
String digStackLocation = "";  
Console.WriteLine(grid.GetGridAsString(player1Turn));  
if (player1Turn) {  
    Console.WriteLine(player1.GetName() + " state your three commands, pressing enter after each command.");  
} else {  
    Console.WriteLine(player2.GetName() + " state your three commands, pressing enter after each command.");  
}  
for (int count = 1; count <= 3; count++) {  
    Console.WriteLine("Enter command: ");  
    commands.add(Console.ReadLine().ToLowerCase());  
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

```

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.getItemsInSupply());
    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.getItemsInSupply());
    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);
    }
}

```



**COPYRIGHT
PROTECTED**



INSPECTION COPY

Code for new makeField method:

```
public void makeField(int tileIndex) {  
    tiles.get(tileIndex).setTerrain("");  
}
```



Code for the makeField method:

```
public int dig(String terrain) {  
    if (!terrain.equals("~")) {  
        return 0;  
    } else {  
        return 1;  
    }  
}
```

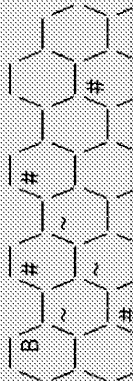


Testing:

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

For example:

```
Player One current state - Ps: 0 Pieces in supply: 5 Lumber: 5 Fuel: 12  
Player Two current state - Ps: 1 Pieces in supply: 5 Lumber: 10 Fuel: 7  
Press Enter to continue...
```



COPYRIGHT
PROTECTED



INSPECTION COPY

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 having it return false otherwise.
- 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 Pawn or a Less.
- 1 mark for correctly converting the piece to a Serf and updating the piece 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;  
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

Code for new executeDowngradeCommand method:

```
private int executeDowngradeCommand(List<String> items) {
    int tileToUse = Integer.parseInt(items.get(1));
    if (!checkPieceAndTileAreValid(tileToUse)) {
        return 0;
    } else {
        thePiece = tiles.get(tileToUse).getPieceInFile();
        if (!thePiece.getPieceType().toUpperCase().equals("P") && !thePiece.getPieceType().toUpperCase().equals("L")) {
            return 0;
        }
        thePiece.destroyPiece();
        thePiece = new Piece(player1Turn);
        pieceToAdd(thePiece);
        tiles.get(tileToUse).setPiece(thePiece);
        return 1;
    }
}
```

- A. 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:

```
Player One state your three commands, pressing enter after each one.
Enter command 1: move 16 24
Enter command: upgrade pbds 24
Enter command: downgrade 24
Command executed
Command executed
Command executed
Player One current state - VPs: 0   Pieces in supply: 5   Lumber: 6   Fuel: 8
Player Two current state - VPs: 1   Pieces in supply: 5   Lumber: 10  Fuel: 10
Press Enter to Continue
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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:

```
Player One state your three commands, pressing enter after each one.
Enter command: move 25 21
Enter command: move 21 18
Enter command: move 18 14
Command executed
Command executed
Command executed
Player One current state - VPs: 10 Pieces in supply: 5 Lumber: 10 Fuel: 6
```

```
Player Two state your three commands, pressing enter after each one.
Enter command: move 2 5
Enter command: move 5 1
Enter command: move 1 4
Command executed
Command executed
Command executed
Player One current state - VPs: 12 Pieces in supply: 5 Lumber: 10 Fuel: 6
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 10 Fuel: 4
Press Enter to continue...
```

COPYRIGHT
PROTECTED



INSPECTION COPY

Task 10

(4 marks)

Coding:

- 1 mark for modifying checkCommandsValid 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 enemy player's piece.
- 1 mark for correctly removing the piece from the board when it is salvaged.

For example:

Code for modified checkCommandsValid 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:

COPYRIGHT
PROTECTED



INSPECTION COPY

```

    }
    break;
    case "l":
    case "p":
    case "s":
        if (!player1Turn) {
            thePiece.destroyPiece();
            return -5;
        }
        break;
    }
    return 0;
}

```

A. 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:

```

Player 1 state your three commands, pressing enter after each one.
Enter command: salvage 31
Enter command: salvage 4
Enter command: salvage 8
Salvaging was not possible
Salvaging was not possible
Command executed
Player One current state - VPs: 0   Pieces in supply: 5   Lumber: 15   Fuel: 10
Player Two current state - VPs: 1   Pieces in supply: 5   Lumber: 10   Fuel: 10
Press Enter to continue...

```

COPYRIGHT
PROTECTED



INSPECTION COPY

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 adding the chances to 3 at the start of each game for all 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 entry 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();
    }
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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) {
            if (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.WriteLine("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.WriteLine("Enter new command " + (commandNo+1) + ": ");
                    command = Console.ReadLine().toLowerCase();
                }
            }
        }
    } while (true);
}
```



**COPYRIGHT
PROTECTED**



INSPECTION COPY

```

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.WriteLine("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.WriteLine("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;
        }
    }
}

```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

```

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.WriteLine("Press Enter to continue...");
    Console.ReadLine();
    } while (!gameOver || !player1Turn);
    Console.WriteLine(grid.GetGridAsString(player1Turn));
    displayEndMessages(player1, player2);
}

```

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

**COPYRIGHT
PROTECTED**



INSPECTION COPY

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 persisting between turns (e.g. Player One's second turn shows 1 chance remaining).
- 1 mark for displaying the final error message correctly and not letting Player One have a fourth chance.

For example:

```

Player One you have 3 chances remaining.
Player One state your three commands, pressing enter after each one.
Enter command 1: move
Enter command 2: move 8 8
Enter command 3: upgrade less 10
Command number 1 is an invalid command
You have 2 chances remaining.
Enter new command 1: move 8 16
Command 1 : Command executed
Command 2 : That move can't be done
Enter new command 2: move 16 20
You have 1 chances remaining.
Command 2 : Command executed
Command 3 : Upgrade not possible
Enter new command 3: move 20 28
You have 0 chances remaining.
Command 3 : Command executed
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 10 Fuel: 7
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 10 Fuel: 10
  
```

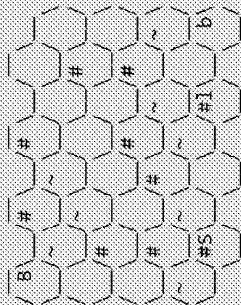
COPYRIGHT
PROTECTED



INSPECTION COPY

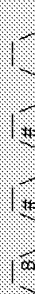
```

Player Two you have 3 chances remaining.
Player Two state your three commands, pressing enter after each one.
Enter command 1: move 23 27
Enter command 2: move 27 30
Enter command 3: upgrade less 30
Command 1 : Command executed
Command 2 : Command executed
Command 3 : Command executed
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 10 Fuel: 7
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 5 Fuel: 7
Press Enter to continue...
  
```



```

Player One you have 0 chances remaining.
Player One state your three commands, pressing enter after each one.
Enter command 1: upgrade less 28
Enter command 2: saw 28
Enter command 3: saw 29
Command 1 : Command executed
Command 2 : Command executed
Command 3 : Couldn't do that
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 6 Fuel: 7
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 5 Fuel: 7
Press Enter to continue...
  
```



Task 12

(13 marks)

Coding:

- 1 mark for modifying the checkCommandsValid method to process the new format of the spawn command.
- 1 mark for correctly checking the format of the new spawn command ideally using the checkSpawnCommandFormat method (do not award if they don't check for the availability of the lumber).
- 1 mark for modifying the spawn command in executeSpawnCommand so that it will correctly spawn a bomb and charge 10 (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 selfCostOfMove 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 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 Self piece for the appropriate player.

For example:

Code for modified checkCommandsValid 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")) {
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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 0;
    }
    boolean ownBaronIsNeighbour = false;
    List<Tile> listOfNeighbours = new ArrayList<>(tiles.get(tileToUse).getNeighbours());
    for (Tile n : listOfNeighbours) {
        thePiece = n.getPieceInTile();
        if (thePiece == null) {
            if (playerTurn && thePiece.getPieceType().equals("B")) {playerTurn = thePiece.getPieceType().equals("b")); }
            ownBaronIsNeighbour = true;
            break;
        }
    }
    if (!ownBaronIsNeighbour) {
        return -1;
    }
}
```



COPYRIGHT
PROTECTED



INSPECTION COPY

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, int startTerrain, String endTerrain) {
        if (distanceBetweenTiles == 1) {
            if (movesBeforePrimed > 0) {
                movesBeforePrimed -= 1;
                return fuelCostOfMove;
            }
            return -1;
        }

        public void destroy(Piece 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();
                    }
                }
            }
        }
    }
}
```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

```

if (tilePiece.getBelongsToPlayer1()) {
    player2VPs += tilePiece.getVPs();
} else {
    player1VPs += tilePiece.getVPs();
}
if (tilePiece.getPieceType().toLowerCase().equals("B")) {
    baronDestroyed = true;
    tilePiece.destroyPiece();
    neighbour.setPiece(null);
}
destroyed = true;
return new Object[] {baronDestroyed, player1VPs, player2VPs};

public boolean pieceDestroyed() {
    return movesPrePrimed == 0;
}
}

```

Code for modified destroyPieceAndCountVPs 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 numberOfConnections = 0;
            for (Tile n : listOfNeighbours) {

```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

```

        if (thePiece.getPieceType().toUpperCase().equals("E")) {
            baronDestroyed = true;
        }
    }
    listOfTilesContainingDestroyedPieces.add(t);
    if (thePiece.getBelongsToTopPlayer()
        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";
        }
    }
}

```



**COPYRIGHT
PROTECTED**



INSPECTION COPY

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).getPieceInfo();
    if (tiles.get(endID).getPieceInTile() != null)
        return -1;
    int distance = tiles.get(startID).getDistanceToTile(tiles.get(endID));
    int fuelCost = thePiece.checkMoveIsValid(distance, tiles.get(startID).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")) {
                Bomb pieceBomb = (BombPiece)pieceInTile;
                if (bomb.isPrimed()) {
                    bomb.destroyPiece(neighbour);
                }
            }
        }
    }
    return fuelCost;
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

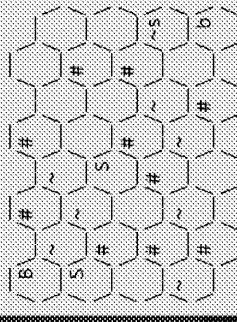
Testing:

- 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 – One:



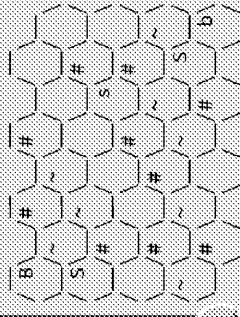
```
Player One state your three commands, pressing enter after each one.
Enter command: spawn 1 4
Enter command: move 1 2
Enter command: move 1 2
Command executed
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 4 Lumber: 20 Fuel: 20
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 30 Fuel: 30
Press Enter to continue...
```



```
Player Two state your three commands, pressing enter after each one.
Enter command: move 23 19
Enter command: move 19 11
Enter command: move 11 14
Command executed
Command executed
```



```
Player One state your three commands, pressing enter after each one.
Enter command: move 13 18
Enter command: move 18 22
Enter command: move 22 27
Command executed
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 4 Lumber: 20 Fuel: 20
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 30 Fuel: 26
Press Enter to continue...
```



```
Player Two state your three commands, pressing enter after each one.
Enter command: move 31 23
Enter command: move 14 18
Enter command: move 18 22
Command executed
Command executed
Command executed
Player One current state - VPs: 10 Pieces in supply: 4 Lumber: 20 Fuel: 20
Player Two current state - VPs: 6 Pieces in supply: 5 Lumber: 30 Fuel: 22
Press Enter to continue...
```

COPYRIGHT
PROTECTED

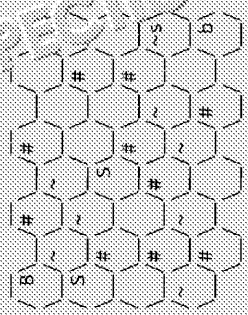


INSPECTION COPY

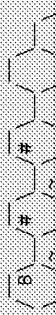
For example -- Game Two:



Player One state your three commands, pressing enter after each one.
Enter command: spawn bomb 4
Enter command: move 4 9
Enter command: move 13 13
Command executed
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 4 Lumber: 20 Fuel: 25
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 30 Fuel: 30
Press Enter to continue...

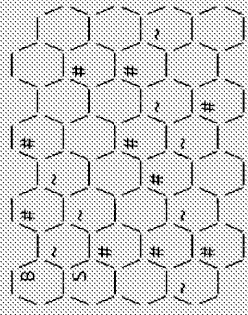


Player Two state your three commands, pressing enter after each one.
Enter command: move 23 19
Enter command: upgrade less 19
Enter command: saw 19
Command executed
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 4 Lumber: 20 Fuel: 25
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 25 Fuel: 28
Press Enter to continue...





Player One state your three commands, pressing enter after each one.
Enter command: move 13 18
Enter command: move 18 22
Enter command: move 22 27
Command executed
Command executed
Command executed
Player One current state - VPs: 13 Pieces in supply: 4 Lumber: 20 Fuel: 20
Player Two current state - VPs: 6 Pieces in supply: 5 Lumber: 26 Fuel: 28
Press Enter to continue...



Player Two state your three commands, pressing enter after each one.
Enter command: saw 19
Enter command: saw 19
Enter command: saw 19
Command executed
Command executed
Command executed
Player One current state - VPs: 13 Pieces in supply: 4 Lumber: 20 Fuel: 20
Player Two current state - VPs: 6 Pieces in supply: 5 Lumber: 26 Fuel: 28
Press Enter to continue...



COPYRIGHT
PROTECTED



INSPECTION COPY

Task 13

(10 marks)

Coding:

- 1 mark for creating the WizardPiece class with a VP value of 3.
- 1 mark for providing the checkMoveIsValid method to implement teleport.
- 1 mark for adding a new attribute to track the use of the teleport.
- 1 mark for having a resetTeleport mechanism that is activated at the end of or beginning of each turn.
- 1 mark for checking executeSpawnCommand to enable a wizard piece to be spawned.
- 1 mark for checking 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 there can do it multiple times in a turn) and charging 5 fuel but allowing moves of 1 for 1 fuel and blocking moves 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 distanceBetweenFiles, String startTerrain, String endTerrain) {
        if (distanceBetweenFiles == 1) {
```

COPYRIGHT
PROTECTED



INSPECTION COPY

Code for modified checkUpgradeCommandFormat subroutine:

```
boolean checkUpgradeCommandFormat(List<String> items) {  
    int result;  
    if (items.size() == 3) {  
        if (!items.get(1).toUpperCase().equals("WIZ") && !items.get(1).toUpperCase().equals("S")) &&  
            !items.get(1).toUpperCase().equals("WIZ") && !items.get(1).toUpperCase().equals("S")) &&  
                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 (!checkPieceAvailability(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).getPieceInFile();  
        if (!thePiece.getPieceType().toUpperCase().equals("S")) {  
            return -1;  
        }  
        thePiece.destroyPiece();  
    }
```

COPYRIGHT
PROTECTED



INSPECTION COPY

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).resetTel();  
            }  
        }  
    }  
}
```

Code for new call to resetWizards in the playGame subroutine:

```
grid.resetWizards();  
commands.clear();
```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

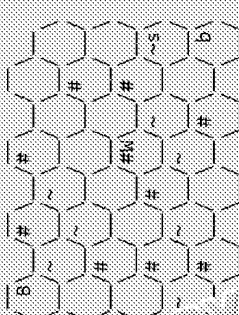
Testing:

- 1 mark for showing the commands entered correctly and the error message 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:

 player One state your three commands, pressing enter after each one.
Enter command: move 8 8
Enter command: move 8 13
Enter command: move 13 18
That move can't be done
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 5 Fuel: 8

 player One state your three commands, pressing enter after each one.
Enter command: upgrade wiz 8
Enter command: move 8 13
Enter command: move 13 18
Command executed
Command executed
Command executed
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 5 Fuel: 14
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 10 Fuel: 10
Press Enter to continue...



player Two state your three commands, pressing enter after each one.
Enter command: move 23 19
Enter command: move 19 11
Enter command: upgrade less 11
Command executed
Command executed
Command executed

COPYRIGHT
PROTECTED



INSPECTION COPY

Task 14

(10 marks)

Coding:

- 1 mark for creating the new SCFWPiece class and setting the VPValue to 3.
- 1 mark for providing the checkMoveIsValid method (or any other suitable 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 disabling the new piece as 'F' for Player One and charging the 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};
        }
    }
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

Code for the modified checkCommandIsValid subroutine:

```
case "move":
    return checkMoveCommandFormat(items);
case "supply":
    if (!supplyFirst) {
        return false;
    }
case "did":
case "":
case "scan":
    return checkStandardCommandFormat(items);
```

Code for the modified executeUpgradeCommandFormat subroutine:

```
boolean checkUpgradeCommandFormat(List<String> items)
int result;
if (items.size() == 3) {
    if (!items.get(1).toUpperCase().equals("LESS") &
        !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:

COPYRIGHT
PROTECTED



INSPECTION COPY

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 (checkPieceValid(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 };
        }
    }
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY


```

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.");
        se {
            Console.WriteLine(player2.getName() + " state your three commands, pressing enter after each one.");
        }
        for (int count = 1; count <= 3; count++) {
            Console.WriteLine("Enter command: ");
            string command = Console.ReadLine().ToLowerCase();
            commands.add(command);
            if (command <String> items = Arrays.asList(commands).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());
                }
            }
        }
    }
} while (true);

```

**COPYRIGHT
PROTECTED**



INSPECTION COPY


```

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;
}

```

- A. 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).
- A. 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 Unit object.

**COPYRIGHT
PROTECTED**



INSPECTION COPY

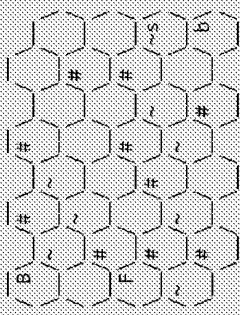
Testing:

- 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 it was terminated when it was entered.

For example:

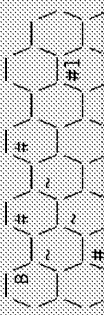


player One state your three commands, pressing enter after each one.
Enter command: move 8 16
Enter command: upgrade scfw 16
Enter command: supply 16
Command executed
Invalid command
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 25 Fuel: 29
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 10 Fuel: 10
Press Enter to continue...





player Two state your three commands, pressing enter after each one.
Enter command: move 23 19
Enter command: move 19 11
Enter command: upgrade less 11
Command executed
Invalid command
Player One current state - VPs: 0 Pieces in supply: 5 Lumber: 25 Fuel: 29
Player Two current state - VPs: 1 Pieces in supply: 5 Lumber: 5 Fuel: 7
Press Enter to continue...



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 adding the loadGame subroutine so that it accepts fileName as a parameter correctly and doesn't ask for it as 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 fileName 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("4. 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: ");  
}
```

COPYRIGHT
PROTECTED



INSPECTION COPY

```
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.File;
import java.io.*;
```

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;
```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

```

Console.WriteLine();
Console.WriteLine("Please enter the grid size (6, 8 or 10): ");
gridSize = Integer.parseInt(Console.ReadLine());
while (gridSize != 6 && gridSize != 8 && gridSize != 10) {
    Console.WriteLine("Invalid grid size, please re-enter (6, 8 or 10): ");
    Size = Integer.parseInt(Console.ReadLine());
}
grid = new HexGrid(gridSize);
for (int terrainIndex=0; terrainIndex<(gridSize*gridSize/2); terrainIndex++) {
    terrainList.add("");
}
grid.setGridTerrain(terrainList);
Console.WriteLine(grid.GetGridAsIndices());

Console.WriteLine("Please enter all the locations of peat bogs separated by commas with no spaces: ");
peatBogs = Console.ReadLine().Split(",");
Console.WriteLine("Please enter all the locations of forests separated by commas with no spaces: ");
forests = 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.WriteLine("Please enter the location of the Baron for Player 1: ");
pllocs.add(Integer.parseInt(Console.ReadLine()));
plpieces.add("Baron");

```

**COPYRIGHT
PROTECTED**



INSPECTION COPY

```

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 1 (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: ");
        p1stats.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")) {

```

**COPYRIGHT
PROTECTED**



INSPECTION COPY


```

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.WriteLine("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 i=0; i<stat : p1stats) {
        p1string += ","+stat;
    }
    out.WriteLine(p1string+"\n");
    for (int i=0; i<stat : p2stats) {
        p2string += ","+stat;
    }
    out.WriteLine(p2string+"\n");
    out.WriteLine("Grid Size "+gridSize+"\n");
    for (int i=0; i<terrainIndex-1; i++) {
        terrainIndex = newTerrain.size();
        gridString += ","+newTerrain.get(terrainIndex);
    }
    out.WriteLine(gridString+"\n");
    for (int i=0; i<plIndex-1; i++) {
        plIndex = p1pieces.size();
        out.WriteLine("1,"+p1pieces.get(plIndex)+","+p1locs.get(plIndex)+"\n");
    }
    for (int i=0; i<p2Index-1; i++) {
        p2Index = p2pieces.size();
        out.WriteLine("2,"+p2pieces.get(p2Index)+","+p2locs.get(p2Index)+"\n");
    }
}

```



**COPYRIGHT
PROTECTED**



INSPECTION COPY

- 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.



Player One state your three commands, pressing enter after each one.
Enter command:

test.csv file.

```
What name would you like to save the file as (please include the extension): test.csv
Would you like to play the game that you've created and saved? y
Player One current state - VPs: 0 Pieces in supply: 5 Number: 15 Fuel: 12
Player Two current state - VPs: 0 Pieces in supply: 5 Number: 15 Fuel: 12
```

Player One state your three commands, pressing enter after each one.
Enter command:

test.csv file.

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Name

ZigZag Education supporting

A Level AQA Computer Science Paper

Summer 2021

 **HEX BARON**

Electronic Answer Document (EAD)

Instructions

- Enter your name in the box at the top of this page
- Answer **all** questions by entering your answers into this document
- Remember to **save** this document regularly
- Save and print this document and any additional pages
- Answer **all** questions
- The marks available for each question are shown in brackets
- You will need:
 - ☐ access to a computer
 - ☐ access to a printer
 - ☐ access to appropriate software
 - ☐ electronic copies of the required skeleton code
 - ☐ EAD (Electronic Answer Document)

Total marks:

INSPECTION COPY

**COPYRIGHT
PROTECTED**



Programming Theory Question

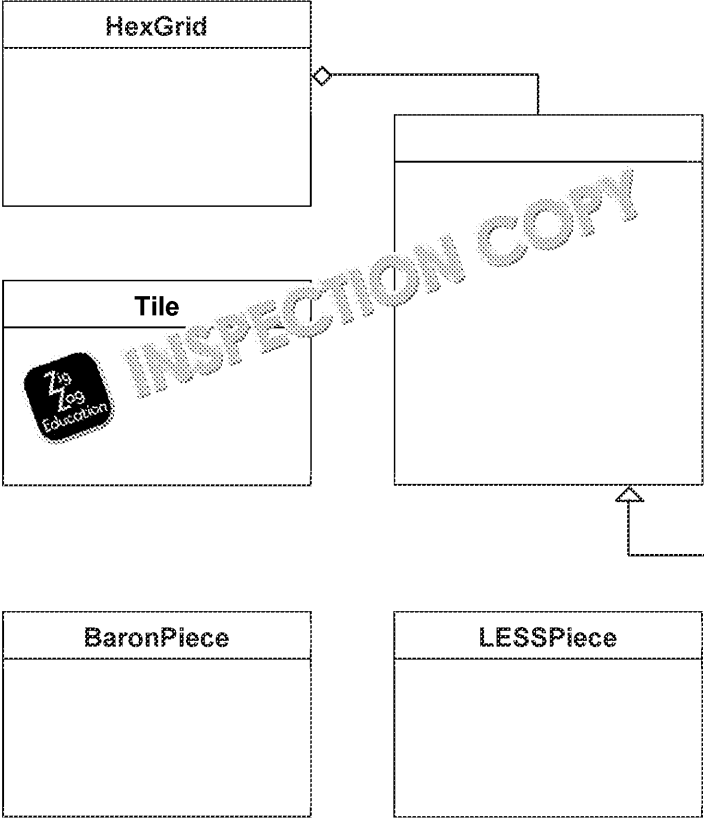
Answer all questions. Remember to save this document

Q	Answer
1	(a)
	(b)
2	
3	
4	(a)
	(b)
5	(a)
	(b)
6	(a)
	(b)
7	(a)
	(b)
8	

INSPECTION COPY

COPYRIGHT
PROTECTED



9	 <pre> classDiagram class HexGrid class Tile class BaronPiece class LESSPiece HexGrid "1" *-- "1" Tile Tile < -- BaronPiece Tile < -- LESSPiece </pre>						
10	<table border="1"> <tr> <td data-bbox="180 1028 247 1104">(a)</td><td data-bbox="247 1028 1077 1104"></td></tr> <tr> <td data-bbox="180 1104 247 1178">(b)</td><td data-bbox="247 1104 1077 1178"></td></tr> </table>	(a)		(b)			
(a)							
(b)							
11							
12	<table border="1"> <tr> <td data-bbox="180 1252 247 1328">(a)</td><td data-bbox="247 1252 1077 1328"></td></tr> <tr> <td data-bbox="180 1328 247 1402">(b)</td><td data-bbox="247 1328 1077 1402"></td></tr> </table>	(a)		(b)			
(a)							
(b)							
13	<table border="1"> <tr> <td data-bbox="180 1402 247 1478">(a)</td><td data-bbox="247 1402 1077 1478"></td></tr> <tr> <td data-bbox="180 1478 247 1554">(b)</td><td data-bbox="247 1478 1077 1554"></td></tr> <tr> <td data-bbox="180 1554 247 1626">(c)</td><td data-bbox="247 1554 1077 1626"></td></tr> </table>	(a)		(b)		(c)	
(a)							
(b)							
(c)							
14	<table border="1"> <tr> <td data-bbox="180 1626 247 1702">(a)</td><td data-bbox="247 1626 1077 1702"></td></tr> <tr> <td data-bbox="180 1702 247 1778">(b)</td><td data-bbox="247 1702 1077 1778"></td></tr> <tr> <td data-bbox="180 1778 247 1850">(c)</td><td data-bbox="247 1778 1077 1850"></td></tr> </table>	(a)		(b)		(c)	
(a)							
(b)							
(c)							
15							

**COPYRIGHT
PROTECTED**



Programming Tasks

Answer all questions. Remember to save this document

Q	Answer
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

INSPECTION COPY

COPYRIGHT
PROTECTED

