6 Testing Parallel Systems: First Steps

JUnit [ref] testing has become a widely accepted way of testing Java classes and there is a great deal of software support for this approach. In the previous chapter examples and exercises were introduced whereby the user had to ascertain for them self that the systems worked in the expected manner. This was achieved by looking at printed output. This may be a satisfactory approach for small example systems but is not appropriate for systems that are to be used in an every day context.

In this chapter the use of JUnit testing is introduced by using examples taken from earlier chapters. This will demonstrate that it is possible to use this approach and give a general architecture for testing parallel systems. The key to JUnit testing is that we test one or more assertions concerning the underlying implementation. In the parallel situation we have to identify a source of inputs that can be compared to the subsequent outputs for the assertion testing.

6.1 Testing Hello World

The testing of the `ProduceHW` and `ConsumeHW` processes (see Chapter 2) demonstrate that from the outset testing has to be considered at the time processes are designed and cannot be retrospectively added. To this end, properties are required that can be accessed once a process has terminated. These properties can then become components in any assertion. In this very simple case the `ProduceHW` process needs no alteration.

6.1.1 Revised ConsumeHW Process

The revised version of `ConsumeHW`, see Listing 6-1 requires the addition of a property `message` {3}, which is assigned {7} the values that have been read in from `inChannel` {5, 6}.

```
01    class ConsumeHW implements CSProcess {

02      def ChannelInput inChannel
03      def message

04      void run() {
05        def first = inChannel.read()
06        def second = inChannel.read()
07        message = "${first} ${second}!!!"
08        println message
09      }
10    }
```

Listing 6-1 The Revised Version of ConsumeHW

6.1.2 The HelloWorldTest Script

Listing 6-2 gives the script used to test ProduceHW and ConsumerHW.

```
11      import c2.ProduceHW

12      class HelloWorldTest extends GroovyTestCase {

13              void testMessage() {

14                      One2OneChannel connect = Channel.createOne2One()

15                      def producer =  new ProduceHW ( outChannel: connect.out() )
16                      def consumer = new ConsumeHW ( inChannel: connect.in() )

17                      def processList = [ producer, consumer ]

18                      new PAR (processList).run()

19                      def expected = "Hello World!!!"
20                      def actual = consumer.message

21                      assertTrue(expected == actual)
22              }
23      }
```

Listing 6-2 The HelloWorldTest Script

The `ProduceHW` process from Chapter 2 is imported {11}. The remainder of the coding is that required to build an instance of Groovy`TestCase`. This requires that a `void` method is defined, the name of which is prefixed with the word `test` and that contains the script necessary to run the processes being tested. The primary requirement is that in order to test the system the processes must terminate.

The crucial elements are that we define each process as an instance {15, 16}. This is required so that we can access the `message` property of `ConsumeHW` when the system terminates. The processes are then run in parallel {17, 18}. The property `expected` is set to the `String` that should be output {19}. The `actual` value is obtained from the `message` property of `ConsumeHW` {20}. These are then compared {21} using the `assertTrue` method which produces an indication of whether the test passed. Typical JUnit style test output is shown in Output 6-1.

```
.Hello World!!!

Time: 0.281

OK (1 test)
```

Output 6-1 Output Generated by the Test of Hello World

6.2 Testing the Queue Process

The `Queue` Process discussed in Chapter 5.2 can be tested by sending a known number of test values into the `Queue` from the `QProducer` process and then ensuring that the same number of values is received by the `QConsumer` process. A more thorough test would be to ensure that the sequence of numbers received by `QConsumer` is the same as that sent by the `QProducer`, but that is left as an exercise for the reader. Listing 6-3 shows the modified `QProducer` process. The only modifications required occur on {28}, where a new `List` property is added called `sequence`, which holds the sequence of produced values and on {34} where each produced value is appended (<<) to `sequence`. The printing of the produced values also has been removed. The `sequence` property is required to ensure we have a value that can be tested once the network of processes being tested has terminated.

```
24     class QProducer implements CSProcess {

25       def ChannelOutput put
26       def int iterations = 100
27       def delay = 0
28       def sequence = []

29       void run () {

30         def timer = new CSTimer()
31         for ( i in 1 .. iterations ) {
32           put.write(i)
33           timer.sleep (delay)

34           sequence = sequence << i

35         }
36         put.write(null)
37       }
38     }
```

Listing 6-3 The Testable Version of QProducer

Listing 6-4 shows the modified `QConsumer` process, which as in the `QProducer` defines a property that can be externally access, called `outSequence` {43}. The processing of the terminating `null` value has been modified {51-55} so that all the received values are appended to `outSequence` unless it is the `null` value, in which case the value of `running` is set `false`, causing the process to terminate.

```
39     class QConsumer implements CSProcess {

40       def ChannelOutput get
41       def ChannelInput receive
42       def long delay = 0
43       def outSequence = []

44       void run () {

45         def timer = new CSTimer()
46         def running = true

47         while (running) {
48           get.write(1)
49           def v = receive.read()
50           timer.sleep (delay)

51           if ( v != null) {
52             outSequence = outSequence << v
53           } else {
54              running = false
55           }

56         }
57       }
58     }
```

Listing 6-4 The Modified QConsumer Process

6.3 The Queue Test Script

Listing 6-5 gives the class that causes the `Queue` process testing. It takes the same basic structure as that used in the test of the Hello World system. It is important to note that the aim is to test the `Queue` process given in Chapter 5 {59} and not the `QProducer` and `QConsumer` processes. The channels required to implement the network defined in Figure 5-2 are specified {62-64}. Instances of each of the processes are then defined {65-68}, so that we can subsequently test the values of the properties `sequence` and `outSequence` of `QProducer` and `QConsumer` respectively. The list of processes is then defined and executed in a `PAR` {69-70}, which must terminate if we are to be able to test the process properties in an assertion {73}.

```
59     import c5.Queue

60     class QueueTest extends GroovyTestCase {

61       void testQueue() {

62         One2OneChannel QP2Q = Channel.createOne2One()
63         One2OneChannel Q2QC = Channel.createOne2One()
64         One2OneChannel QC2Q = Channel.createOne2One()

65         def qProducer = new QProducer ( put: QP2Q.out(), iterations: 50 )
66         def queue     = new Queue ( put: QP2Q.in(), get: QC2Q.in(),
67                                     receive: Q2QC.out(), elements: 5)
68         def qConsumer = new QConsumer ( get: QC2Q.out(), receive: Q2QC.in() )

69         def testList = [ qProducer, queue, qConsumer ]
70         new PAR ( testList ).run()

71         def expected = qProducer.sequence
72         def actual = qConsumer.outSequence

73         assertTrue(expected == actual)
74       }
75     }
```

Listing 6-5 The QueueTest Script

The values of the `expected` and `actual` returned values are obtained from their processes and tested {71-72}. This will cause the result of the test to be printed on the console as shown in Output 6-2. In more complex examples the construction of assertions is likely to be more elaborate depending upon the nature of the data being input and generated.

```
.Q finished

Time: 0.266

OK (1 test)
```

Output 6-2 The Output From the Queue Test

6.4 Summary

In this chapter we have introduced the concept of testing parallel systems, using the JUnit testing framework within a Groovy environment. The key requirement is that the network of processes must terminate. Further, the processes used to test the operation of the process network under test must contain properties that can be populated with data that can then be tested in an assertion. In Chapter 10 we reflect further on the testing of parallel systems and show how we can test systems that are designed not to terminate.

6.5 Exercises

1. Construct a Test Case for the Three-To-Eight system constructed in the exercise for Chapter 2.

6.