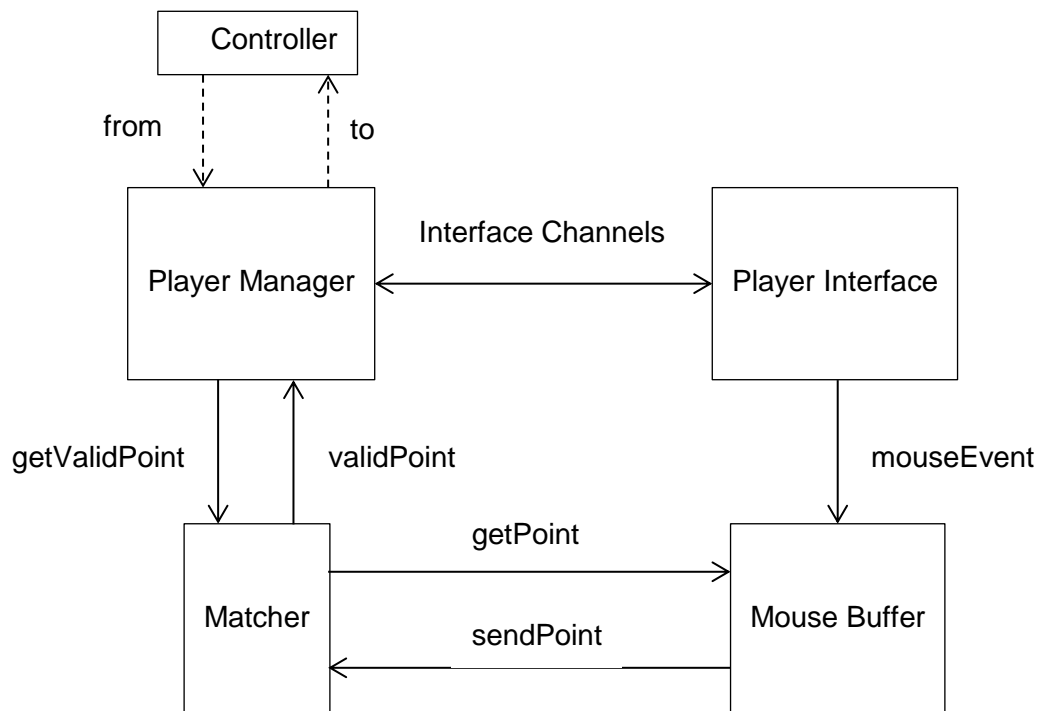


Interactions within the Pairs Game

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.



```
sequenceDiagram
    participant Controller
    participant PlayerManager
    participant Matcher
    participant MouseBuffer
    participant PlayerInterface

    Controller->>PlayerManager: EnrolPlayer
    PlayerManager->>Controller: EnrolDetails
    Note over Controller: Initialised

    Note over PlayerInterface: At any time after initialised
    PlayerInterface->>MouseBuffer: MouseEvent
    MouseBuffer->>PlayerManager: 'Withdraw from Game' button pressed

    Note over Controller: while enrolled
    Controller->>PlayerManager: GetGameDetails
    PlayerManager->>Controller: GameDetails
    PlayerManager->>PlayerInterface: change display, player names and pairs won

    Note over PlayerManager: while notMatched and only one square chosen and enrolled
    PlayerManager->>Matcher: GetValidPoint
    Matcher->>MouseBuffer: signal on getPoint
    MouseBuffer->>Matcher: MousePoint
    Matcher->>PlayerManager: SquareCoords
    PlayerManager->>PlayerInterface: change display to highlight chosen square

    Note over PlayerManager: if not matched
    PlayerManager->>PlayerInterface: highlight SELECT NEXT PAIR button
    PlayerInterface->>PlayerManager: SELECT NEXT BUTTON pressed
    PlayerManager->>PlayerInterface: change display to grey out chosen squares

    Note over Controller: else matched
    PlayerManager->>Controller: ClaimPair
    Controller->>PlayerManager: ClaimPair
```

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

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 repeats the outer loop by obtaining the state of the game, which will have the effect of updating both 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 reduced. 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
3 import org.jcsp.awt.*
4 import org.jcsp.groovy.*
5 import org.jcsp.lang.*
6 import java.awt.*
7 import java.awt.Color.*
8 import org.jcsp.net2.*;
9 import org.jcsp.net2.tcpiip.*;
10 import org.jcsp.net2.mobile.*;
11 import java.awt.event.*
12
13 class PlayerManager_v2 implements CSProcess {
14     DisplayList dList
15     ChannelOutputList playerNames
16     ChannelOutputList pairsWon
17     ChannelOutput ILabel
18     ChannelInput IPfield
19     ChannelOutput IPconfig
20     ChannelInput withdrawButton
21     ChannelInput nextButton
22     ChannelOutput getValidPoint
23     ChannelInput validPoint
24     ChannelOutput nextPairConfig
25
26     int side = 50
27     int minPairs = 5
28     int maxPairs = 10
29
30     void run(){
31
32         int gap = 5
33         def offset = [gap, gap]
34         int graphicsPos = (side / 2)
35         def rectSize = ((side+gap) *10) + gap
36
37         GraphicsCommand[] display = new GraphicsCommand[504]
38         GraphicsCommand[] changeGraphics = new GraphicsCommand[5]
39         changeGraphics[0] = new GraphicsCommand.SetColor(Color.WHITE)
40         changeGraphics[1] = new GraphicsCommand.FillRect(0, 0, 0, 0)
41         changeGraphics[2] = new GraphicsCommand.SetColor(Color.BLACK)
42         changeGraphics[3] = new GraphicsCommand.DrawRect(0, 0, 0, 0)
43         changeGraphics[4] = new GraphicsCommand.DrawString(" ",graphicsPos,graphicsPos)
```

```

44
45 def createBoard = {
46     display[0] = new GraphicsCommand.SetColor(Color.WHITE)
47     display[1] = new GraphicsCommand.FillRect(0, 0, rectSize, rectSize)
48     display[2] = new GraphicsCommand.SetColor(Color.BLACK)
49     display[3] = new GraphicsCommand.DrawRect(0, 0, rectSize, rectSize)
50     def cg = 4
51     for ( x in 0..9){
52         for ( y in 0..9){
53             def int xPos = offset[0]+(gap*x)+ (side*x)
54             def int yPos = offset[1]+(gap*y)+ (side*y)
55             display[cg] = new GraphicsCommand.SetColor(Color.WHITE)
56             cg = cg+1
57             display[cg] = new GraphicsCommand.FillRect(xPos, yPos, side, side)
58             cg = cg+1
59             display[cg] = new GraphicsCommand.SetColor(Color.BLACK)
60             cg = cg+1
61             display[cg] = new GraphicsCommand.DrawRect(xPos, yPos, side, side)
62             cg = cg+1
63             xPos = xPos + graphicsPos
64             yPos = yPos + graphicsPos
65             display[cg] = new GraphicsCommand.DrawString("  ",xPos, yPos)
66             cg = cg+1
67         }
68     }
69 } // end createBoard
70
71 def pairLocations = []
72 def colours = [Color.MAGENTA, Color.CYAN, Color.YELLOW, Color.PINK]
73
74 def changePairs = {x, y, colour, p ->
75     def int xPos = offset[0]+(gap*x)+ (side*x)
76     def int yPos = offset[1]+(gap*y)+ (side*y)
77     changeGraphics[0] = new GraphicsCommand.SetColor(colour)
78     changeGraphics[1] = new GraphicsCommand.FillRect(xPos, yPos, side, side)
79     changeGraphics[2] = new GraphicsCommand.SetColor(Color.BLACK)
80     changeGraphics[3] = new GraphicsCommand.DrawRect(xPos, yPos, side, side)
81     xPos = xPos + graphicsPos
82     yPos = yPos + graphicsPos
83     if ( p >= 0)
84         changeGraphics[4] = new GraphicsCommand.DrawString("  " + p, xPos, yPos)
85     else
86         changeGraphics[4] = new GraphicsCommand.DrawString(" ??", xPos, yPos)
87     dList.change(changeGraphics, 4 + (x*50) + (y*5))

```

```

88     }
89
90     def pairsMatch = {pairsMap, cp ->
91         // cp is a list comprising two elements each of which is a list with the [x,y]
92         // location of a square
93         // returns 0 if only one square has been chosen so far
94         //         1 if the two chosen squares have the same value (and colour)
95         //         2 if the chosen squares have different values
96         if (cp[1] == null) return 0
97         else {
98             def p1Data = pairsMap.get(cp[0])
99             def p2Data = pairsMap.get(cp[1])
100             if (p1Data[0] == p2Data[0]) return 1 else return 2
101         }
102     }
103
104     def outerAlt = new ALT([validPoint, withdrawButton])
105     def innerAlt = new ALT([nextButton, withdrawButton])
106     def NEXT = 0
107     def VALIDPOINT = 0
108     def WITHDRAW = 1
109     createBoard()
110     dList.set(display)
111     ILabel.write("What is your name?")
112     def playerName = IPfield.read()
113     IPconfig.write(" ")
114     ILabel.write("What is the IP address of the game controller?")
115     def controllerIP = IPfield.read().trim()
116     IPconfig.write(" ")
117     ILabel.write("Connecting to the GameController")
118
119     // create Node and Net Channel Addresses
120     def nodeAddr = new TCIPNodeAddress (4000)
121     Node.getInstance().init (nodeAddr)
122     def toControllerAddr = new TCIPNodeAddress ( controllerIP, 3000)
123     def toController = NetChannel.any2net(toControllerAddr, 50 )
124     def fromController = NetChannel.net2one()
125     def fromControllerLoc = fromController.getLocation()
126

```

```

127 // connect to game controller
128 IPconfig.write("Now Connected - sending your name to Controller")
129 def enrolPlayer = new EnrolPlayer( name: playerName,
130                                     toPlayerChannelLocation: fromControllerLoc)
131 toController.write(enrolPlayer)
132 def enrolDetails = (EnrolDetails)fromController.read()
133 def myPlayerId = enrolDetails.id
134 def enroled = true
135 def unclaimedPairs = 0
136 if (myPlayerId == -1) {
137     enroled = false
138     ILabel.write("Sorry " + playerName + ", there are too many players enroled in this PAIRS game")
139     IPconfig.write(" Please close the game window")
140 }
141 else {
142     ILabel.write("Hi " + playerName + ", you are now enroled in the PAIRS game")
143     IPconfig.write(" ")
144
145     // main loop
146     while (enroled) {
147         def chosenPairs = [null, null]
148         createBoard()
149         dList.change (display, 0)
150         toController.write(new GetGameDetails(id: myPlayerId))
151         def gameDetails = (GameDetails)fromController.read()
152         def gameId = gameDetails.gameId
153         IPconfig.write("Playing Game Number - " + gameId)
154         def playerMap = gameDetails.playerDetails
155         def pairsMap = gameDetails.pairsSpecification
156         def playerIds = playerMap.keySet()
157         playerIds.each { p ->
158             def pData = playerMap.get(p)
159             playerNames[p].write(pData[0])
160             pairsWon[p].write(" " + pData[1])
161         }
162
163         // now use pairsMap to create the board
164         def pairLocs = pairsMap.keySet()
165         pairLocs.each {loc ->
166             changePairs(loc[0], loc[1], Color.LIGHT_GRAY, -1)
167         }

```



```

168 def currentPair = 0
169 def notMatched = true
170 while ((chosenPairs[1] == null) && (enroled) && (notMatched)) {
171     getValidPoint.write (new GetValidPoint( side: side,
172                                             gap: gap,
173                                             pairsMap: pairsMap))
174     switch ( outerAlt.select() ) {
175         case WITHDRAW:
176             withdrawButton.read()
177             toController.write(new WithdrawFromGame(id: myPlayerId))
178             enroled = false
179             break
180         case VALIDPOINT:
181             def vPoint = ((SquareCoords)validPoint.read()).location
182             chosenPairs[currentPair] = vPoint
183             currentPair = currentPair + 1
184             def pairData = pairsMap.get(vPoint)
185             changePairs(vPoint[0], vPoint[1], pairData[1], pairData[0])
186             def matchOutcome = pairsMatch(pairsMap, chosenPairs)
187             if ( matchOutcome == 2 ) {
188                 nextPairConfig.write("SELECT NEXT PAIR")
189                 switch (innerAlt.select()){
190                     case NEXT:
191                         nextButton.read()
192                         nextPairConfig.write(" ")
193                         def p1 = chosenPairs[0]
194                         def p2 = chosenPairs[1]
195                         changePairs(p1[0], p1[1], Color.LIGHT_GRAY, -1)
196                         changePairs(p2[0], p2[1], Color.LIGHT_GRAY, -1)
197                         chosenPairs = [null, null]
198                         currentPair = 0
199                         break
200                     case WITHDRAW:
201                         withdrawButton.read()
202                         toController.write(new WithdrawFromGame(id: myPlayerId))
203                         enroled = false
204                         break
205                 } // end inner switch

```

```

206         } else if ( matchOutcome == 1) {
207             notMatched = false
208             toController.write(new ClaimPair ( id: myPlayerId,
209                                     gameId: gameId,
210                                     p1: chosenPairs[0],
211                                     p2: chosenPairs[1]))
212         }
213         break
214     } // end of outer switch
215 } // end of while getting two pairs
216 } // end of while enrolled loop
217 ILabel.write("Goodbye " + playerName + ", please close game window")
218 } //end of enrolling test
219 } // end run
220 }

```