# Interactions within the Pairs Game

This description is given from the point of view of the Player process.

# Process Network Diagram

The network diagram does not show all the channels between the Player Manager and Player Interface processes as this would tend to over complicate the diagram. The use of these other channels can be deduced from the interface components they interact with. The channels to and from the Controller process are created dynamically within the Player process as net channels.

Player Interface

Player Manager

Matcher

Mouse Buffer

getPoint

sendPoint

getValidPoint

validPoint

mouseEvent

Interface Channels

Controller

from

to

# Channel Interaction Sequence

.Controller Player Matcher Mouse Player  
 Manager Buffer Interface  
Initialise  
 EnrolPlayer  
 EnrolDetails  
   
At any time after initialised  
 MouseEvent  
  
 ‘Withdraw from Game’ button pressed  
. while enroled  
 GetGameDetails  
 GameDetails  
 change display, player names and pairs won  
 while notMatched and only one square chosen and enrolled  
 GetValidPoint  
 signal on getPoint  
 MousePoint  
 SquareCoords  
 change display to highlight chosen square  
 if not matched  
 highlight SELECT NEXT PAIR button  
 SELECT NEXT BUTTON pressed  
 change display to grey out chosen squares  
 else matched  
 ClaimPair

# Initialisation

Initially a connection is made to the game controller as follows:

The user interface contains a text enter field that allows the player to enter their name {111,112} followed by the IP address of the game controller {114-116}.

The node for the Player process is created using port 4000 {120-121}

The Game Controller listens on port 3000 and thus a net channel, called toController is created that enables this player to write to that channel {122-123}. A net channel is then created, called fromController that allows the Controller Process to write to this Player process {124-125}.

The Player process then attempts to enrol on the game by sending an EnrolPlayer record to the Controller that contains the player name and the location of the channel it has just created {127-131}.

The Player process then reads the enrol details from the Controller {132}. The Controller will not allow more than a fixed number of players to join a game, so that they all get a good game experience. Thus a player may be refused access {136-140}.

Assuming the player has been enrolled on the game then the main loop of the application commences.

# Main Processing Loop

Within the main loop two Alternatives are used {104 & 105}, outerAlt and innerAlt.

The main loop {146} is controlled by the value of enroled, which is only set false when the player presses the “Withdraw from game” button. This can be pressed at any time. The above alternatives both have the Withdraw from Game button as one of the guards to enable withdrawal from the game at any time.

At the start of each loop a blank board is created {148-149}. A request is sent to the game Controller {150} for the current state of the game, which is returned {151}. The game details are extracted {152-156} and used to update any change in the players playing the game {157-161}. The pairs that are still available are held in pairsMap and this is used to update the currently blank board {164-167} by the calls to the changePairs closure {74-88}.

An internal loop is now executed {170} to obtain the locations of two squares that hold available pairs. The mechanism begins with a request to the Matcher process to get a valid point. The outerAlt {174} is then used to either receive a valid point or for the player to press the withdraw button. Assuming a valid point is select a SquareCoords record is returned from the Matcher process. If this is the first point selected then the loop is repeated until a second valid point is obtained. The display is updated for each valid point to show the squares selected. If the outcome of the pairsMatch closure {90-102} is then obtained. If the matchOutcome has the value 2 this implies that the values (colour and numeric) held in the selected square did not match. The selected squares are returned to the grey colour and the loop repeated to select another pair of valid squares. This means that the Player Manager process does not have to check with the Controller Manager to see if the selected pair matches.

If the matchOutcome has the value 1, this implies that the squares match in both colour and value. A ClaimPair record is sent to the Controller Manager which checks to make sure the selected pair is still available. The selected pair may not be available because another player had already claimed them. The state of the Pairs Board is only updated after a player has selected a pair of squares that match.

The process now reapets the outer loop by obtaining the state of the game, which will have the effect of updating botht the board and the number of pairs each player has been able to claim.

# The Matcher and MouseBuffer Processes

This process waits to receive a GetValidPoint record, which contains the pairsMap and the parameters that govern the size of the board. The process then obtains mouse pressed points from the MouseBuffer process, checks to make sure the point is within a valid square that is known to hold a colour and value, which it then returns to the PlayerManager process for processing.

The MouseBuffer process simply holds the last mouse pressed event from the ActiveCanvas in the PlayerInterface process. When asked for a point it returns it to the Matcher process.

# The PairsMap Data Structure

The size of the board is predefined to be a square of size 10x10 squares. In the Controller Manager process, a closure createPairs {108-137}, is used to randomly generate the locations of a set of pairs. The number of pairs created is also random lying between minPairs and maxPairs. As the pairs are generated they are stored in the pairsMap as follows. The key is the [x, y] location of one member of the pair stored as a list. The map value is a list comprising the colour and numeric value associated with the pair. The map value will occur twice with different keys, representing the fact that the same colour and pair value occur twice at two different locations. The generation process ensures that two different colour pair combinations are not allocated to the same square.

As pairs are claimed by the players the number of unclaimed pairs is reduce. Once the number of unclaimed pairs reaches zero a new game is generated automatically with another randomly generated number of pairs. Only one player can claim the last pair in a game. Thus each game is given a unique identifier. Thus as the remaining players attempt to claim the last pair of the previous game they will not succeed, even though they think they can claim the last pair! The next time they request Game Details they will in fact receive the board for the next game that has just started.

# The Controller Manager Process

This process simply responds to the requests from the PlayerManager process. It is designed as a server. Every communication it receives apart from the ClaimPair record requires a response. All the PlayerManager processes communicate on the same fromPlayers {156} channel that is connected by default to port 3000. The process runs for ever and just reads objects from the fromPlayers channel. The action undertaken depends on the object type that has been read.

In particular the Controller Manager keeps record of the player names and the specific numeric identifier it has allocated in the playerNames list. Similar sized lists are used to hold the number of pairs each player has been able to win (pairsWon) and also the location of the net channel (toPlayers) by which the controller writes responses to each player.

# Listing

1. **package** turnOverGame\_v2
2. **import** org.jcsp.awt.\*
3. **import** org.jcsp.groovy.\*
4. **import** org.jcsp.lang.\*
5. **import** java.awt.\*
6. **import** java.awt.Color.\*
7. **import** org.jcsp.net2.\*;
8. **import** org.jcsp.net2.tcpip.\*;
9. **import** org.jcsp.net2.mobile.\*;
10. **import** java.awt.event.\*
11. **class** PlayerManager\_v2 **implements** CSProcess {
12. DisplayList dList
13. ChannelOutputList playerNames
14. ChannelOutputList pairsWon
15. ChannelOutput IPlabel
16. ChannelInput IPfield
17. ChannelOutput IPconfig
18. ChannelInput withdrawButton
19. ChannelInput nextButton
20. ChannelOutput getValidPoint
21. ChannelInput validPoint
22. ChannelOutput nextPairConfig
24. **int** side = 50
25. **int** minPairs = 5
26. **int** maxPairs = 10
28. **void** run(){
30. **int** gap = 5
31. **def** offset = [gap, gap]
32. **int** graphicsPos = (side / 2)
33. **def** rectSize = ((side+gap) \*10) + gap
34. GraphicsCommand[] display = **new** GraphicsCommand[504]
35. GraphicsCommand[] changeGraphics = **new** GraphicsCommand[5]
36. changeGraphics[0] = **new** GraphicsCommand.SetColor(Color.*WHITE*)
37. changeGraphics[1] = **new** GraphicsCommand.FillRect(0, 0, 0, 0)
38. changeGraphics[2] = **new** GraphicsCommand.SetColor(Color.*BLACK*)
39. changeGraphics[3] = **new** GraphicsCommand.DrawRect(0, 0, 0, 0)
40. changeGraphics[4] = **new** GraphicsCommand.DrawString(" ",graphicsPos,graphicsPos)
41. **def** createBoard = {
42. display[0] = **new** GraphicsCommand.SetColor(Color.*WHITE*)
43. display[1] = **new** GraphicsCommand.FillRect(0, 0, rectSize, rectSize)
44. display[2] = **new** GraphicsCommand.SetColor(Color.*BLACK*)
45. display[3] = **new** GraphicsCommand.DrawRect(0, 0, rectSize, rectSize)
46. **def** cg = 4
47. **for** ( x **in** 0..9){
48. **for** ( y **in** 0..9){
49. **def** **int** xPos = offset[0]+(gap\*x)+ (side\*x)
50. **def** **int** yPos = offset[1]+(gap\*y)+ (side\*y)
51. display[cg] = **new** GraphicsCommand.SetColor(Color.*WHITE*)
52. cg = cg+1
53. display[cg] = **new** GraphicsCommand.FillRect(xPos, yPos, side, side)
54. cg = cg+1
55. display[cg] = **new** GraphicsCommand.SetColor(Color.*BLACK*)
56. cg = cg+1
57. display[cg] = **new** GraphicsCommand.DrawRect(xPos, yPos, side, side)
58. cg = cg+1
59. xPos = xPos + graphicsPos
60. yPos = yPos + graphicsPos
61. display[cg] = **new** GraphicsCommand.DrawString(" ",xPos, yPos)
62. cg = cg+1
63. }
64. }
65. } // end createBoard
67. **def** pairLocations = []
68. **def** colours = [Color.*MAGENTA*, Color.*CYAN*, Color.*YELLOW*, Color.*PINK*]
70. **def** changePairs = {x, y, colour, p ->
71. **def** **int** xPos = offset[0]+(gap\*x)+ (side\*x)
72. **def** **int** yPos = offset[1]+(gap\*y)+ (side\*y)
73. changeGraphics[0] = **new** GraphicsCommand.SetColor(colour)
74. changeGraphics[1] = **new** GraphicsCommand.FillRect(xPos, yPos, side, side)
75. changeGraphics[2] = **new** GraphicsCommand.SetColor(Color.*BLACK*)
76. changeGraphics[3] = **new** GraphicsCommand.DrawRect(xPos, yPos, side, side)
77. xPos = xPos + graphicsPos
78. yPos = yPos + graphicsPos
79. **if** ( p >= 0)
80. changeGraphics[4] = **new** GraphicsCommand.DrawString(" " + p, xPos, yPos)
81. **else**
82. changeGraphics[4] = **new** GraphicsCommand.DrawString(" ??", xPos, yPos)
83. dList.change(changeGraphics, 4 + (x\*50) + (y\*5))
84. }
86. **def** pairsMatch = {pairsMap, cp ->
87. // cp is a list comprising two elements each of which is a list with the [x,y]
88. // location of a sqaure
89. // returns 0 if only one square has been chosen so far
90. // 1 if the two chosen squares have the same value (and colour)
91. // 2 if the chosen sqaures have different values
92. **if** (cp[1] == **null**) **return** 0
93. **else** {
94. **def** p1Data = pairsMap.get(cp[0])
95. **def** p2Data = pairsMap.get(cp[1])
96. **if** (p1Data[0] == p2Data[0]) **return** 1 **else** **return** 2
97. }
98. }
100. **def** outerAlt = **new** ALT([validPoint, withdrawButton])
101. **def** innerAlt = **new** ALT([nextButton, withdrawButton])
102. **def** NEXT = 0
103. **def** VALIDPOINT = 0
104. **def** WITHDRAW = 1
105. createBoard()
106. dList.set(display)
107. IPlabel.write("What is your name?")
108. **def** playerName = IPfield.read()
109. IPconfig.write(" ")
110. IPlabel.write("What is the IP address of the game controller?")
111. **def** controllerIP = IPfield.read().trim()
112. IPconfig.write(" ")
113. IPlabel.write("Connecting to the GameController")
115. // create Node and Net Channel Addresses
116. **def** nodeAddr = **new** TCPIPNodeAddress (4000)
117. Node.*getInstance*().init (nodeAddr)
118. **def** toControllerAddr = **new** TCPIPNodeAddress ( controllerIP, 3000)
119. **def** toController = NetChannel.*any2net*(toControllerAddr, 50 )
120. **def** fromController = NetChannel.*net2one*()
121. **def** fromControllerLoc = fromController.getLocation()
123. // connect to game controller
124. IPconfig.write("Now Connected - sending your name to Controller")
125. **def** enrolPlayer = **new** EnrolPlayer( name: playerName,
126. toPlayerChannelLocation: fromControllerLoc)
127. toController.write(enrolPlayer)
128. **def** enrolDetails = (EnrolDetails)fromController.read()
129. **def** myPlayerId = enrolDetails.id
130. **def** enroled = **true**
131. **def** unclaimedPairs = 0
132. **if** (myPlayerId == -1) {
133. enroled = **false**
134. IPlabel.write("Sorry " + playerName + ", there are too many players enroled in this PAIRS game")
135. IPconfig.write(" Please close the game window")
136. }
137. **else** {
138. IPlabel.write("Hi " + playerName + ", you are now enroled in the PAIRS game")
139. IPconfig.write(" ")
141. // main loop
142. **while** (enroled) {
143. **def** chosenPairs = [**null**, **null**]
144. createBoard()
145. dList.change (display, 0)
146. toController.write(**new** GetGameDetails(id: myPlayerId))
147. **def** gameDetails = (GameDetails)fromController.read()
148. **def** gameId = gameDetails.gameId
149. IPconfig.write("Playing Game Number - " + gameId)
150. **def** playerMap = gameDetails.playerDetails
151. **def** pairsMap = gameDetails.pairsSpecification
152. **def** playerIds = playerMap.keySet()
153. playerIds.*each* { p ->
154. **def** pData = playerMap.get(p)
155. playerNames[p].write(pData[0])
156. pairsWon[p].write(" " + pData[1])
157. }
159. // now use pairsMap to create the board
160. **def** pairLocs = pairsMap.keySet()
161. pairLocs.*each* {loc ->
162. changePairs(loc[0], loc[1], Color.*LIGHT\_GRAY*, -1)
163. }
164. **def** currentPair = 0
165. **def** notMatched = **true**
166. **while** ((chosenPairs[1] == **null**) && (enroled) && (notMatched)) {
167. getValidPoint.write (**new** GetValidPoint( side: side,
168. gap: gap,
169. pairsMap: pairsMap))
170. **switch** ( outerAlt.select() ) {
171. **case** WITHDRAW:
172. withdrawButton.read()
173. toController.write(**new** WithdrawFromGame(id: myPlayerId))
174. enroled = **false**
175. **break**
176. **case** VALIDPOINT:
177. **def** vPoint = ((SquareCoords)validPoint.read()).location
178. chosenPairs[currentPair] = vPoint
179. currentPair = currentPair + 1
180. **def** pairData = pairsMap.get(vPoint)
181. changePairs(vPoint[0], vPoint[1], pairData[1], pairData[0])
182. **def** matchOutcome = pairsMatch(pairsMap, chosenPairs)
183. **if** ( matchOutcome == 2) {
184. nextPairConfig.write("SELECT NEXT PAIR")
185. **switch** (innerAlt.select()){
186. **case** NEXT:
187. nextButton.read()
188. nextPairConfig.write(" ")
189. **def** p1 = chosenPairs[0]
190. **def** p2 = chosenPairs[1]
191. changePairs(p1[0], p1[1], Color.*LIGHT\_GRAY*, -1)
192. changePairs(p2[0], p2[1], Color.*LIGHT\_GRAY*, -1)
193. chosenPairs = [**null**, **null**]
194. currentPair = 0
195. **break**
196. **case** WITHDRAW:
197. withdrawButton.read()
198. toController.write(**new** WithdrawFromGame(id: myPlayerId))
199. enroled = **false**
200. **break**
201. } // end inner switch
202. } **else** **if** ( matchOutcome == 1) {
203. notMatched = **false**
204. toController.write(**new** ClaimPair ( id: myPlayerId,
205. gameId: gameId,
206. p1: chosenPairs[0],
207. p2: chosenPairs[1]))
208. }
209. **break**
210. }// end of outer switch
211. } // end of while getting two pairs
212. } // end of while enrolled loop
213. IPlabel.write("Goodbye " + playerName + ", please close game window")
214. } //end of enrolling test
215. } // end run
216. }