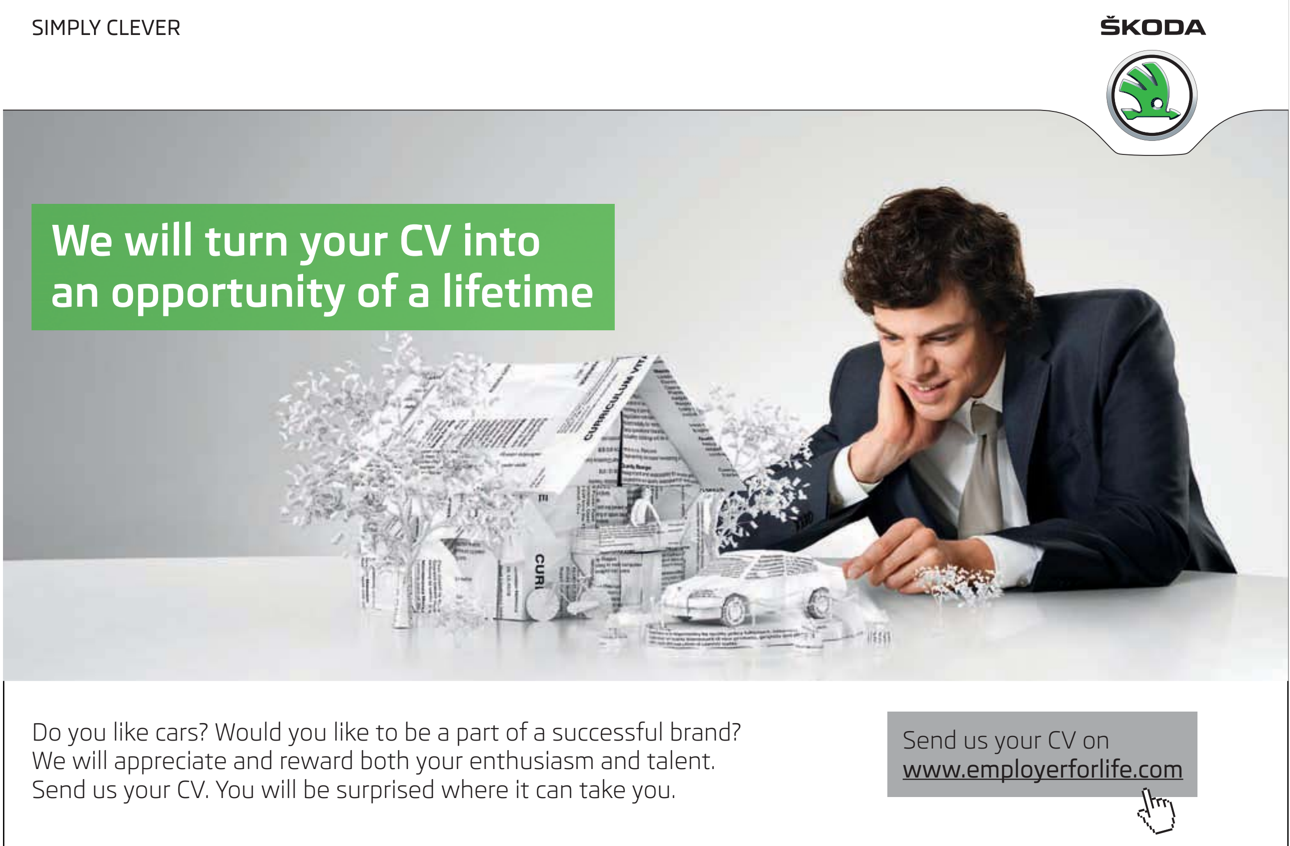
13 Accessing Shared Resources: CREW

Shared resource management is commonly used to access large amounts of data by many users;

- the shared resource concept is defined
- concurrent read exclusive write is explained
- a simple example is developed demonstrating the concept

This chapter describes techniques that were developed for, and are used most often in shared memory multi-processing systems. In such systems great care has to be taken to ensure that processes running on the same processor do not access an area of shared memory in an uncontrolled manner. Up to now the solutions have simply ignored this problem because all data has been local to and encapsulated within a process. One process has communicated data to another as required by the needs of the solution. The process and channel mechanisms have implicitly provided two capabilities, namely synchronisation between processes and mutual exclusion of data areas. In shared memory environments the programmer has to be fully aware of both these aspects to ensure that neither is violated.



Click on the ad to read more

Mutual exclusion ensures that while one process is accessing a piece of shared data no other process will be allowed access regardless of the interleaving of the processes on the processor. Synchronisation ensures that processes gain access to such shared data areas in a manner that enables them to undertake useful work. The simplest solution to both these problems is to use a pattern named CREW, Concurrent Read Exclusive Write, which, as its names suggests, allows any number of reader processes to access a piece of shared data at the same time but only one writer process to access the same piece of data at one time. The CREW mechanism manages this requirement and in sensible implementations also imposes some concept of fairness. If access is by multiple instances of reader and writer processes then one could envisage a situation where the readers could exclude writers and vice versa and this should be ameliorated as far as is possible. The JCSP implementation of a CREW does exhibit this capability of fairness, as shall be demonstrated.

At the simplest level the CREW has to be able to protect accesses to the shared data and the easiest way of doing this is to surround each access, be it a read or write with a call to a method that allows the start of an operation and subsequently when the operation is finished to indicate that it has ended. Between such pairs of method calls the operation of the CREW is guaranteed. Thus the programmer has to surround access to shared data with the required start and end method calls be they a read or write to the shared data. It is up to the programmer to ensure that all such accesses to the shared data are suitably protected.

In the JCSP implementation of CREW we extend an existing storage collection with a `Crew` class. Then we ensure that each access that `put`s data into the collection is surrounded by a `startWrite`() and `endWrite`() pair of method calls on the `Crew`. Similarly, that each `get` access is surrounded by a `startRead`() and `endRead`() method call. Internally, the `Crew` then ensures that access to the shared storage collection is undertaken in accordance with the required behaviour. Further, fairness can be implemented quite simply by ensuring that if the shared data is currently being accessed by one or more reader processes then as soon as a writer process indicates that it wishes to put some data into the shared collection then no further reader processes are permitted to start reading until the write has finished. Similarly, a sequence of write processes, each of which requires exclusive access, will be interposed by reader process accesses as necessary.

13.1 CrewMap

Listing 13-1 shows a simple extension of a `HashMap` {10} by means of an instance of a `Crew` {12}. The `put` and `get` methods of `HashMap` are then overwritten with new versions that surround them with the appropriate start and end method calls {15, 17} and {21, 24}, between which the normal `HashMap`'s `get` and `put` methods can be called as usual.

Download free eBooks at bookboon.com

```
10 class CrewMap extends HashMap<Object, Object> {
11
12   def theCrew = new Crew()
13
14   def Object put ( Object itsKey, Object itsValue ) {
15     theCrew.startWrite()
16     super.put ( itsKey, itsValue )
17     theCrew.endWrite()
18   }
19
20   def Object get ( Object itsKey ) {
21     theCrew.startRead()
22     def result = super.get ( itsKey )
23     theCrew.endRead()
24     return result
25   }
26
27 }
```

Listing 13-1 The CrewMap Class Definition

At this point a word of caution has to be given. This arises because Java allows exceptions to be thrown at any point. Thus in the above formulation it might be possible for the lines that represent normal access to the shared resource {16, 22} to fail. In such a case the call to the `end` synchronisation method {17, 23} will never happen and thus the `Crew` will fail in due course as the required locks will not be released. The associated documentation for JCSP Crew discusses this in more detail. The solution is to encapsulate the access in a `try .. catch .. finally` block. The problem arises because Java invokes code sequences that are not part of the coding sequence and thus the programmer has to be very wary of these possibilities. In the following description we shall presume that all access is well behaved and such a fault will not occur.

Once the `CrewMap` has been defined it can be used in a solution that requires multiple processes access to its shared data collection. Figure 13-1 shows such a typical application. In this case two `Read` and two `Write` processes access the shared `DataBase` resource. The coding of the `DataBase` process is shown in Listing 13-2.

Download free eBooks at bookboon.com

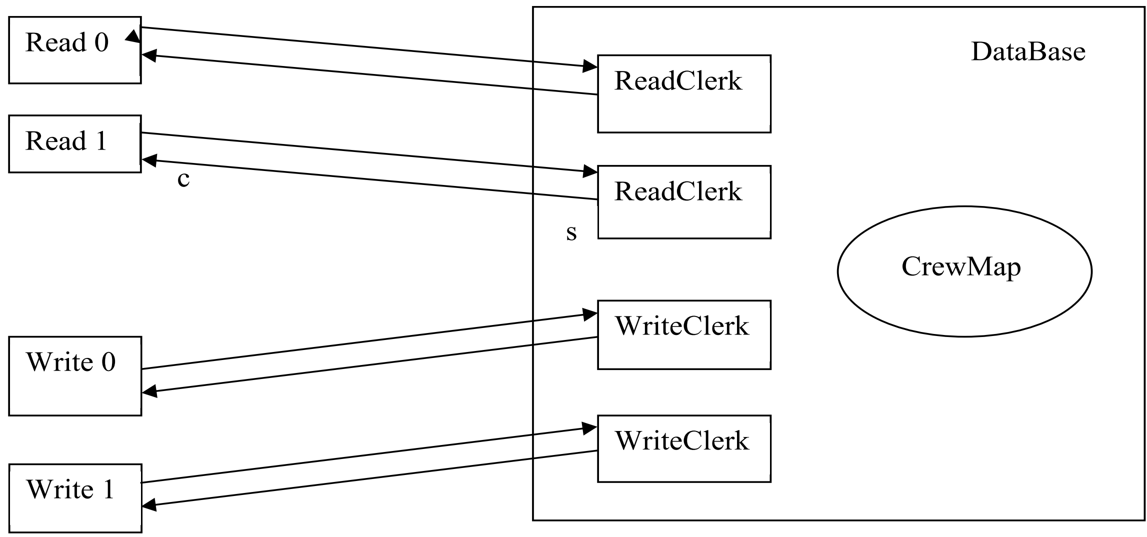


Figure 13-1 A Simple Use of CrewMap

13.2 The DataBase Process

The `DataBase` process has two channel list properties {12, 13} comprising the channels used by the `Read` and `Write` processes to access it. Additionally, properties are required that define the number of such `Read` and `Write` processes, `readers` and `writers` respectively {14, 15}.

Download free eBooks at bookboon.com

```
10 class DataBase implements CSProcess {
11
12    def ChannelInputList inChannels
13    def ChannelOutputList outChannels
14    def int readers
15    def int writers
16
17    void run () {
18      println "DataBase has started"
19      def crewDataBase = new CrewMap()
20      for ( i in 0 ..< 10 ) {
21        crewDataBase.put ( i, 100 + i)
22      }
23      for ( i in 0 ..< 10 ) {
24          println "DB: Location $i contains ${crewDataBase.get( i )} "
25      }
26      def processList = []
27      for (i in 0..< readers) {
28          processList.putAt (i, new ReadClerk ( cin: inChannels[i],
29                                                cout: outChannels[i],
30                                                data: crewDataBase) )
31      }
32      for ( i in 0 ..< writers ) {
33          processList.putAt ( ( i + readers), new WriteClerk (cin:
            inChannels[i + readers],
34                                      cout: outChannels[i + readers],
35                                      data: crewDataBase ) )
36      }
37      new PAR (processList).run()
38    }
39 }
```

Listing 13-2 The DataBase Process definition

The `run` method {17} essentially creates the structure shown in Figure 13-1. An instance of `CrewMap` is defined called `crewDataBase` {19}. The shared resource `crewDataBase` is then populated with initial values {20–22}, which initialises the first ten locations with the values 100 to 109 in sequence. An empty `processList` {26} is then defined that will hold instances of the required `ReadClerk` and `WriteClerk` processes. The required number of `ReadClerk` processes are then created {27–36} and placed in `processList`. Each `ReadClerk` is allocated the corresponding element of the `inChannels` and `outChannels` channel lists {28, 29}. Finally, the `ReadClerk` process has its `data` property initialised to the `crewDataBase` itself {30}. The `WriteClerk` processes are instantiated in the same manner {32–36} ensuring that the correct elements of the `inChannels` and `outChannels` lists are allocated to the processes. This means that the all the `ReadClerk` and `WriteClerk` processes have shared access to the `crewDataBase`. The `processList` can now be passed to a `PAR` for `run`ning {37}.

Download free eBooks at bookboon.com

Communication between the Read and Write processes and the DataBase is achieved by a single class called DataObject {10}, see Listing 13-3. DataObject comprises three properties {12–14}, pid hold the identity number of the accessing Read or Write process, location holds the index of the resource element to be accessed and value is either the value read from that element or that is to be written to the element.

```
10 class DataObject implements Serializable, JCSPCopy {
11
12    def int pid
13    def int location
14    def int value
15 }
```

Listing 13-3 The Definition of DataObject (Omitting Methods copy and toString)

It should be noted that this formulation of the DataBase contains no alternative (ALT) as might be expected from previous examples. This arises because we are using a formulation that contains a CREW that essentially provides the same functionality, but only for shared memory applications. The advantage of the alternative is that it can be used to alternate over networked channels and thus is more flexible. It also has the advantage of exposing the alternative concept that is so important in the modelling of parallel systems.

13.3 The Read Clerk Process

Listing 13-4 shows the ReadClerk process, which has channel input and output properties cin {12} and cout {13} respectively and a data property {14} that accesses the CREW resource.

```
10 class ReadClerk implements CSProcess {
11
12    def ChannelInput cin
13    def ChannelOutput cout
14    def CrewMap data
15
16    void run () {
17      println "ReadClerk has started "
18      while (true) {
19        def d = new DataObject()
20        d = cin.read()
21        d.value = data.get ( d.location )
22        println "RC: Reader ${d.pid} has read ${d.value} from ${d.location}"
23        cout.write(d)
24      }
25    }
26 }
```

Listing 13-4 The ReadClerk Process

Download free eBooks at bookboon.com

The `run` method {16-25} defines an instance `d` of type `DataObject` {19} after which the value of `d` is read from `cin` {20}. The `location` property of `d` is then used to access the `CrewMap` property `data` {21} to `get` the corresponding value which is then stored in the `value` property of `d`. The revised value of `d` is then written to the channel `cout` {23}, after an appropriate message is printed.

13.4 The Write Clerk Process

The `WriteClerk` process is shown in Listing 13-5 and is fundamentally the same as that shown in the `ReadClerk` process except that a new `value` is `put` into the shared resource {21}. The unmodified `DataObject d` is written back to the corresponding `Write` process to confirm that the operation has taken place {23}.

```
10 class WriteClerk implements CSProcess {
11
12   def ChannelInput cin
13   def ChannelOutput cout
14   def CrewMap data
15
16   void run () {
17     println "WriteClerk has started "
18     while (true) {
19       def d = new DataObject()
20       d = cin.read()
```

Download free eBooks at bookboon.com

```
21        data.put ( d.location, d.value )
22        println "WC: Writer ${d.pid} has written ${d.value} to ${d.location}"
23        cout.write(d)
24      }
25    }
26 }
```

Listing 13-5 The WriteClerk Process

Each of the Clerk processes behaves as a pure server. The server behaviour is guaranteed provided access to the shared data resource always complete in finite time. This will happen provided no exception is thrown and handled incorrectly in the shared data resource.

13.5 The Read Process

The `Read` process is shown in Listing 13-6. It has three properties. A channel by which it writes to the database `r2db` {12} and one by which it reads returned values `db2r` {13}. The last property, `id` {14}, is the identity number of the `Read` process. The channel `toConsole` {15} writes messages to an associated `GConsole` process. The `run` method {17} initialises a `DataObject` with the `Read` process' `id` {22} and then reads a value from each location of the shared resource in sequence {20}, printing out each returned value {25}. This is achieved by allocating the loop value `i` to the `location` property of `d` {22}. The instance `d` is then written to the shared resource using the channel `r2db` {23}. The process then waits until it can read the returned `DataObject` into `d` using the channel `db2r` {24}. This means that the process behaves as a pure client.

```
10 class Read implements CSProcess {
11
12   def ChannelOutput r2db
13   def ChannelInput db2r
14   def int id
15   def ChannelOutput toConsole
16
17   void run () {
18     def timer = new CSTimer()
19     toConsole.write ( "Reader $id has started \n")
20     for ( i in 0 ..<10 ) {
21       def d = new DataObject(pid:id)
22       d.location = i
23       r2db.write(d)
24       d = db2r.read()
25       toConsole.write ( "Location "d.location+" has value "+d.value + "\n")
26       timer.sleep(100)
27     }
28     toConsole.write ( "Reader $id has finished \n")
29   }
30 }
```

Listing 13-6 The Read Process

Download free eBooks at bookboon.com

13.6 The Write Process

The `Write` process is shown in Listing 13-7 and is very similar to the `Read` process except that the elements of the shared resource are accessed in reverse order, that is from 9 to 0 {22}. The value written to the shared resource is dependent upon the `id` of the writing process {24} and is sufficiently different to make observation of the resulting behaviour easier.

```
10 class Write implements CSProcess {
11
12   def ChannelOutput w2db
13   def ChannelInput db2w
14   def int id
15   def ChannelOutput toConsole
16
17   void run () {
18     def timer = new CSTimer()
19     toConsole.write ( "Writer $id has started \n" )
20     for ( j in 0 ..<10 ) {
21       def d = new DataObject(pid:id)
22       def i = 9 - j // write in reverse order
23       d.location = i
24       d.value = i + ((id+1)*1000)
25       w2db.write(d)
26       d = db2w.read()
27       toConsole.write ("Location "+d.location+" now contains "+d.value+"\n")
28       timer.sleep(100)
29     }
30     toConsole.write ( "Writer $id has finished \n" )
31   }
32 }
```

Listing 13-7 The Write Process

13.7 Creating the System

The script that invokes the `DataBase` system is shown in Listing 13-8.

```
10 def nReaders = Ask.Int ( "Number of Readers ? ", 1, 5)
11 def nWriters = Ask.Int ( "Number of Writers ? ", 1, 5)
12 def connections = nReaders + nWriters
13
14 def toDatabase = Channel.one2oneArray(connections)
15 def fromDatabase = Channel.one2oneArray(connections)
16 def consoleData = Channel.one2oneArray(connections)
17
18 def toDB = new ChannelInputList(toDatabase)
19 def fromDB = new ChannelOutputList(fromDatabase)
20
```

Download free eBooks at bookboon.com

```
21 def readers = ( 0 ..< nReaders).collect { r ->
22                         return new Read (id: r,
23                                          r2db: toDatabase[r].out(),
24                                          db2r: fromDatabase[r].in(),
25                                          toConsole: consoleData[r].out())
26                         }
27
28 def writers = ( 0 ..<nWriters).collect { w ->
29                         int wNo = w + nReaders
30                         return new Write ( id: w,
31                                          w2db: toDatabase[wNo].out(),
32                                          db2w: fromDatabase[wNo].in(),
33                                          toConsole: consoleData[wNo].
                                          out())
34                         }
35
36 def database = new DataBase ( inChannels: toDB,
37                               outChannels: fromDB,
38                               readers: nReaders,
39                               writers: nWriters)
40
41 def consoles = ( 0 ..< connections).collect { c ->
42                         def frameString = c < nReaders ?
43                                   "Reader " + c :
44                                   "Writer " + (c - nReaders)
45                         return new GConsole (toConsole: consoleData[c].in(),
46                                          frameLabel: frameString )
47                         }
48 def procList = readers + writers + database + consoles
49
50 new PAR(procList).run()
```

Listing 13-8 The Script to Invoke the DataBase System

Initially, the number of Read and Write processes is obtained {10, 11} by a console interaction. The total number of connections to the DataBase is then calculated as connections {12}. The system uses a GConsole process for each Read and Write process to display the outcome of the interactions with the DataBase. The channels used to connect the Read and Write processes to the Database and the GConsoles are then defined {14–16}. The corresponding channel lists toDb and fromDb are then defined {18, 19}, which connect the Read and Write processes to the DataBase.

The required number of Read processes is then created in the list readers {21–26}. Each instance uses the closure property r to identify the required element of the previously declared channel arrays that connect the process to the DataBase and its GConsole process. Similarly, the required number of Write processes is defined {28–34}. The variable wNo {29} is used to ensure that the index used to associate Write process channel indeces is offset by the number of Read processes.

Download free eBooks at bookboon.com

An instance of the DataBase process is then created {36–39}, using the previously declared channel lists. The list consoles {41–47} contains the instances of GConsole required to connect to the Read and Write processes. Finally, procList is created as the addition of all the process lists and the database process {48} and then run {50}.

Outputs 13-1 and 13-2 show the output from the running of the system when it is started with two Read and two Write processes. The order in which the Write process have been executed can be determined from the values that have been read by the two Read processes. Recall that the Write processes access the database locations in reverse order to the Read processes. The outputs indicate that the implementation of the Crew class is inherently fair because the values read by the Read processes change from the initial values to the modified values about half way through the cycle. The values read from locations 5 and 6 also vary indicating that state of the DataBase was in flux at that point in the access cycles with read and write operations fully interleaved.

Download free eBooks at bookboon.com
Click on the ad to read more

```
Location 1 has value 101
Location 2 has value 102
Location 3 has value 103
Location 4 has value 104
Location 5 has value 105
Location 6 has value 1006
Location 7 has value 2007
Location 8 has value 2008
Location 9 has value 2009
Reader has finished
```

Output 13 – 1 Output From Read process 0

```
Location 1 has value 101
Location 2 has value 102
Location 3 has value 103
Location 4 has value 104
Location 5 has value 1005
Location 6 has value 2006
Location 7 has value 2007
Location 8 has value 2008
Location 9 has value 2009
Reader has finished
```

Output 13 – 2 Output From Read process 1

13.8 Summary

In this chapter we have investigated a typical mechanism used in shared memory multi-processing system. The formulation tends to hide the interactions that take place because these are captured somewhat remotely in the `CrewMap` class definition.

13.9 Challenge

Rewrite the system so that a Crew is not used and the DataBase process alternates over the input channels from the Read and Write processes. The system should capture the same concept of fairness as exhibited in the CREW based solution.

Download free eBooks at bookboon.com

14 Barriers and Buckets: Hand-Eye Co-ordination Test

This chapter develops a solution to a highly dynamic system using a number of shared meory synchronisation capabilities including:

- barrier
- alting barrier
- bucket
- channel data stores are used to overcome inconsistenices in the underlying Java user interface model

Three shared memory synchronisation techniques are combined to provide control of a highly dynamic environment. A `Barrier` provides a means whereby a known number of processes collectively control their operation so they all wait at the barrier until all of them have synchronised with the barrier at which time they are all released to run in parallel. An `AltingBarrier` is a specialisation of the `Barrier` that allows it to act also as a guard in an Alternative (Welch, et al., 2010). Finally, a `Bucket` (Kerridge, et al., 1999) provides a flexible refinement of a barrier. Typically, there will be a collection of `Bucket`s into which processes are placed depending upon some criterion. Another process then, subsequently, causes a `Bucket` to flush all its processes so they are executed concurrently. These processes will in due course, become idle, whereupon they place themselves in other buckets. The next `Bucket` in sequence is then flushed and so the cycle is repeated. `Bucket`s can be used to control discrete event simulations in a very simple manner. The process that undertakes the flushing of the buckets must not be one of the processes that can reside in a `Bucket`.

The aim of this example is to present a user with a randomly chosen set of targets that each appear for a different random time. During the time the targets are available the user clicks the mouse over each of the targets in an attempt to hit as many of the targets as possible. The display includes information of how many targets have been hit and the total number of targets that have been displayed. The targets are represented by different coloured squares on a black background and a hit target is coloured white. A target that is not 'hit' before its self determined random time has elapsed is coloured grey. There is a gap between the end of one set of targets and the display of the next set during which time the screen is made all black. The minimum time for which a target is displayed is set by the user; obviously the longer this time the easier it is to hit the targets. Targets will be available for a period between the shortest time and twice that time. Figure 14-1 shows the screen, at the point when six targets have been displayed, and none have yet been hit. The system has displayed a total of 88 targets of which 15 targets have been hit. The minimum target delay was 900 milliseconds. It can be deduced there are 16 targets in a 4 × 4 matrix.

Download free eBooks at bookboon.com

The solution presumes that each target is managed by its own process and that it is these processes that are held in a `Bucket` until it is the turn of that `Bucket` to be flushed. When a target is enabled it displays itself until either it is 'hit' by a mouse-press, in which case it turns white, or the time for which it appears elapses and it is coloured grey. It is obvious that each of these target processes will finish at a different time and because the number of targets is not predetermined a barrier is used to establish when all the enabled target processes have finished. After this, the target process determines into which bucket it is going fall and thereby remains inactive until that bucket is flushed. The other processes used in the solution are shown in Figure 14-2.

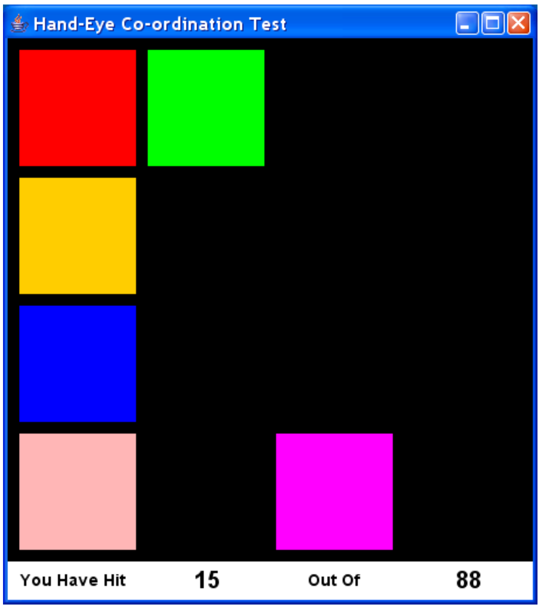


Figure 14-1 The Screen for the Hand-Eye Co-ordination Test

The system comprises a number of distinct phases each of which is controlled by its own barrier, which depending on the context is either a simple `Barrier` or an `AltingBarrier`.

Figure 14-2 shows the system at the point where it is about to synchronise on the `setUpBarrier`. During this setup phase there are no channel communications but the processes that synchronise on `setUpBarrier` either have to initialise themselves in some manner or must not progress beyond a certain point to ensure the system will not get out of step with itself. The setup phase only occurs once when the system is initially executed. The processes that are not part of the `setUpBarrier` cannot make any progress because they are dependent on other barriers or communications with processes that synchronise on the `setUpBarrier`.

Download free eBooks at bookboon.com

The `BarrierManager` is a process that is used to manage phase synchronisations and as such will be seen in subsequent figures to be part of a number of other barriers. For ease of description the structure of each phase will show only the relevant barrier and channels that are operative at that time. The separation into these distinct phases also makes it easier to analyse the system from the point of view of its client-server architecture, thereby enabling deadlock and livelock analysis.

The `TargetFlusher` and `TargetProcess` processes are the only processes that can manipulate the array of `Bucket`s. The `Buckets` are not shown on the diagram. The `TargetProcess`es are able to identify which `Bucket` they are going to enter when they stop running. `TargetFlusher` is the only process that can cause the flush and subsequent execution of the processes contained with a `Bucket`. The processing cycle of a `TargetProcess` is to wait until it is flushed from a `Bucket`; it then runs until it determines, itself, that it has ceased to run at which point it causes itself to `fallInto` a `Bucket`, which it also determines.

The `DisplayController` process initialises the display window to black. It also initialises, to zero, the information contained in the display widow as to the number of hits that have occurred and the total number of targets that have been displayed.

Download free eBooks at bookboon.com
Click on the ad to read more

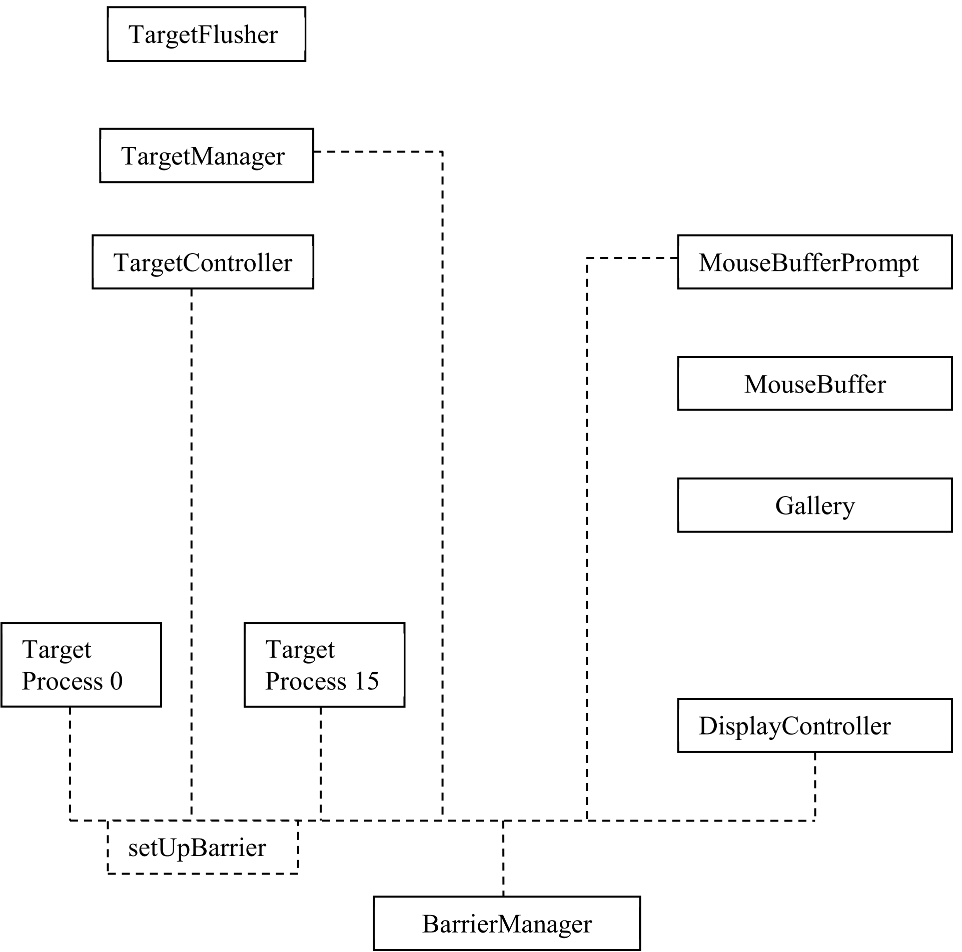


Figure 14-2 System At Setup Barrier Synchronisation

Figure 14-3 shows the system at the `initBarrier` synchronisation, which is the point at which those targets that are executing have initialised themselves and the associated display window is showing the targets. Prior to the `initBarrier` the only process that can execute is `TargetController`. The `TargetController` requests the `TargetManager` to flush the next `Bucket`; a request that is passed onto the `TargetFlusher` process. The `TargetFlusher` accesses the `Bucket`s in sequence until it finds a non-empty one. It then initialises the `initBarrier` with the number of `TargetProcess`es. It returns this number to the `TargetManager` and then flushes the `TargetProcess`es, which start running. The `TargetManager` then determines which of the `TargetProcess`es has been started by waiting for a communication from each of them informing it of the identity of the running targets. These identities are then formed into a `List`, which is then communicated to both the `TargetController` and `DisplayController` processes.

The TargetController can now construct a `ChannelOutputList` that will be subsequently used to communicate the location where mouse presses occur to each of the `TargetProcess`es. Similarly, the `DisplayController` can modify the display window to show the running targets.

Download free eBooks at bookboon.com

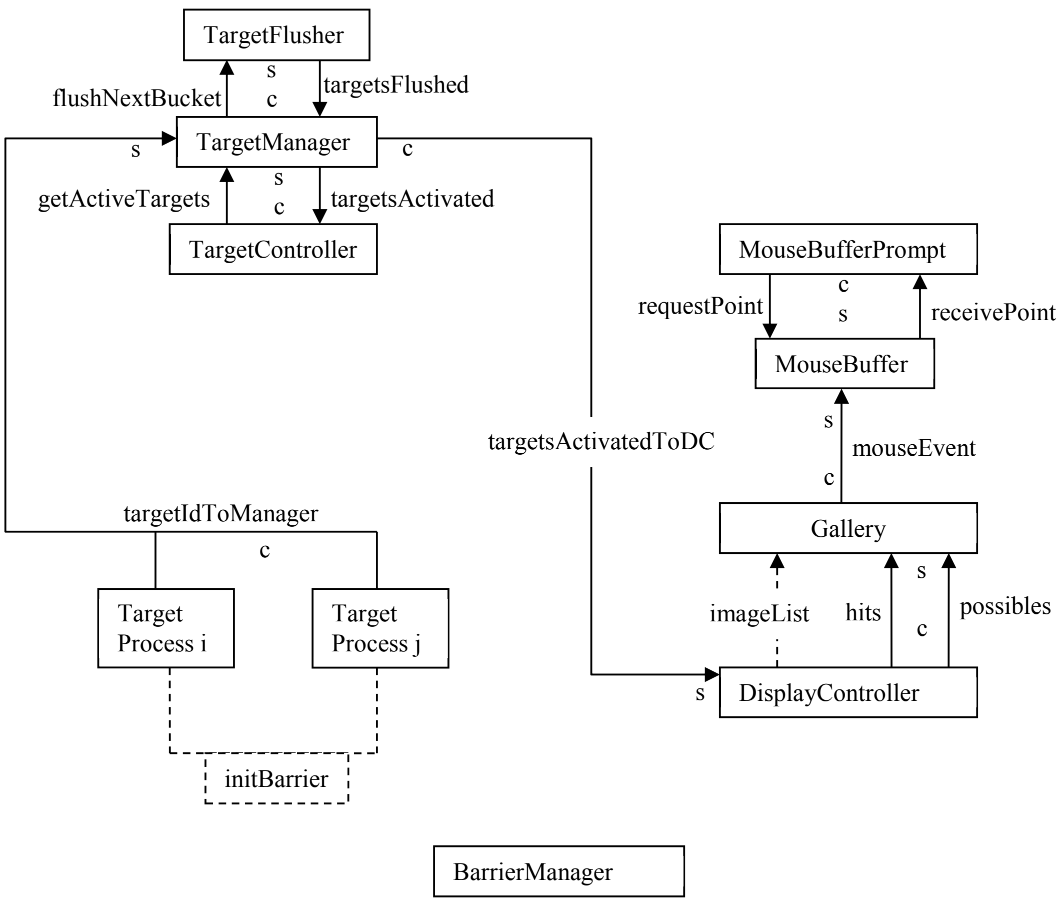


Figure 14-3 System At the Initialise Barrier Synchronisation

The `MouseBufferPrompt` and `MouseBuffer` have a design similar to that used previously in the manipulation of a queue (Chapter 6.2) and event handling (Chapter 11.2). `MouseBuffer` only accepts a request from `MouseBufferPrompt` when it has already received an event on its `mouseEvent` channel. The `Gallery` process is responsible both for the `ActiveCanvas` upon which the targets are displayed and the detection and communication of mouse click events. At this stage the `MouseBufferPrompt` process has no channel on which it can output points but that is not required until the system progresses to the next, `goBarrier` phase.

The `goBarrier` is simply required to ensure that all the running `TargetProcess`es, the `TargetController` and `DisplayController` have reached a state whereby the system can start execution from a known state. As such this phase does not require any channel communication as shown in Figure 14-4. Once these processes have synchronised the system enters the normal running state of the system with some of the `TargetProcess`es executing.

Download free eBooks at bookboon.com

Each of the `Barriers` used so far are of the simple variety because the number of processes that require synchronising can be predetermined and there is no need for any of these `Barriers` to interact with a possible communication or timer in an alternative. The communications are all required to have completed before the processes can reach the synchronisation point. The remaining `Barriers` are of the `AltingBarrier` variety because the requirement to synchronise can happen at the same time as a timer alarm or communication occurs.

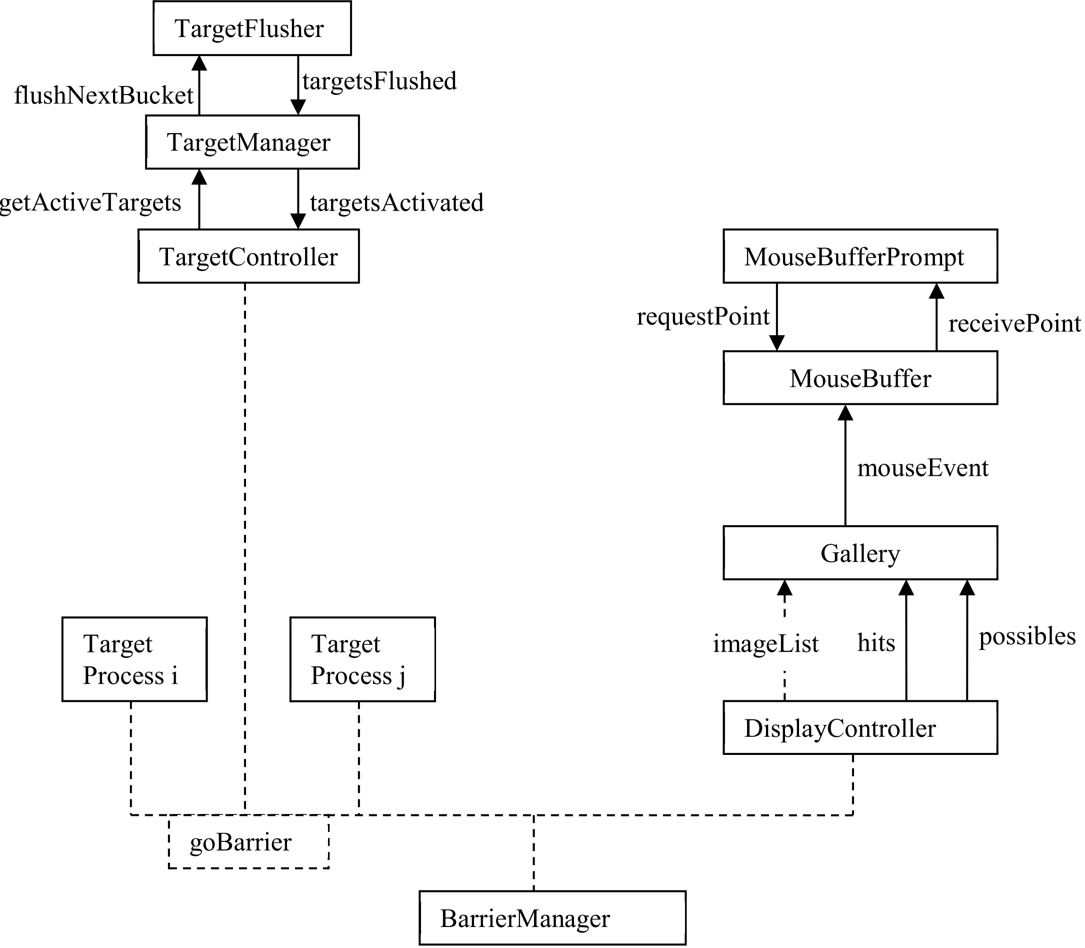


Figure 14-4 System At the Go Barrier Synchronisation

Figure 14-5 shows the system structure when the TargetProcesses are waiting for mouse clicks to determine whether or not they have been hit. The figure also shows the client-server analysis appropriate to this phase of the system's operation.

Download free eBooks at bookboon.com

Initial, cursory inspection, would seem to suggest that there a client-server loop has been created. However, it can be seen that the `MouseBuffer` is a pure server and therefore ensures that no loop is formed. Furthermore, the `Gallery` process provides a user interface capability that has some unusual properties. Any incoming communication is always fully acted upon within the process and is not transmitted further. Thus for its inputs the `Gallery` acts as a pure server. For any mouse events that it might generate, the Gallery acts as a pure client provided any event channels are communicated by a channel that utilises an overwriting buffer. This requirement is expounded further in the JCSP documentation and was discussed in Chapter 11.2.3.

The operation of a `TargetProcess` is specified as follows. After synchronising on the `goBarrier` it calculates its own random alarm time, which then forms part of an alternative that comprises the alarm and channel communications on its `mousePoints` channel. This alternative is looped around until either the alarm time occurs or the target is hit. In either case the target is no longer active. Another alternative is then entered that comprises communications on its `mousePoints` channel or the `timeAndHitBarrier`. Even though a target is inactive other targets may still not yet have timed out and thus mouse clicks will still be received. The `timeAndHitBarrier` determines when either all the targets have been hit or they have all timed out or some combination of these situations has occurred. It also has the effect of breaking the connection between `TargetController` and `MouseBufferPrompt` until the next set of targets are initialised. To ensure this does not cause a problem the channel `pointsToTC` uses an `OverWriteOldestBuffer` data store.

Click on the ad to read more

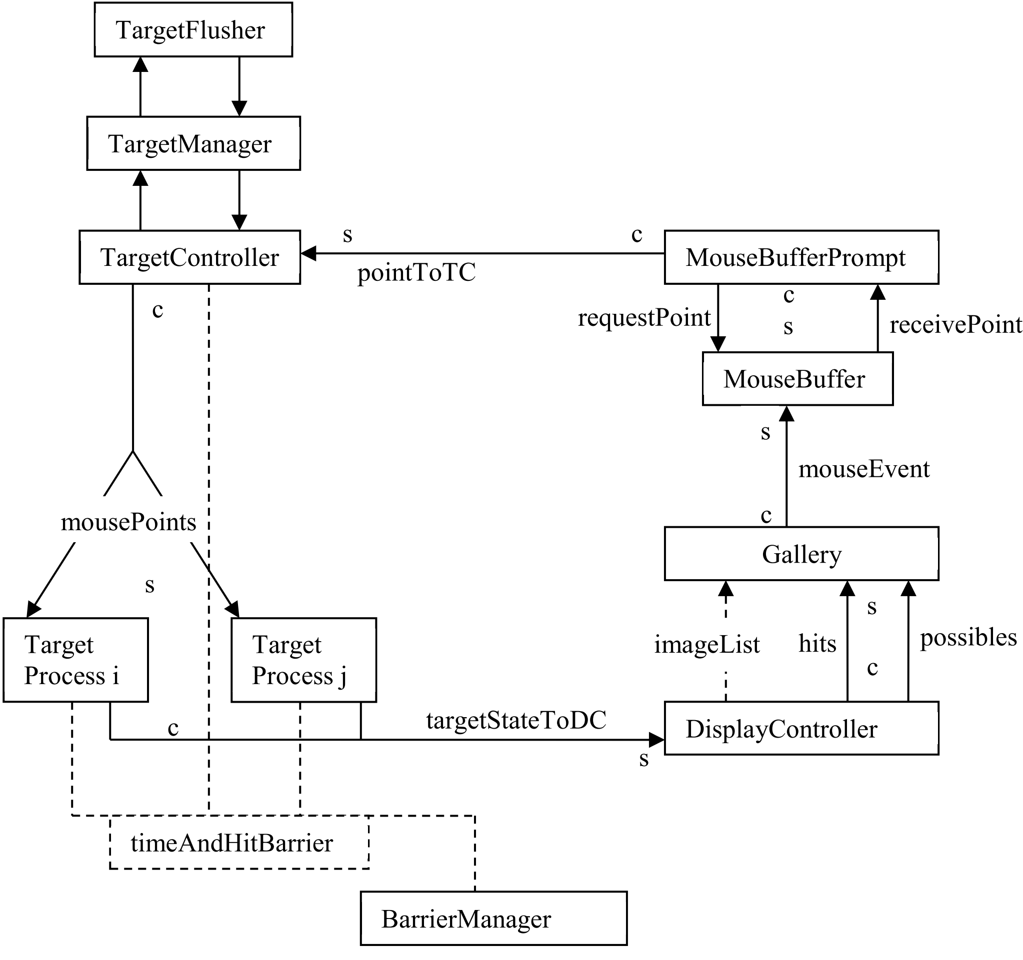


Figure 14-5 System Running Awaiting timeAndHitBarrier

When the state of a target changes (timed out or hit) it sends a communication to the `DisplayController` accordingly, which can then update the display maintained by `Gallery` appropriately. `TargetController` receives a `java.awt.Point` from `MouseBufferPrompt` that give the coordinates where the mouse has been pressed. The `TargetController` then outputs this `Point` value to each of the `TargetProcess`es in parallel using the `ChannelOutputList mousePoints`. Once all the targets have either been hit or timed out the `timeAndHitBarrier` synchronises at which point the `TargetProcess`es individually determine into which randomly chosen `Bucket` they are going to fall.

Download free eBooks at bookboon.com

The system then moves on to the final phase of processing shown in Figure 14-6. The `DisplayController` process contains an alternative with guards comprising the `finalBarrier` and the channel `targetStateToDC`. Thus when it is offering the guard `finalBarrier` together with `BarrierManager` the barrier synchronises and the system is able to progress onto another initial phase as described previously. The only process to undertake any substantial processing in the final phase is the `DisplayController` which leaves the final state of the display for a preset constant time, then sets all the targets to black, thereby obliterating them and then waits for another preset constant time. The coding of each of the processes now follows.

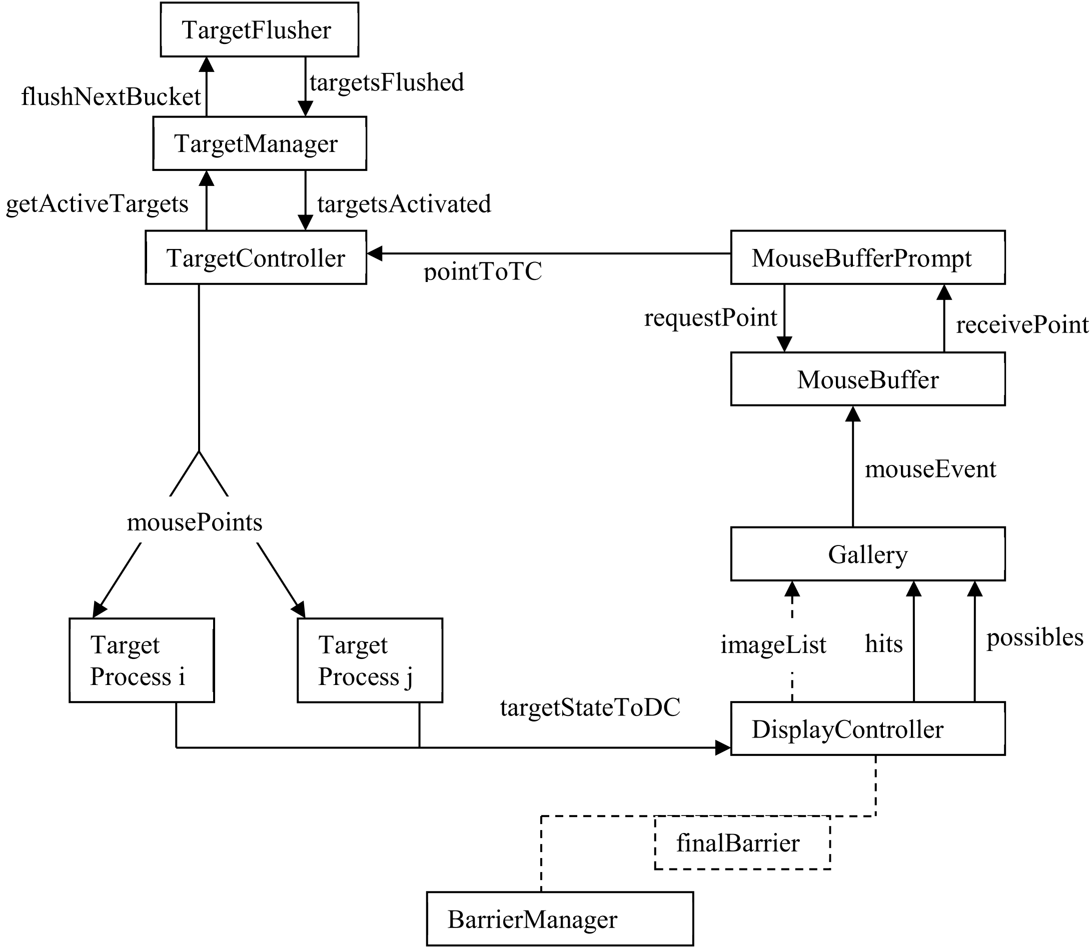


Figure 14-6 System At Final Barrier Synchronisation

Download free eBooks at bookboon.com

14.1 Barrier Manager

The BarrierManager, shown in Listing 14-1, simply defines as properties all the barriers in which it participates {12–15}. By definition an AltingBarrier must be part of an alternative and thus two ALTs are defined {18, 19} in which the particular AltingBarrier is the only guard. BarrierManager then waits to synchronise on setUpBarrier {20}. Thereafter, the process repeatedly synchronises on the goBarrier, timeAndHitBarrier and finalBarrier in sequence {23–25}. A Barrier synchronises using the sync() method call, whereas synchronisation on an AltingBarrier is achieved by calling the select() method call of the ALT that contains the barrier as a guard. In this case because the guard is the only element in the alternative a simple call of the select() method is sufficient, the value returned is of no importance. An alting barrier becomes enabled when all other members of the AltingBarrier also select() the same alting barrier.

```
10 class BarrierManager implements CSProcess{
11
12   def AltingBarrier timeAndHitBarrier
13   def AltingBarrier finalBarrier
14   def Barrier goBarrier
15   def Barrier setUpBarrier
16
17   void run() {
18     def timeHitAlt = new ALT ([timeAndHitBarrier])
19     def finalAlt = new ALT ([finalBarrier])
20     setUpBarrier.sync()
21
22     while (true){
23       goBarrier.sync()
24       def t = timeHitAlt.select()
25       def f = finalAlt.select()
26     }
27   }
28 }
```

Listing 14-1 Barrier Manager

14.2 Target Controller

Listing 14-2 shows the coding of the TargetController process, which is the process that effectively controls the operation of the complete system. The properties of the process are defined {12–20} and these directly implement the channel and barrier structures shown in Figures 14-2 to 14-6.

Download free eBooks at bookboon.com

```
10 class TargetController implements CSProcess {
11
12   def ChannelOutput getActiveTargets
13   def ChannelInput activatedTargets
14   def ChannelInput receivePoint
15   def ChannelOutputList sendPoint
16
17   def Barrier setUpBarrier
18   def Barrier goBarrier
19   def AltingBarrier timeAndHitBarrier
20   def int targets = 16
21
22     void run() {
23       def POINT = 1
24       def BARRIER = 0
25       def controllerAlt = new ALT ( [ timeAndHitBarrier, receivePoint] )
26
27       setUpBarrier.sync()
28       while (true) {
29         getActiveTargets.write(1)
30         def activeTargets = activatedTargets.read()
31         def runningTargets = activeTargets.size
32         def ChannelOutputList sendList = []
33         for ( t in activeTargets) sendList.append(sendPoint[t])
34         def active = true
```

Download free eBooks at bookboon.com

Click on the ad to read more

```
35          goBarrier.sync()
36          while (active) {
37            switch ( controllerAlt.priSelect() ) {
38              case BARRIER:
39                active = false
40                break
41              case POINT:
42                def point = receivePoint.read()
43                sendList.write(point)
44                break
45            } // end switch
46          } // end while active
47        } // end while true
48    } // end run
49 }
```

Listing 14-2 Target Controller

Within the `run` method some constants used to identify guards are defined {23, 24} of an alternative {25}. The zero'th guard of the alternative `controllerAlt` is the `AltingBarrier timeAndHitBarrier` and as such is incorporated into an `ALT` like any other guard. The process then waits for all the other enrolled processes to synchronise on `setUpBarrier` {27} before continuing with the unending loop {28–47} that is the main body of the process.

Download free eBooks at bookboon.com

Click on the ad to read more

The first action of the process is to send a signal {29} to the TargetManager process using the channel getActiveTargets. This is the first part of a client-server request and response pair of communications, the second of which is the receipt of a list of the targetIds of the activeTargets from the channel activatedTargets {30}. The activeTargets list is then used to create {33} a subset of the ChannelOutputList property sendPoint {15} in another ChannelOutputList sendList, which is used subsequently to communicate with each of the TargetProcesses. The Boolean property active is then defined {34} and will be used to control the subsequent operation of the process. The process now waits to synchronise on the goBarrier {35}. Prior to the goBarrier synchronisation all the TargetProcesses will have synchronised on the initBarrier but that is of no concern to the TargetController process.

The goBarrier is used to synchronise the operation of all the targets in the running TargetProcesses, the BarrierManager and the DisplayController as well as TargetController. The synchronisation enables each of these processes to run in that part of the system which allows users to move their mouse over the active targets and to try and hit each of them, by means of a mouse press, before each target times out. Thus the only actions that can occur are either, a mouse press occurs, or all the targets have either been hit or timed out. The mouse press manifests itself as the input of a Point on the receivePoint channel {42}. The value of point is then communicated, in parallel {43}, to all the members of sendList to each of the running TargetProcesses. (A write on a ChannelOutputList causes the writing of the method call parameter to all the channels in the list in parallel). If the barrier guard is selected then the loop terminates as soon as all the other processes on the timeAndHitBarrier have been selected {38}. The value of active is set false {39}which causes the inner while loop to terminate {36} ready for the process to cycle again round the outer non-terminating while loop {28}.

14.3 Target Manager

Listing 14-3 shows the coding of the TargetManager process. Its properties are defined {12–18}. The process does not have anything to do prior to the setUpBarrier synchronisation {21}. Its body comprises a non-terminating loop {22–34}. Initially, it reads the signal from TargetController on its getActiveTargets channel {24}, which causes the writing of yet a further signal to the TargetFlusher process on the flushNextBucket channel {25}. This is also the first part of the client-server communication pattern between TargetManager and TargetFlusher. The corresponding response is read from the targetsFlushed channel, which is the number of TargetProcesses that have been initialised into the variable targetsRunning {26}. The next phase {27–30} is to read from each of the initialised TargetProcesses their identity on the targetIdFromTarget channel and append it to the targetList {28}. This list is then written to the TargetController process {31} using the activatedTargets channel, thereby completing the client-server interaction between TargetManager and TargetController. Finally, the list of initialised targets is written to the DisplayController using the channel activatedTargetsToDC {32}. These two communications allow the receiving process to complete their initialisation prior to further operation. The process then cycles waiting to read the next group of actived targets {24}, which cannot be undertaken until the next bucket is flushed.

Download free eBooks at bookboon.com

```
10 class TargetManager implements CSProcess {
11
12   def ChannelInput targetIdFromTarget
13   def ChannelInput getActiveTargets
14   def ChannelOutput activatedTargets
15   def ChannelOutput activatedTargetsToDC
16   def ChannelInput targetsFlushed
17   def ChannelOutput flushNextBucket
18   def Barrier setUpBarrier
19
20   void run() {
21     setUpBarrier.sync()
22     while (true) {
23       def targetList = [ ]
24       getActiveTargets.read()
25       flushNextBucket.write(1)
26       def targetsRunning = targetsFlushed.read()
27       while (targetsRunning > 0) {
28           targetList << targetIdFromTarget.read()
29           targetsRunning = targetsRunning - 1
30       } // end of while targetsRunning
31       activatedTargets.write(targetList)
32       activatedTargetsToDC.write(targetList)
33     } // end of while true
34   }
34 }
```

Listing 14-3 Target Manager

14.4 Target Flusher

The role of the `TargetFlusher` process, shown in Listing 14-4, is to manage the `Buckets` into which the `TargetProcess`es fall.

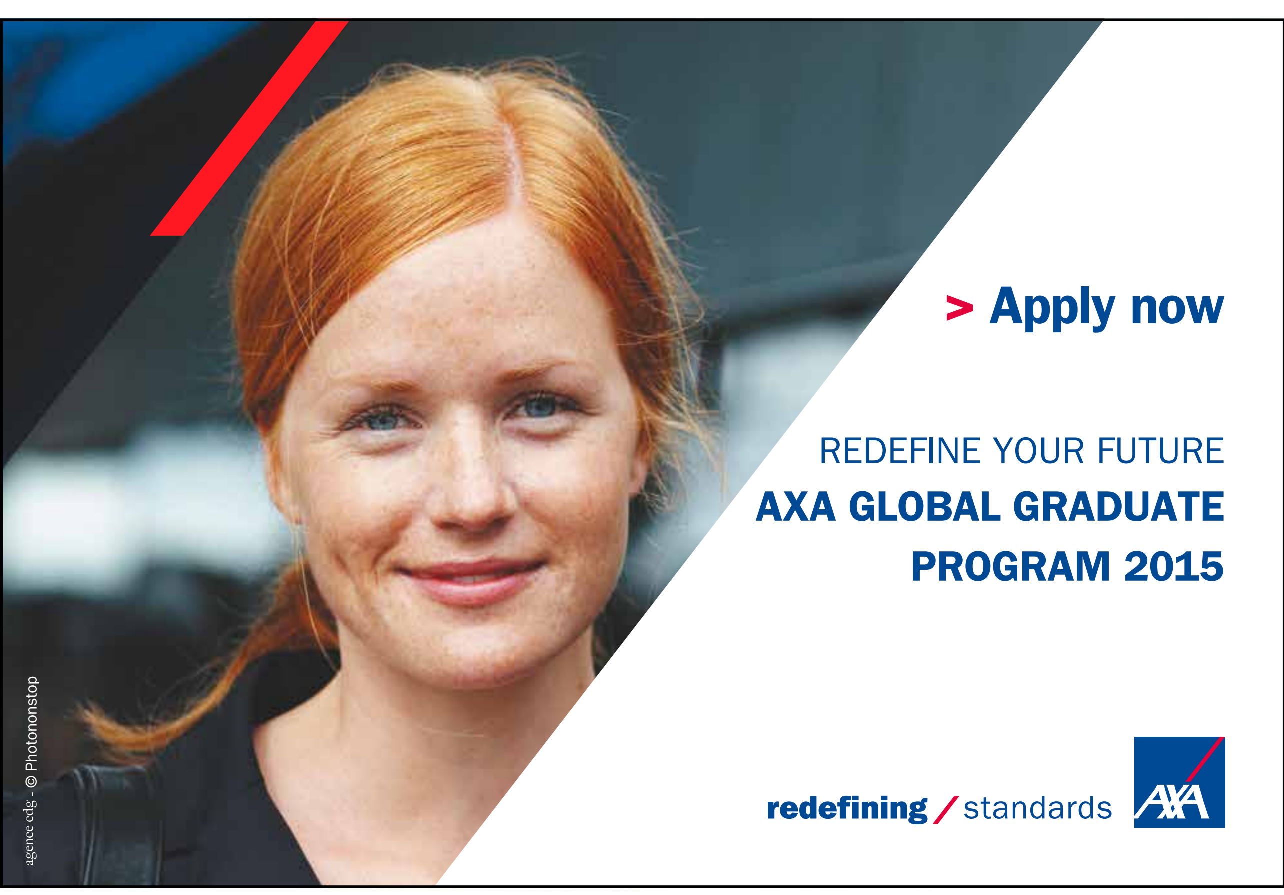
```
10 class TargetFlusher implements CSProcess {
11
12   def buckets
13   def ChannelOutput targetsFlushed
14   def ChannelInput flushNextBucket
15   def Barrier initBarrier
16
17   void run() {
18     def nBuckets = buckets.size()
19     def currentBucket = 0
20     def targetsInBucket = 0
21     while (true) {
22       flushNextBucket.read()
23       targetsInBucket = buckets[currentBucket].holding()
```

Download free eBooks at bookboon.com

```
24        while ( targetsInBucket == 0) {
25          currentBucket = (currentBucket + 1) % nBuckets
26          targetsInBucket = buckets[currentBucket].holding()
27        } // end of while targetsInBucket
28        initBarrier.reset( targetsInBucket)
29        targetsFlushed.write(targetsInBucket)
30        buckets[currentBucket].flush()
31        currentBucket = (currentBucket + 1) % nBuckets
32      } // end of while true
33    }
34 }
```

Listing 14-4 Target Flusher

The process also completes the client-server interaction with the `TargetManager` process. Its properties are defined {12–15}. Some variables are initialised {18–20} in the first part of the `run` method. The main loop of the process {21–32} initially reads the signal {22} that causes it to start the initialisation of some `TargetProcess`es. The number of `TargetProcess`es in the `currentBucket` is determined by means of a call of the `holding()` method {23}. The next piece of coding {24–27} ensures that the number of `TargetProcess`es that are `flush`ed is greater then zero.



Download free eBooks at bookboon.com
Click on the ad to read more

At this stage initBarrier can be set to the number of targetsInBucket {28} by means of a call to the reset method. The number of targetsInBucket can now be written to the TargetManager process {29}. Now the TargetProcesses contained in the currentBucket can be flushed {30} and therefore start running. Finally, the value of currentBucket can be incremented subject to its value staying within zero to the number of Buckets, nBuckets {31}.

14.5 Display Controller

The DisplayController process is shown in Listings 14-5 to 14-8 and manages the interaction between the TargetProcesses and the user interface provided by the Gallery process, described in the next section.

The TargetProcesses communicate with the DisplayController by means of the channel stateChange {11}, which is the 'one' end of an any2one channel. The channel activeTargets {12} is used to input the list of running targets during the initial phase of a cycle. The displayList property {14} provides the connection between this process and the ActiveCanvas contained in the Gallery process. The channels hitsToGallery and possiblesToGallery {15, 16} are used to send values to the ActiveLabels in the Gallery process that display the number of targets that have been hit and the total number of targets displayed. Finally, the barriers upon which DisplayController synchronises are defined {18–20}.

```
10 class DisplayController implements CSProcess {
11    def ChannelInput stateChange
12    def ChannelInput activeTargets
13
14    def DisplayList displayList
15    def ChannelOutput hitsToGallery
16    def ChannelOutput possiblesToGallery
17
18    def Barrier setUpBarrier
19    def Barrier goBarrier
20    def AltingBarrier finalBarrier
21
```

Listing 14-5 Display Controller Properties

Listing 14-6 gives the array of GraphicsCommands and list of values used to change the displayList. These are not shown complete, but are those parts that relate to the first and last. The array targetGraphics is used to initially create the displayList. Each of the elements of the list targetColour comprises the colour of the target and the element of targetGraphics that has to be changed in order to display the target. The first two elements of targetGraphics {25, 26} have the effect of completely 'blacking' out the display canvas prior to its repainting within the Canvas thread.

Download free eBooks at bookboon.com

```
22 void run() {
23
24   def GraphicsCommand [] targetGraphics = new GraphicsCommand [ 34 ]
25   targetGraphics[0] = new GraphicsCommand.SetColor (Color.BLACK)
26   targetGraphics[1] = new GraphicsCommand.FillRect (0, 0, 450, 450)
27   targetGraphics[2] = new GraphicsCommand.SetColor (Color.BLACK)
28   targetGraphics[3] = new GraphicsCommand.FillRect (10, 10, 100, 100)
29   targetGraphics[4] = new GraphicsCommand.SetColor (Color.BLACK)
30   targetGraphics[5] = new GraphicsCommand.FillRect (120, 10, 100, 100)
31   targetGraphics[6] = new GraphicsCommand.SetColor (Color.BLACK)
32   targetGraphics[7] = new GraphicsCommand.FillRect (230, 10, 100, 100)
33 …
34   targetGraphics[30] = new GraphicsCommand.SetColor (Color.BLACK)
35   targetGraphics[31] = new GraphicsCommand.FillRect (230, 340, 100, 100)
36   targetGraphics[32] = new GraphicsCommand.SetColor (Color.BLACK)
37   targetGraphics[33] = new GraphicsCommand.FillRect (340, 340, 100, 100)
38
39   def targetColour = [
40     [new GraphicsCommand.SetColor (Color.RED), 2],
41     [new GraphicsCommand.SetColor (Color.GREEN), 4],
42     [new GraphicsCommand.SetColor (Color.YELLOW), 6],
43     [new GraphicsCommand.SetColor (Color.BLUE), 8],
44 …
45     [new GraphicsCommand.SetColor (Color.MAGENTA), 30],
46     [new GraphicsCommand.SetColor (Color.ORANGE), 32]
47                        ]
```

Listing 14-6 Graphics definitions

The `run` method has some further properties that are shown in Listing 14-7, which include the constants {48, 49} used to identify the selected alternative defined as `controllerAlt` {52}. The constants {54–56} define the `GraphicsCommand` that can be used to colour a square as indicated by their name. Finally, variables that tally the number of hits and possible hits are defined {58, 59} together with a timer {60} that is used to control the time the display stays static at the end of a cycle.

```
48      def CHANGE = 1
49      def BARRIER = 0
50      def TIMED_OUT = 0
51      def HIT = 1
52      def controllerAlt = new ALT ( [ finalBarrier, stateChange ] )
53
54      def whiteSquare = new GraphicsCommand.SetColor(Color.WHITE)
55      def blackSquare = new GraphicsCommand.SetColor(Color.BLACK)
56      def graySquare = new GraphicsCommand.SetColor(Color.GRAY)
57
58      def totalHits = 0
59      def possibleTargets = 0
60      def timer = new CSTimer()
```

Listing 14-7 Other Run Method Properties

Download free eBooks at bookboon.com

The body of the `run` method is shown in Listing 14-8. Prior to the `setUpBarrier` synchronisation {64} the `displayList` is initialised by a call to the `set` method {61} and the initial, zero, values of `totalHits` and `possibleHits` are written to the `Gallery` {62, 63}.

The never ending loop of the `run` method is then entered {66–99}, which comprises some initialisation prior to the `goBarrier` synchronisation {67–73} followed by the active part of the cycle {74–93} until the `finalBarrier` is selected {89–90}.

```
61  displayList.set (targetGraphics)
62  hitsToGallery.write (" " + totalHits)
63  possiblesToGallery.write ( " " + possibleTargets )
64  setUpBarrier.sync()
65
66  while (true) {
67    def active = true
68    def runningTargets = activeTargets.read()
69    possibleTargets = possibleTargets + runningTargets.size
70    possiblesToGallery.write ( " " + possibleTargets )
71    for ( t in runningTargets)
72      displayList.change ( targetColour[t][0], targetColour[t][1])
73    goBarrier.sync()
74    while (active) {
75      switch (controllerAlt.priSelect()) {
```

Empowering People.
Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

Download free eBooks at bookboon.com

```
76          case CHANGE:
77              def modification = stateChange.read() // [ tId, state ]
78              switch ( modification[1]) {
79                case HIT:
80                  displayList.change ( whiteSquare, targetColour[modification[0]][1])
81                  totalHits = totalHits + 1
82                  hitsToGallery.write (" " + totalHits)
83                 break
84                case TIMED_OUT:
85                  displayList.change ( graySquare, targetColour[modification[0]][1])
86                break
87              } // end switch modification
88              break
89          case BARRIER:
90              active = false
91              break
92          } // end switch controllerAlt
93        } // end of while active
94        timer.sleep(1500)
95        for ( tId in runningTargets )
96            displayList.change ( blackSquare, targetColour[tId][1])
97        timer.sleep ( 500)
98      } // end while true
99    }
100 }
```

Listing 14 – 8 Run Method Definition

The process `DisplayController` is initialised by reading the identities of the running targets into the list `runningTargets` from `TargetManager` using the channel `activeTargets` {68}. The `size` of this list is then used to update the total number of possible targets in the `Gallery` {69–70}. The members of the list are then used to `change` the `displayList`, which cause the targets to appear in the `Gallery` {71–72}. The process then synchronises on the `goBarrier` {73}.

The process remains `active` {74} until the `finalBarrier` is selected {89–90}. It should be noted that the order of the guards in `controllerAlt` is important, with priority given to inputs from the `TargetProcess`es, so that all changes to the targets are completed before the `finalBarrier` is `select`ed. While the process is `active`, communications from the running `TargetProcess`es are `read` from the channel `stateChange` {77} which are used to modify the state of the targets in the `Gallery` by `chang`ing the `displayList`. The input from a `TargetProcess` is in the form of a list comprising the identity of the target and the state to which it should be changed. Two state changes are possible indicated by `HIT`, when the target's image is changed to white {80} and the number of targets hit is also updated {81–82} and `TIMED_OUT` when the square is coloured grey {85}.

Download free eBooks at bookboon.com

Once the finalBarrier has been selected {89, 90} the process sleeps for 1.5 seconds {94} to allow the user to determine the final state of that cycle. The running targets, which are now all coloured either white or grey are returned to the colour black {95–96}. The process sleeps for a further 0.5 seconds {97}, to provide the user a break between cycles of the system. It then resumes the main loop of the process, which is initiated by reading the identities of the targets that have been flushed from the next Bucket.

14.6 Gallery

The Gallery process shown in Listing 14-9 is similar to other user interface processes that have been discussed previously. The only aspect of particular note is that a mouse event channel {15} is added to the ActiveCanvas {39}. There is no need for the programmer to add anything further in terms of listener of event handling methods. Any mouse event is communicated on the mouseEvent channel to the MouseBuffer process. The components of the interface can be seen, by observation, to produce that shown in Figure 14-1.

```
10 class Gallery implements CSProcess{
11
12    def ActiveCanvas targetCanvas
13    def ChannelInput hitsFromGallery
14    def ChannelInput possiblesFromGallery
15    def ChannelOutput mouseEvent
16    def canvasSize = 450
17
18    void run() {
19      def root = new ActiveClosingFrame ("Hand-Eye Co-ordination Test")
20      def mainFrame = root.getActiveFrame()
21      def m1 = new Label ("You Have Hit")
22      def m2 = new Label ("Out Of")
23      def hitLabel = new ActiveLabel (hitsFromGallery)
24      def possLabel = new ActiveLabel (possiblesFromGallery)
25      m1.setAlignment( Label.CENTER)
26      m2.setAlignment( Label.CENTER)
27      hitLabel.setAlignment( Label.CENTER)
28      possLabel.setAlignment( Label.CENTER)
29      m1.setFont(new Font("sans-serif", Font.BOLD, 14))
30      m2.setFont(new Font("sans-serif", Font.BOLD, 14))
31      hitLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
32      possLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
33      def message = new Container()
34      message.setLayout ( new GridLayout ( 1, 4 ) )
35      message.add (m1)
36      message.add (hitLabel)
37      message.add (m2)
38      message.add (possLabel)
39      targetCanvas.addMouseEventChannel ( mouseEvent )
```

Download free eBooks at bookboon.com

```
40      mainFrame.setLayout( new BorderLayout() )
41      targetCanvas.setSize (canvasSize, canvasSize)
42      mainFrame.add (targetCanvas, BorderLayout.CENTER)
43      mainFrame.add (message, BorderLayout.SOUTH)
44      mainFrame.pack()
45      mainFrame.setVisible ( true )
46      def network = [ root, targetCanvas, hitLabel, possLabel ]
47      new PAR (network).run()
48    }
49 }
```

Listing 14-9 Gallery Process

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Go to www.helpmyassignment.co.uk for more info

Download free eBooks at bookboon.com

14.7 Mouse Buffer

The `MouseBuffer`, shown in Listing 14-10 process `read`s mouse events on its `mouseEvent` channel {12}. Only when the event is a `MOUSE_PRESSED` event does it store the location of the click {35} in the variable `point`. At this stage it modifies {34} the pre-condition on the process' alternative, `mouseBufferAlt` so as to be able to accept requests for a `point` {26}, which can then be transferred to the `MouseBufferPrompt` process {28}, after which the pre-condition is again modified {29} so as not to accept further prompt requests until another mouse click `point` has been received. This mechanism was used previously in the `Queue` and `Event Handling Systems` and is an idiom or pattern used to manage requests for external non-deterministic events. In this case we note that the `mouseEvent` channel is always available to read events and thus does not block the `Gallery` process with its implicit threads that are used to implement events and a canvas. This is further demonstrated by the `mouseEvent` channel having a data store associated with it that enables the overwriting of the oldest member of the associated buffer (see 14.10).

```
10  class MouseBuffer implements CSProcess{
11
12     def ChannelInput mouseEvent
13     def ChannelInput getClick
14     def ChannelOutput sendPoint
15
16     void run() {
17       def mouseBufferAlt = new ALT ( [ getClick, mouseEvent ] )
18       def preCon = new boolean [2]
19       def EVENT = 1
20       def GET = 0
21       preCon[EVENT]= true
22       preCon[GET] = false
23       def point
24       while (true) {
25         switch (mouseBufferAlt.select(preCon)) {
26           case GET:
27             getClick.read()
28             sendPoint.write(point)
29             preCon[GET] = false
30             break
31           case EVENT:
32             def mEvent = mouseEvent.read()
33             if ( mEvent.getID() == MouseEvent.MOUSE_PRESSED) {
34               preCon[GET] = true
35               point = mEvent.getPoint()
36             }
37             break
38         }
39       }
40     }
41  }
```

Listing 14-10 Mouse Buffer Process

Download free eBooks at bookboon.com

14.8 Mouse Buffer Prompt

The MouseBufferPrompt process shown in Listing 14-11, simply writes a request to the getPoint channel {20} and then waits to read a point on the receivePoint channel {21} which it then writes to the TargetController process on the returnPoint channel {22}. The combination of MouseBufferPrompt and MouseBuffer ensures that the MouseBuffer process is a pure server in a client-server analysis and also has the effect of decoupling the generation of mouse events in the Gallery from the process in which they are consumed, TargetController. Furthermore, any delay in reading a point by the TargetController does not cause a delay that might cause the blocking of the implicit event handling thread of Gallery.

```
10  class MouseBufferPrompt implements CSProcess{
11
12      def ChannelOutput returnPoint
13      def ChannelOutput getPoint
14      def ChannelInput receivePoint
15      def Barrier setUpBarrier
16
17      void run () {
18        setUpBarrier.sync()
19        while (true) {
20          getPoint.write( 1 )
21          def point = receivePoint.read()
22          returnPoint.write( point )
23        }
24      }
25  }
```

Listing 14-11 Mouse Buffer Prompt Process

14.9 Target Process

The TargetProcess is shown in Listings 14-12 to 14-14. The channel targetRunning {12} is used by TargetProcess to inform the TargetManager process that it has been flushed from a Bucket and has been made active. The channel stateToDC {13} is used to inform the DisplayController of any change in state of this target that is, either hit or timed-out. The channel mousePoint {14} is used to input the java.awt.Point at which the mouse has been pressed. The process is a member of the setUp, init, go and timeAndHit barriers {15–18}. It also requires access to the array of buckets {19}. The property targetId {20} is a unique integer identifying the instance of TargetProcess and the values x {21} and y {22} are the pixel co-ordinates of the upper left corner of the target in the display window. The property delay {23} specifies the minimum period for which the target will be displayed before it times out. The target will be visible for a random time between delay and twice delay. The method within {25–33} determines if a java.awt.Point p is within the target area. All targets are square with a side of 100 pixels.

Download free eBooks at bookboon.com

```
10  class TargetProcess implements CSProcess {
11
12     def ChannelOutput targetRunning
13     def ChannelOutput stateToDC
14     def ChannelInput mousePoint
15     def Barrier setUpBarrier
16     def Barrier initBarrier
17     def Barrier goBarrier
18     def AltingBarrier timeAndHitBarrier
19     def buckets
20     def int targetId
21     def int x
22     def int y
23     def delay = 2000
24
25     def boolean within ( Point p, int x, int y) {
26       def maxX = x + 100
27       def maxY = y + 100
28       if ( p.x < x ) return false
29       if ( p.y < y ) return false
30       if ( p.x > maxX ) return false
31       if ( p.y > maxY ) return false
32       return true
33     }
34
```

Listing 14-12 The Properties and Within Method of target process

Download free eBooks at bookboon.com

Click on the ad to read more

The first part of the run method is executed during the setup phase of the system and is only executed once, Listing 14-13. A Random number generator rng {36} is defined and then used to specify the initial bucket, bucketId {38, 39} into which the TargetProcess will subsequently fall. Initially all TargetProcesses will fall into a bucket in the first half of the array of buckets. A timer and some constants are also defined {37, 40–44}.

Two alternatives are then defined. The alternative preTimeOutAlt {46} is used prior to the TargetProcess being timed out and postTimeOutAlt {47} is used once a time out has occurred or the target has been hit. The latter alternative includes the AltingBarrier timeAndHitBarrier. The operation of such an AltingBarrier is straightforward. It must appear as a guard in an alternative. Whenever any select method on the alternative is called a check is made to determine whether all the other members of the AltingBarrier have also requested and are waiting on such a select. If they have, then, the AltingBarrier as a whole can be selected. If one of the members of an AltingBarrier accepts another guard in such an alternative then the AltingBarrier cannot be selected. Thus it is possible for a process to offer an AltingBarrier guard and then withdraw from that guard if the dynamics of the system cause that to happen.

The TargetProcess now resigns from timeAndHitBarrier {49}, which at first sight may seem perverse. All TargetProcesses are initially enrolled on this barrier. However we only want running targets to be counted as part of the barrier so we must first resign from the barrier and then enroll only when the TargetProcess is executed.

The mechanism of enroll and resign can be applied to all types of barrier. A process that enrolls on a barrier can now call the sync method (Barrier) or be a guard in an alternative and thus can be selected (AltingBarrier). Similarly a process can resign which means that the process is no longer part of the barrier. In the case of a Barrier resignation it also implies that if this is the last process to synchronise on the Barrier then this is equivalent to all the processes having synchronised. A process cannot resign if it is not enrolled. In the case of AltingBarriers this enrolment and resignation has to be undertaken with care as no process can be running and selecting the barrier onto which it is intended to either enrol or resign another process from. The associated documentation for JCSP specifies this requirement more fully.

The TargetProcesses now synchronise on the setUpBarrier {50} and when this is achieved they then fallInto the bucket with subscript bucketId {51}. This has the effect of stopping the process. It will only be rescheduled when the TargetFlusher process causes the bucket into which the process has fallen is flushed {Listing 14-4, 30}.

```
35    void run() {
36      def rng = new Random()
37      def timer = new CSTimer()
```

Download free eBooks at bookboon.com

```
38      def int range = buckets.size() / 2
39      def bucketId = rng.nextInt( range )
40      def POINT= 1
41      def TIMER = 0
42      def BARRIER = 0
43      def TIMED_OUT = 0
44      def HIT = 1
45
46      def preTimeOutAlt = new ALT ([ timer, mousePoint ])
47      def postTimeOutAlt = new ALT ([ timeAndHitBarrier, mousePoint ])
48
49      timeAndHitBarrier.resign()
50      setUpBarrier.sync()
51      buckets[bucketId].fallInto()
```

Listing 14-13 Target process: The Setup Phase of Run

The remainder of the `run` method, Listing 14-14, only gets executed when the process has been `flush`ed. It comprises a never ending loop {52–94}, which as its final statement {93} causes itself to fall into another bucket, prior to returning to the start of the loop. The loop itself has three phases comprising the phases managed by `initBarrier` and then that managed by the `goBarrier` before finally running until either the target is hit or times out which is managed by the `timeAndHitBarrier`.

In the initial phase, the process `enroll`s on the `timeAndHitBarrier` {53} and also the `goBarrier` {54}. Enrolling on the `timeAndHitBarrier` causes no problem because at this stage no process can be selecting a guard from an alternative in which `timeAndHitBarrier` is involved. Similarly, enrolling on the `goBarrier` is an operation that can be undertaken dynamically because it is a `Barrier`. The running process now writes its unique identity, `targetId` to its `targetRunning` channel {55}. This communication means that the `TargetManager` now can determine {Listing 14-4, 27–30} which targets are active. It then synchronises on the `initBarrier` {56}. The number of running `TargetProcess`es associated with the `initBarrier` is specified by `TargetFlusher` {Listing 14-3, 29} at a time when none of these processes can be running because they have yet to be flushed. Only the running `TargetProcess`es are allowed to access the `initBarrier` and thus once the `initBarrier` has synchronised we know that all the `TargetProcess`es are in the same state and that any dependent processes such as `DisplayController` will be able to complete any further initialisation prior to the `goBarrier` synchronisation. The Boolean `running` is initialised {57}, which will be used subsequently to control the operation of the process. Similarly, the variable `resultList` is initialised {58} and will be used to indicate the change of state that will occur in the target. The process can now synchronise on the `goBarrier` by `resign`ing from it {59}. The only permanent members of the `goBarrier` are `BarrierManager`, `TargetController` and `DisplayController`, all of which simply call the method `sync()` on the barrier . The `goBarrier` is augmented by the active `TargetProcess`es to ensure that all the processes are in a state that will be suitable for the whole system to become active.

Download free eBooks at bookboon.com

Once the process has synchronised on the goBarrier it determines the time for which the target will be displayed and sets the timer alarm {60} which is a guard in the preTimeOutAlt (46). Prior to the alarm occurring only two things can occur, either the TIMER alarm does happen {63} or a mouse click POINT is received {68}. In the former case, the value TIMED_OUT can be appended to the resultList {65} and this list can be written to the DisplayController using the channel stateToDC {66}. Otherwise, an input can be processed {69} which, if it is within the target area {70} causes the value HIT to be appended to the resultList {72} and as before written to the DisplayController process {73}. If the point is not within the target then the loop repeats until one of the above cases occurs. Once this happens the value of running is set false {64} and the loop {61–79} terminates.

The process now has take account of the case where other targets are still running; awaiting a time out or a hit, and so mouse clicks and their associated point data will still be received by the TargetProcess. Such point data can be ignored {87–89} and only when all the TargetProcesses are selecting the timeAndHitBarrier, together with TargetController and BarrierManager processes can the awaiting loop {81–91} terminate. When this occurs the process resigns from the timeAndHitBarrier and causes the loop to exit.

Download free eBooks at bookboon.com

Click on the ad to read more

The TargetProcess can now prepare itself for falling into another bucket by calculating {92} into which bucket it will fall and then calling the `fallInto` method {93}. The chosen bucket is at least two further on than the current bucket which means that it cannot be flushed in the next iteration of `TargetFlusher`, unless the next bucket happens to be empty.

```
52      while (true) {
53        timeAndHitBarrier.enroll()
54        goBarrier.enroll()
55        targetRunning.write(targetId)
56        initBarrier.sync() //ensures all targets have initialised
57        def running = true
58        def resultList = [targetId]
59        goBarrier.resign()
60        timer.setAlarm( timer.read() + delay + rng.nextInt(delay) )
61        while ( running ) {
62          switch (preTimeOutAlt.priSelect() ) {
63            case TIMER:
64              running = false
65              resultList << TIMED_OUT
66              stateToDC.write(resultList)
67              break
68            case POINT:
69              def point = mousePoint.read()
70              if ( within(point, x, y) ) {
71                running = false
72                resultList << HIT
73                stateToDC.write(resultList)
74              }
75              else {
76              }
77              break
78          }
79        } // end while running
80        def awaiting = true
81        while (awaiting) {
82          switch (postTimeOutAlt.priSelect() ) {
83            case BARRIER:
84              awaiting = false
85              timeAndHitBarrier.resign()
86              break
87            case POINT:
88              mousePoint.read()
89              break
90          }
91        } // end while awaiting
92        bucketId = ( bucketId + 2 + rng.nextInt(8) ) % buckets.size()
```

Download free eBooks at bookboon.com

```
93          buckets[bucketId].fallInto()
94        }// end while true
95      }
96 }
```

Listing 14-14 Target Process: The Active Phase of the Run Method

14.10 Running the System

Listing 14-15 gives the declarations of the channels, barriers and other data required to create the network according to the process network diagrams given in Figures 14-2 to 14-6 and as such are not particularly noteworthy apart from those described below. The `Barrier`s are defined with the required number of processes. Thus `setUpBarrier` {18} is defined with the number of `targets` plus five for the other processes that use this barrier, see Figure 14-2. The `initBarrier` {19} is defined with no members because only the running `TargetProcess`es use this barrier and the number is `reset` explicitly in `TargetFlusher`, see Figure 14-3. Finally, the `goBarrier` {20} is defined has having three members, which are the permanently attached processes as shown in Figure 14-4.

The `AltingBarrier`s are defined as an array, with sufficient members such that every process that access them may have a so-called 'front-end'. The `finalBarrier` {23} only requires two front-ends because only `BarrierManager` and `DisplayController` participate in this barrier. The barrier `timeAndHitBarrier` {22} requires a front-end for each `TargetProcess`, the `TargetController` and `BarrierManager`. Each process participating in an `AltingBarrier` needs to be allocated its own front-end so that it can access the barrier during an alternative `select()` method call. Recall that as a `TargetProcess` becomes active it specifically `enrolls` on the `timeAndHitBarrier` thereby activating its membership of the barrier and when its turn is complete it `resigns` from it. Thus the number of processes that are members of the `timeAndHitBarrier` is determined dynamically at run time. The `Bucket`s are defined by means of a `create` method call {25} and this could be any sensible number to provide a wide variety of target initiations per cycle, too many buckets and we would get too few running targets to make the challenge interesting!

```
10  def delay = Ask.Int("Target visible period (2000 to 3500)? ", 2000, 3500)
11
12  def targets = 16
13  def targetOrigins = [ [10, 10],[120, 10],[230, 10],[340, 10],
14                        [10, 120],[120, 120],[230, 120],[340, 120],
15                        [10, 230],[120, 230],[230, 230],[340, 230],
16                        [10, 340],[120, 340],[230, 340],[340, 340] ]
17
18  def setUpBarrier = new Barrier(targets + 5)
19  def initBarrier = new Barrier()
20  def goBarrier= new Barrier(3)
21
22  def timeAndHitBarrier = AltingBarrier.create(targets+2)
```

Download free eBooks at bookboon.com

```
23  def finalBarrier = AltingBarrier.create(2)
24
25  def buckets = Bucket.create(targets)
26
27  def mouseEvent = Channel.one2one ( new OverWriteOldestBuffer(20) )
28  def requestPoint = Channel.one2one()
29  def receivePoint = Channel.one2one()
30  def pointToTC = Channel.one2one( new OverWriteOldestBuffer(1) )
31
32  def targetsFlushed = Channel.one2one()
33  def flushNextBucket = Channel.one2one()
34
35  def targetsActivated = Channel.one2one()
36  def targetsActivatedToDC = Channel.one2one()
37  def getActiveTargets = Channel.one2one()
38
39  def hitsToGallery = Channel.one2one()
40  def possiblesToGallery = Channel.one2one()
41
42  def targetIdToManager = Channel.any2one()
43  def targetStateToDC = Channel.any2one()
44
45  def mousePointToTP = Channel.one2oneArray(targets)
```

Download free eBooks at bookboon.com
Click on the ad to read more

```
46  def mousePoints = new ChannelOutputList ( mousePointToTP )
47
48  def imageList = new DisplayList()
49  def targetCanvas = new ActiveCanvas ()
50  targetCanvas.setPaintable ( imageList )
51
```

Listing 14-15 Running the System Property Definitions

The `mouseEvent` channel {27} must be defined with a data store of type `OverWriteOldestBuffer` so that the event handling thread associated with the user interface does not block; see the JCSP documentation for `ActiveCanvas`. Similarly the `pointToTC` channel also uses a one place `OverWriteOldestBuffer` {30} so that if mouse clicks are received too quickly the system does not block. Given the normal performance of a PC this is very unlikely to occur as the user time to move the mouse to another target is relatively long.

The channels that connect `TargetController` to the `TargetProcesses` are defined as an array, `mousePointToTP` {45}, the input end of which is passed directly to the `TargetProcess`. The output ends are formed into a `ChannelOutputList`, `mousePoints` {46}, so that they can be written to in parallel by a `write` method call by `TargetController`.

The `DisplayList` and `ActiveCanvas` components are defined {48–50} prior to being passed as properties of the required processes.

Listing 14-16 shows the definition of the `TargetProcess`es and also of `BarrierManager`. The other processes can be found in the accompanying software because they are very similar to the definition of processes in other systems. It is a matter of tying together the property definition in the process and the defined variable in the script that causes the system to execute. The barriers are straightforward but the allocation of a `timeAndHitBarrier` requires that a specific front-end is allocated to each `TargetProcess` {60} and also to `BarrierManager` {70}. The origin co-ordinates of each `TargetProcess` {63, 64} for the associated display is obtained from the list `targetOrigins`.

```
52  def targetList = ( 0 ..< targets ).collect { i ->
53                  return new TargetProcess (
54                       targetRunning: targetIdToManager.out(),
55                       stateToDC: targetStateToDC.out(),
56                       mousePoint: mousePointToTP[i].in(),
57                       setUpBarrier: setUpBarrier,
58                       initBarrier: initBarrier,
59                       goBarrier: goBarrier,
60                       timeAndHitBarrier: timeAndHitBarrier[i],
61                       buckets: buckets,
62                       targetId: i,
```

Download free eBooks at bookboon.com

```
63                          x: targetOrigins[i][0],
64                          y: targetOrigins[i][1],
65                          delay: delay
66                          )
67                     }
68
69  def barrierManager = new BarrierManager (
70                          timeAndHitBarrier: timeAndHitBarrier[targets],
71                          finalBarrier: finalBarrier[0]  ,
72                          goBarrier: goBarrier,
73                          setUpBarrier: setUpBarrier
74                          )
75
```

Listing 14-16 Decalring the TargetProcesses and BarrierManager

14.11 Summary

This chapter has introduced the concepts of buckets and barriers as a means of providing synchronisation between processes that are executing on a single processor within a single JVM. It has been shown how an `AltingBarrier` can be used to manage highly dynamic situations and to provide a high-level control mechanism to manage complex interactions. A description of the implementation mechanism underlying `AltingBarrier` is to be found in (Welch, et al., 2007) and a different use of `AltingBarrier` using a syntactically different but conceptually identical formulation is to be found in (Ritson & Welch, 2007)

Download free eBooks at bookboon.com