

**Programming Tasks (Mark Scheme) Part 1**

# Task 1 (2 marks)

## Coding

* Create a new method CreateCustomPlayers which allows the user to enter in two player names and won’t continue until the names are different. **[1 mark]**

### Example Solution

Modify constructor in Dastan:

public Dastan(int R, int C, int NoOfPieces)

{

//CHANGE

CreateCustomPlayers(); //Q1

//CHANGE

CreateMoveOptions();

New private method:

//CHANGE

private void CreateCustomPlayers() //Q1

{

Console.WriteLine("Enter in the name for player one: ");

Players.Add(new Player(Console.ReadLine(), 1));

bool NameInvalid = true;

string PlayerTwoName = "";

while (NameInvalid)

{

Console.WriteLine("Enter in the second player name: ");

PlayerTwoName = Console.ReadLine();

if (Players[0].GetName() == PlayerTwoName)

{

Console.WriteLine("You can't have that name - it has already been taken by player 1");

}

else

{

NameInvalid = false;

}

}

Players.Add(new Player(PlayerTwoName, -1));

}

//END CHANGE

## Testing:

* Display an appropriate error message if the user enters in two matching names. Correctly identify player one with a custom name. **[1 mark]**

Text

Description automatically generated

# Task 2 (4 marks)

## Coding

* Adding a new MoveOptionOffer to the CreateMoveOptionOffer method, a new option in the CreateMoveOption method, and adding the move option to both players in the Player list with the Direction parameter set correctly. [1 mark]
* Adding a Faris to the CreateMoveOption method, a new option which calls the CreateFarisMoveOption method. [1 mark]
* Create a new method CreateFarisMoveOption which correctly uses the Direction parameter to identify all of the possible positions for the Faris move. [1 mark]

### Example Solution

Changes to CreateMoveOptionOffer:

private void CreateMoveOptionOffer()

{

//CHANGE

MoveOptionOffer.Add("faris"); //Q2

//END CHANGE

Changes to CreateMoveOption:

private MoveOption CreateMoveOption(string Name, int Direction)

{

//CHANGE

if (Name == "faris")

{

return CreateFarisMoveOption(Direction); //Q2

}

else if (Name == "chowkidar")

{

return CreateChowkidarMoveOption(Direction);

}

//END CHANGE

Code for CreateFarisMoveOption:

//CHANGE

private MoveOption CreateFarisMoveOption(int Direction) //Q2

{

MoveOption NewMoveOption = new MoveOption("faris");

Move NewMove = new Move(-1 \* Direction, -2 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(-2 \* Direction, -1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(-2 \* Direction, 1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(-1 \* Direction, 2 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(1 \* Direction, -2 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(2 \* Direction, -1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(2 \* Direction, 1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(1 \* Direction, 2 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

return NewMoveOption;

}

//END CHANGE

Changes to CreateMoveOptions:

private void CreateMoveOptions()

{

Players[0].AddToMoveOptionQueue(CreateMoveOption("ryott", 1));

//CHANGE

Players[0].AddToMoveOptionQueue(CreateMoveOption("faris", 1)); //Q2

//END CHANGE

Players[0].AddToMoveOptionQueue(CreateMoveOption("chowkidar", 1));

Players[0].AddToMoveOptionQueue(CreateMoveOption("cuirassier", 1));

Players[0].AddToMoveOptionQueue(CreateMoveOption("faujdar", 1));

Players[0].AddToMoveOptionQueue(CreateMoveOption("jazair", 1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("ryott", -1));

//CHANGE

Players[1].AddToMoveOptionQueue(CreateMoveOption("faris", -1)); //Q2

//END CHANGE

Players[1].AddToMoveOptionQueue(CreateMoveOption("chowkidar", -1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("jazair", -1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("faujdar", -1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("cuirassier", -1));

}

## Testing:

* Displaying the Faris move option correctly in the player one queue and moving a player one piece appropriately for a legal Faris move. [1 mark]

Text

Description automatically generated

# Task 3 (4 marks)

## Coding

* Adding a new MoveOptionOffer to the CreateMoveOptionOffer method, a new option in the CreateMoveOption method, and adding the move option to both players in the Player list with the Direction parameter set correctly. **[1 mark]**
* Adding a Sarukh to the CreateMoveOption method, a new option which calls the CreateSarukhMoveOption method. **[1 mark]**
* Create a new method CreateSarukhMoveOption which correctly uses the Direction parameter to identify all of the possible positions for the Sarukh move. **[1 mark]**

### Example Solution

Changes to CreateMoveOptionOffer:

private void CreateMoveOptionOffer()

{

//CHANGE

MoveOptionOffer.Add("sarukh"); //Q3

//END CHANGE

Changes to CreateMoveOption:

private MoveOption CreateMoveOption(string Name, int Direction)

{

//CHANGE

if (Name == "sarukh")

{

return CreateSarukhMoveOption(Direction); //Q3

}

else if (Name == "chowkidar")

{

return CreateChowkidarMoveOption(Direction);

}

//END CHANGE

Code for CreateSarukhMoveOption:

//CHANGE

private MoveOption CreateSarukhMoveOption(int Direction) //Q3

{

MoveOption NewMoveOption = new MoveOption("sarukh");

Move NewMove = new Move(0, -1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(1 \* Direction, -1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(2 \* Direction, 0);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(1 \* Direction, 1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

NewMove = new Move(0, 1 \* Direction);

NewMoveOption.AddToPossibleMoves(NewMove);

return NewMoveOption;

}

//END CHANGE

Changes to CreateMoveOptions:

private void CreateMoveOptions()

{

Players[0].AddToMoveOptionQueue(CreateMoveOption("ryott", 1));

//CHANGE

Players[0].AddToMoveOptionQueue(CreateMoveOption("sarukh", 1)); //Q3

//END CHANGE

Players[0].AddToMoveOptionQueue(CreateMoveOption("chowkidar", 1));

Players[0].AddToMoveOptionQueue(CreateMoveOption("cuirassier", 1));

Players[0].AddToMoveOptionQueue(CreateMoveOption("faujdar", 1));

Players[0].AddToMoveOptionQueue(CreateMoveOption("jazair", 1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("ryott", -1));

//CHANGE

Players[1].AddToMoveOptionQueue(CreateMoveOption("sarukh", -1)); //Q3

//END CHANGE

Players[1].AddToMoveOptionQueue(CreateMoveOption("chowkidar", -1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("jazair", -1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("faujdar", -1));

Players[1].AddToMoveOptionQueue(CreateMoveOption("cuirassier", -1));

}

## Testing:

* Displaying the Sarukh move option correctly in the player one queue and moving a player one piece appropriately for a legal Sarukh move. **[1 mark]**

Text

Description automatically generated

# Task 4 (5 marks)

## Coding

* Change PlayGame to randomly award a Wafr to the current player and if one has been awarded, advise the player that they can select any queue position without cost. **[1 mark]**
* Change PlayGame so that if a move is legal and a Wafr has been awarded to the player, the move does not incur a points cost to the player. **[1 mark]**
* Create a new method AwardWafr in the Dastan class which has a 25% chance of returning true. **[1 mark]**
* Adding the WafrAwarded attribute to Player with get/set methods for WafrAwarded. **[1 mark]**

### Example Solution

Changes to Player:

class Player

{

private string Name;

private int Direction, Score;

private MoveOptionQueue Queue = new MoveOptionQueue();

//CHANGE

private bool WafrAwarded = false; //Q4

//END CHANGE

//CHANGE

public bool GetWafrAwarded() //Q4

{

return WafrAwarded;

}

public void SetWafrAwarded() //Q4

{

WafrAwarded = true;

}

//END CHANGE

Code for AwardWafr:

//CHANGE

private bool AwardWafr() //Q4

{

if (RGen.Next(0, 4) == 0) //25% chance of returning true.

{

return true;

}

else

{

return false;

}

}

//END CHANGE

Changes to PlayGame:

public void PlayGame()

{

bool GameOver = false;

while (!GameOver)

{

DisplayState();

bool SquareIsValid = false;

//CHANGE

int Choice = 0; //Q4

bool Wafr = false;

if (AwardWafr() && !CurrentPlayer.GetWafrAwarded())

{

Console.WriteLine("You have been offered a Wafr!");

Console.WriteLine("You can select any move from your queue for free this turn.");

Wafr = true;

CurrentPlayer.SetWafrAwarded();

}

if (Wafr)

{

do

{

Console.Write("Choose move option to use from queue (1 to 5): ");

Choice = Convert.ToInt32(Console.ReadLine());

}

while (Choice < 1 || Choice > 5);

}

else

{

do

{

Console.Write("Choose move option to use from queue (1 to 3) or 9 to take the offer: ");

Choice = Convert.ToInt32(Console.ReadLine());

if (Choice == 9)

{

UseMoveOptionOffer();

DisplayState();

}

}

while (Choice < 1 || Choice > 3);

}

//END CHANGE

int StartSquareReference = 0;

while (!SquareIsValid)

{

StartSquareReference = GetSquareReference("containing the piece to move");

SquareIsValid = CheckSquareIsValid(StartSquareReference, true);

}

int FinishSquareReference = 0;

SquareIsValid = false;

while (!SquareIsValid)

{

FinishSquareReference = GetSquareReference("to move to");

SquareIsValid = CheckSquareIsValid(FinishSquareReference, false);

}

bool MoveLegal = CurrentPlayer.CheckPlayerMove(Choice, StartSquareReference, FinishSquareReference);

if (MoveLegal)

{

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

//CHANGE

if (!Wafr) //Q4

{

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

}

else

{

Console.WriteLine();

Console.WriteLine("Using Wafr move at zero cost");

}

//END CHANGE

## Testing:

* Show player one being awarded a Wafr and selecting a move from position 4 or 5 in the move option queue without incurring a cost. **[1 mark]**

Text

Description automatically generated

# Task 5 (5 marks)

## Coding

* Change PlayGame to give new menu option 8 and reduction of player score by 5 if option 8 is selected. **[1 mark]**
* Adding the Opponent variable (or similar) to PlayGame and correctly assigning it to the player whose turn it is not **[1 mark]**
* Correctly printing out the opponent’s queue. **[1 mark]**
* Creation of GetJustQueueAsString which calls the GetQueueAsString method for the private attribute Queue in Player. **[1 mark]**

### Example Solution

Changes to PlayGame:

do

{

//CHANGE

Console.Write("Choose move option to use from queue (1 to 3) or 9 to take the offer or 8 to spy on your opponent's queue: "); //Q5

Choice = Convert.ToInt32(Console.ReadLine());

if (Choice == 8)

{

Console.WriteLine("Selecting this option costs 5 score points." + Environment.NewLine);

Player Opponent;

if (CurrentPlayer.SameAs(Players[0]))

{

Opponent = Players[1];

}

else

{

Opponent = Players[0];

}

Console.WriteLine(Opponent.GetName() + "'s queue is: ");

Console.WriteLine(Opponent.GetJustQueueAsString() + Environment.NewLine);

CurrentPlayer.ChangeScore(-5);

DisplayState();

}

else if (Choice == 9)

{

UseMoveOptionOffer();

DisplayState();

}

//END CHANGE

}

Changes to Player:

//CHANGE

public string GetJustQueueAsString() //Q5

{

return Queue.GetQueueAsString();

}

//END CHANGE

## Testing:

* Display new menu option. Player one to select option 8 to view player two’s queue. **[1 mark]**

Text

Description automatically generated

# Task 6 (5 marks)

## Coding

* Create a new method GetValidInt which returns the user input if the user enters in a valid integer and if required, also if it is in range. It should print out a suitable message and force the user to retry until they have entered a valid integer or in range. **[1 mark]**
* Change PlayGame to use the GetValidInt method on the main game menu to only allow valid integer input for the move queue choice. **[1 mark]**
* Change GetSquareReference to use the GetValidInt method for choosing a position square to only allow valid integer input. **[1 mark]**
* Change UseMoveOptionOffer to use the GetValidInt method for choosing which move option to replace. Only allow valid integer input and include a range of 1 to 5 to prevent an invalid queue position being selected. **[1 mark]**

### Example Solution

Code for GetValidInt:

//CHANGE

private int GetValidInt(string MessageToDisplay, int OptionalRange = 0) //Q6

{

bool ValidInput = false;

int UserInput = -1;

while (!ValidInput)

{

Console.WriteLine(MessageToDisplay);

try

{

UserInput = Convert.ToInt32(Console.ReadLine());

if (OptionalRange != 0)

{

if (UserInput > 0 && UserInput <= OptionalRange)

{

ValidInput = true;

}

else

{

Console.WriteLine("Value out of range - try again.");

}

}

else

{

ValidInput = true; //Condition where range handling is done elsewhere in the code

}

}

catch

{

Console.WriteLine("Not a valid input - try again" + Environment.NewLine);

}

}

return UserInput;

}

//END CHANGE

Changes to PlayGame:

do

{

//CHANGE

Choice = GetValidInt("Choose move option to use from queue (1 to 3) or 9 to take the offer: "); //Q6

if (Choice == 9)

{

UseMoveOptionOffer();

DisplayState();

}

else if (Choice < 1 || Choice > 3)

{

Console.WriteLine("That is not a valid option - select from option 1 to 3 or 9 to take the offer:" + Environment.NewLine);

}

}

while (Choice < 1 || Choice > 3);

int StartSquareReference = 0;

while (!SquareIsValid)

{

StartSquareReference = GetSquareReference("containing the piece to move");

SquareIsValid = CheckSquareIsValid(StartSquareReference, true);

if (!SquareIsValid)

{

Console.WriteLine("You must enter a valid square");

}

}

int FinishSquareReference = 0;

SquareIsValid = false;

while (!SquareIsValid)

{

FinishSquareReference = GetSquareReference("to move to");

SquareIsValid = CheckSquareIsValid(FinishSquareReference, false);

if (!SquareIsValid)

{

Console.WriteLine("You must enter a valid square");

}

}

bool MoveLegal = CurrentPlayer.CheckPlayerMove(Choice, StartSquareReference, FinishSquareReference);

//END CHANGE

Changes to GetSquareReference:

//CHANGE

private int GetSquareReference(string Description) //Q6

{

int SelectedSquare;

SelectedSquare = GetValidInt("Enter the square " + Description + " (row number followed by column number): ");

return SelectedSquare;

}

//END CHANGE

Changes to UseMoveOptionOffer:

//CHANGE

private void UseMoveOptionOffer() //Q6

{

int ReplaceChoice;

ReplaceChoice = GetValidInt("Choose the move option from your queue to replace (1 to 5): ", 5);

CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1, CreateMoveOption(MoveOptionOffer[MoveOptionOfferPosition], CurrentPlayer.GetDirection()));

CurrentPlayer.ChangeScore(-(10 - (ReplaceChoice \* 2)));

MoveOptionOfferPosition = RGen.Next(0, 5);

}

//END CHANGE

## Testing:

* Display an appropriate error message if the user enters in non-valid inputs for the main game menu and as a position to place MoveOptionOffer in the queue. **[1 mark]**

Text

Description automatically generated

# Task 7 (5 marks)

## Coding

* Adding the ChoiceOptionsLeft attribute to Player with getter method. Initialising attribute to 3. **[1 mark]**
* Create a new method DecreaseChoiceOptionsLeft in Player which decrements ChoiceOptionsLeft.   
  **[1 mark]**
* Change PlayGame to test if the player has used all of their offer options and if so, do not display option 9. **[1 mark]**
* Change UseMoveOptionOffer to call DecreaseChoiceOptionsLeft for the current player when they choose a move option from the menu AND advise the player how many move options they have left. **[1 mark]**

### Example Solution

Changes to Player:

class Player

{

private string Name;

private int Direction, Score;

private MoveOptionQueue Queue = new MoveOptionQueue();

//CHANGE

private int ChoiceOptionsLeft = 3; //Q7

//CHANGE

//CHANGE

public void DecreaseChoiceOptionsLeft() //Q7

{

ChoiceOptionsLeft--;

}

public int GetChoiceOptionsLeft() //Q7

{

return ChoiceOptionsLeft;

}

//END CHANGE

Changes to PlayGame:

do

{

//CHANGE

if (CurrentPlayer.GetChoiceOptionsLeft() > 0) //Q7

{

Console.Write("Choose move option to use from queue (1 to 3) or 9 to take the offer: ");

}

else

{

Console.Write("Choose move option to use from queue (1 to 3): ");

}

Choice = Convert.ToInt32(Console.ReadLine());

if (CurrentPlayer.GetChoiceOptionsLeft() > 0 && Choice == 9)

{

UseMoveOptionOffer();

DisplayState();

}

//END CHANGE

}

Changes to UseMoveOptionOffer:

private void UseMoveOptionOffer()

{

int ReplaceChoice;

Console.Write("Choose the move option from your queue to replace (1 to 5): ");

ReplaceChoice = Convert.ToInt32(Console.ReadLine());

CurrentPlayer.UpdateMoveOptionQueueWithOffer(ReplaceChoice - 1, CreateMoveOption(MoveOptionOffer[MoveOptionOfferPosition], CurrentPlayer.GetDirection()));

CurrentPlayer.ChangeScore(-(10 - (ReplaceChoice \* 2)));

MoveOptionOfferPosition = RGen.Next(0, 5);

//CHANGE

CurrentPlayer.DecreaseChoiceOptionsLeft(); //Q7

Console.WriteLine("You have " + Convert.ToString(CurrentPlayer.GetChoiceOptionsLeft() + " move options left."));

//END CHANGE

}

## Testing:

* Show player one selecting a move from the move option offer menu and decreasing the offer counter by 1. **[1 mark]**

Text

Description automatically generated

Text

Description automatically generated

# Task 8 (5 marks)

## Coding

* Writing the ResetQueueBackAfterUndo method which calls the ResetQueueBack method with one parameter and successfully pops the item from the end of the queue and returns it to its original position. **[1 mark]**
* Asking the player if they would like to undo after they have played their move and seen its effect. **[1 mark]**
* Correctly handling the undo to deduct 5 points and reset the board and queue. **[1 mark]**

### Example Solution

Changes to Player:

//CHANGE

public void ResetQueueBackAfterUndo(int Position) //Q8

{

Queue.ResetQueueBack(Position);

}

//END CHANGE

Changes to MoveOptionQueue:

//CHANGE

public void ResetQueueBack(int Position) //Q8

{

MoveOption RearElement = Queue[Queue.Count - 1];

Queue.RemoveAt(Queue.Count - 1);

Queue.Insert(Position - 1, RearElement);

}

//END CHANGE

Changes to PlayGame:

if (MoveLegal)

{

//CHANGE

int StartScore = CurrentPlayer.GetScore(); //Q8

//END CHANGE

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

CurrentPlayer.UpdateQueueAfterMove(Choice);

UpdateBoard(StartSquareReference, FinishSquareReference);

UpdatePlayerScore(PointsForPieceCapture);

Console.WriteLine("New score: " + CurrentPlayer.GetScore() + Environment.NewLine);

//CHANGE

int UndoScore = StartScore - CurrentPlayer.GetScore(); //Q8

Console.WriteLine("Would you like to undo this move at the cost of 5 score points? yes/no");

string UndoChoice = Console.ReadLine().ToLower();

if (UndoChoice == "yes")

{

CurrentPlayer.ChangeScore(UndoScore - 5);

UpdateBoard(FinishSquareReference, StartSquareReference);

CurrentPlayer.ResetQueueBackAfterUndo(Choice);

}

else

{

if (CurrentPlayer.SameAs(Players[0]))

{

CurrentPlayer = Players[1];

}

else

{

CurrentPlayer = Players[0];

}

GameOver = CheckIfGameOver();

}

}

//END CHANGE

## Testing:

* Showing that a move can be undone and that 5 points are deducted. **[1 mark]**
* Showing that the same player can still play their turn and that the game can continue normally after an undo.   
  [1 mark]

Text

Description automatically generated

Text

Description automatically generated

# Task 9 (7 marks)

## Coding

* CreateMoveOptionOffer has been modified to append “sahm” as a move option offer and correctly dealing with the Name parameter of “sahm” in the CreateMoveOption method. [1 mark]
* Making the Sahm the move option for both players on their first turn. **[1 mark]**
* Correctly creating the SahmUsed attribute with getter/setter methods. **[1 mark]**
* Only allowing a player to fire a single Sahm in a game. **[1 mark]**
* Correctly removing all the pieces in the Sahm’s line of fire from the board (except any in a Kotla) in the CalculateSahmMove method. **[1 mark]**
* Correctly awarding points for all removed/destroyed pieces (even if a piece was removed in error) in the CalculateSahmMove method. **[1 mark]**

### Example Solution

Changes to CreateMoveOptionOffer:

private void CreateMoveOptionOffer()

{

//CHANGE

MoveOptionOffer.Add("sahm"); //Q9

//END CHANGE

MoveOptionOffer.Add("jazair");

MoveOptionOffer.Add("chowkidar");

MoveOptionOffer.Add("cuirassier");

MoveOptionOffer.Add("ryott");

MoveOptionOffer.Add("faujdar");

}

Code for CreateSahmMoveOption:

//CHANGE

private MoveOption CreateSahmMoveOption(int Direction) //Q9

{

MoveOption NewMoveOption = new MoveOption("sahm");

Move NewMove = new Move(0, 0);

NewMoveOption.AddToPossibleMoves(NewMove);

return NewMoveOption;

}

//END CHANGE

Changes to CreateMoveOption:

private MoveOption CreateMoveOption(string Name, int Direction)

{

//CHANGE

if (Name == "sahm") //Q9

{

return CreateSahmMoveOption(Direction);

}

else if (Name == "chowkidar")

//END CHANGE

Changes to PlayGame:

while (!SquareIsValid)

{

StartSquareReference = GetSquareReference("containing the piece to move");

SquareIsValid = CheckSquareIsValid(StartSquareReference, true);

}

//CHANGE

if (CurrentPlayer.ChoiceIsSahm(Choice)) //Q9

{

if (CurrentPlayer.GetSahmUsed())

{

Console.WriteLine("You have already used your Sahm Move Option. Please select an alternative move.");

}

else

{

int PointsForPieceCapture = CalculateSahmMove(StartSquareReference);

CurrentPlayer.SetSahmUsed();

CurrentPlayer.UpdateQueueAfterMove(Choice);

UpdatePlayerScore(PointsForPieceCapture);

}

}

else

{

int FinishSquareReference = 0;

SquareIsValid = false;

while (!SquareIsValid)

{

FinishSquareReference = GetSquareReference("to move to");

SquareIsValid = CheckSquareIsValid(FinishSquareReference, false);

}

bool MoveLegal = CurrentPlayer.CheckPlayerMove(Choice, StartSquareReference, FinishSquareReference);

if (MoveLegal)

{

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

CurrentPlayer.UpdateQueueAfterMove(Choice);

UpdateBoard(StartSquareReference, FinishSquareReference);

UpdatePlayerScore(PointsForPieceCapture);

Console.WriteLine("New score: " + CurrentPlayer.GetScore() + Environment.NewLine);

}

}

//END CHANGE

Code for CalculateSahmMove:

//CHANGE

private int CalculateSahmMove(int StartSquareReference) //Q9

{

Console.WriteLine("Sahm Deployed!");

int StartRow = StartSquareReference / 10;

int StartCol = StartSquareReference % 10;

int SahmPointsCollector = 0;

int RowCounter = StartRow; ;

int Rowdifference = StartRow;

if (CurrentPlayer.GetDirection() == 1)

{

Rowdifference = 6 - StartRow;

}

for (int i = 0; i < Rowdifference; i++)

{

int TargetSquareReference = int.Parse(RowCounter.ToString() + StartCol.ToString());

if (Board[GetIndexOfSquare(TargetSquareReference)].GetPieceInSquare() != null && !Board[GetIndexOfSquare(TargetSquareReference)].ContainsKotla())

{

if (!CurrentPlayer.SameAs(Board[GetIndexOfSquare(TargetSquareReference)].  
GetPieceInSquare().GetBelongsTo()))

{

RaaketPointsCollector += CalculatePieceCapturePoints(TargetSquareReference);

Board[GetIndexOfSquare(TargetSquareReference)].RemovePiece();

}

}

if (CurrentPlayer.GetDirection() == 1)

{

RowCounter += 1;

}

else

{

RowCounter -= 1;

}

}

Console.WriteLine("Points Collected from Sahm: " + Convert.ToString(SahmPointsCollector));

return SahmPointsCollector;

}

//END CHANGE

Code for SahmMoveSelected:

//CHANGE

public bool SahmMoveSelected(int Choice) //Q9

{

if (Choice != 9)

{

if (Queue[Choice - 1].GetName() == "sahm")

{

return true;

}

else

{

return false;

}

}

return false;

}

//END CHANGE

Changes to Player:

class Player

{

private string Name;

private int Direction, Score;

private MoveOptionQueue Queue = new MoveOptionQueue();

//CHANGE

private bool SahmUsed = false; //Q9

//CHANGE

public Player(string N, int D)

{

Score = 100;

Name = N;

Direction = D;

}

//CHANGE

public void SetSahmUsed() //Q9

{

SahmUsed = true;

}

public bool GetSahmUsed() //Q9

{

return SahmUsed;

}

public bool ChoiceIsSahm(int Choice) //Q9

{

if (Queue.SahmMoveSelected(Choice))

{

return true;

}

else

{

return false;

}

}

//END CHANGE

## Testing:

* Showing the board correctly after the Sahm has been fired (allow follow-through for pieces in the Kotla destroyed). The pieces on 23 and 33 must have been destroyed to award the mark. [1 mark]

Text

Description automatically generated

Text

Description automatically generated

# Task 10 (5 marks)

## Coding

* Adding 7 to the menu to create a Kotla. **[1 mark]**
* Checking that the square in which the player wishes to create the Kotla is empty. **[1 mark]**
* Creating a Kotla of the correct type in the square and removing the piece from that square. [1 mark]
* Ensuring that the players turn ends after choosing 7 (dealt with in the example code by the use of the PieceSacrificed variable). **[1 mark]**

### Example Solution

Changes to PlayGame:

public void PlayGame()

{

bool GameOver = false;

while (!GameOver)

{

DisplayState();

bool SquareIsValid = false;

int Choice;

//CHANGE

int TargetSquareReference = 0; //Q10

bool PieceSacrificed = false;

do

{

Console.Write("Choose move option to use from queue (1 to 3) or 9 to take the offer or 7 to replace a piece with a new Kotla: "); //Q9

Choice = Convert.ToInt32(Console.ReadLine());

if (Choice == 7)

{

Console.WriteLine("Select one of your players to make the ultimate sacrifice and be converted to a Kotla");

while (!SquareIsValid)

{

TargetSquareReference = GetSquareReference("containing the piece to replace with a new Kotla");

SquareIsValid = CheckSquareIsValid(TargetSquareReference, true);

if (Board[GetIndexOfSquare(TargetSquareReference)].GetPieceInSquare() == null || Board[GetIndexOfSquare(TargetSquareReference)].ContainsKotla())

{

Console.WriteLine("That is not a valid selection. Please try again.");

SquareIsValid = false;

}

}

Board[GetIndexOfSquare(TargetSquareReference)].RemovePiece();

Square S;

if (CurrentPlayer.GetName() == "Player One")

{

S = new Kotla(CurrentPlayer, "K");

}

else

{

S = new Kotla(CurrentPlayer, "k");

}

Board[GetIndexOfSquare(TargetSquareReference)] = S;

SquareIsValid = false;

PieceSacrificed = true;

}

else if (Choice == 9)

{

UseMoveOptionOffer();

DisplayState();

}

}

while (Choice < 1 || Choice > 3 && !PieceSacrificed);

if (!PieceSacrificed)

{

int StartSquareReference = 0;

while (!SquareIsValid)

{

StartSquareReference = GetSquareReference("containing the piece to move");

SquareIsValid = CheckSquareIsValid(StartSquareReference, true);

}

int FinishSquareReference = 0;

SquareIsValid = false;

while (!SquareIsValid)

{

FinishSquareReference = GetSquareReference("to move to");

SquareIsValid = CheckSquareIsValid(FinishSquareReference, false);

}

bool MoveLegal = CurrentPlayer.CheckPlayerMove(Choice, StartSquareReference, FinishSquareReference);

if (MoveLegal)

{

int PointsForPieceCapture = CalculatePieceCapturePoints(FinishSquareReference);

CurrentPlayer.ChangeScore(-(Choice + (2 \* (Choice - 1))));

CurrentPlayer.UpdateQueueAfterMove(Choice);

UpdateBoard(StartSquareReference, FinishSquareReference);

UpdatePlayerScore(PointsForPieceCapture);

Console.WriteLine("New score: " + CurrentPlayer.GetScore() + Environment.NewLine);

}

}

//END CHANGE

## Testing:

* Showing the creation of the new Kotla (even if the letter is wrong) and removal of the sacrificed piece. **[1 mark]**

Text

Description automatically generated