2 Producer Consumer: A Fundamental Design Pattern

This chapter provides an introduction to

- a simple producer – consumer design pattern
- shows how a set of processes can be invoked using the PAR helper class
- shows how processes and channels interact with one another
- demonstrates the ease with which processes can be reused

For many people, the first program they write in a new language is to print "Hello World", followed by the inputting of a person's name so the program can be extended to print "Hello *name*". In the parallel world this is modified to a program that captures one of the fundamental design patterns of parallel systems, namely a Producer – Consumer system.

A Producer process is one that outputs a sequence of distinct data values. A Consumer process is one that inputs a stream of such data values and then processes them in some way. The composition of a Producer and Consumer together immediately generate some interesting possibilities for things to go wrong. What happens if?

the Producer process is ready to output some data before the Consumer is ready or

the Consumer process is ready to input but no data is available from the Producer

In many situations, the programmer would resort to introducing some form of buffer between the Producer and Consumer to take account of any variation in the execution rate of the processes. This then introduces further difficulties in our ability to reason about the operation of the combined system; such as the buffer becomes full so the Producer has to stop outputting, or conversely it becomes empty and the Consumer cannot input any data. We have just put off the decision. In fact, we have made it much harder to both program the system and to reason about it. In addition, we now have to consider the situation when the buffer fails for some reason. Fortunately, the definitions of process and channel given in Chapter 1 come to our rescue.

If the Producer process is connected to the Consumer process by a channel then we know that the processes synchronise with each other when they transfer data over the channel. Thus if the Producer tries to output or write data before the Consumer is ready to input or read data then the Producer waits until the Consumer is ready and vice-versa. It is therefore impossible for any data to be lost or spurious values created during the data communication.

2.1 A Parallel Hello World

2.1.1 Hello World Producer

The producer process for Hello-World is shown in Listing 2-1. Line {10–20} defines the class `ProduceHW` that implements the interface `CSProcess`, which defines a single method `run()` that is used to invoke the process. The interface `CSProcess` is contained in the package `org.jcsp.lang,` which has to be imported (not shown).

```
10 class ProduceHW implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run() {
15     def hi = "Hello"
16     def thing = "World"
17     outChannel.write ( hi )
18     outChannel.write ( thing )
19   }
20 }
```

Listing 2-1 Hello World Producer Process

The only class property, `outChannel` {12}, of type `ChannelOutput`, is the channel upon which the process will output using a `write()` method. Strictly, Groovy does not require the typing of properties, or any other defined variable, however, for documentation purposes we adopt the convention of giving the type of all properties. This also has the advantage of allowing the compiler to check type rules and provides additional safety when processes are formed into process networks. Each process has only one method, the `run()` method as shown starting at line {14}. Two variables are defined {15, 16}, `hi` and `thing`, that hold the strings "Hello" and "World" respectively. These are then written in sequence to `outChannel` {17, 18}.

2.1.2 Hello World Consumer

The `ConsumeHello` process, Listing 2-2, has a property `inChannel` {12} of type `ChannelInput`. Such channels can only input objects from the channel using a `read()` method. Its `run()` method firstly, reads in two variables, `first` and `second`, from its `inChannel` {15, 16}, which are then printed {17} to the console window with preceding and following new lines (\n). The notation `$v` indicates that the variable `v` should be evaluated to its `String` representation

```
10 class ConsumeHello implements CSProcess {
11
12   def ChannelInput inChannel
13
14   void run() {
    def first = inChannel.read()
    def second = inChannel.read()
    println "\n$first $second!\n"
  }
}
```

Listing 2-2 Hello World Consumer Process

2.1.3 Hello World Script

Figure 2-1 shows the process network diagram for this simple system comprising two processes `ProduceHW` and `ConsumeHello` that communicate over a channel named `connect`.

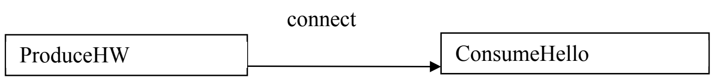

Figure 2-1 Producer Consumer Process Network

The script, called `RunHelloWorld` for executing the processes `ProduceHW` and `ConsumeHW` is shown in Listing 2-3. It is this script that is invoked to execute the process network.

```
10 def connect = Channel.one2one()
11
12 def processList = [
13                         new ProduceHW ( outChannel: connect.out() ),
14                         new ConsumeHello ( inChannel: connect.in() )
15                     ]
16 new PAR (processList).run()
```

Listing 2-3 Hello World Script

An imported package `org.jcsp.lang` contains the classes required for the JCSP library. Another package `org.jcsp.groovy` contains the definitions of the Groovy parallel helper classes. The referenced libraries and documentation contain a more complete specification and description of their use. The `PAR` class {16} causes the parallel invocation of a list of processes. This is achieved by calling the `run()` method of `PAR`, which in turn causes the execution of the `run()` method of each of the processes in the list.

The channel `connect` is of type `Channel.one2one` {10}. The channel is created by means of a static method `one2one` in the class `Channel` contained within the package `org.jcsp.lang`. The `processList` {12} comprises an instance of `ProducerHW` with it's `outChannel` property set to the `out` end of `connect`, and the processes `ConsumerHW` with its `inChannel` property set to the `in` end of `connect` {13, 14}.

The underlying JCSP library attempts, as far as possible, to ensure networks of processes are connected in a manner that can be easily checked. The channel `connect` {10} is defined to have a `one2one` interface and therefore it has one output end and one input end. These are defined by the methods `out()` {13} and `in()` {14} respectively. A class that contains a property of type `ChannelOutput` must be passed an actual parameter that has been defined with a call to `out()` and within that process only `write()` method calls are permitted on the channel property. The converse is true for input channels. In all process network diagrams the arrow head associated with a channel will refer to the input end of the channel.

The output from executing this script is shown in Output 2-1.

```
Hello World!
```

Output 2-1 Output from Hello World Script

2.2 Hello Name

The Hello Name system is a simple extension of the Hello World system. The only change is that the `ProducerHN` {10} process asks the user for their name and then sends this to the `Consumer` process as the `thing` variable {16, 18} in Listing 2-4.

```
10 class ProduceHN implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run() {
15     def hi = "Hello"
16     def thing = Ask.string ("\nName ? ")
17     outChannel.write ( hi )
18     outChannel.write ( thing )
19   }
20 }
```

Listing 2-4 The ProduceHN Process

An imported package `phw.util` contains some simple console interaction methods that can be used to obtain input from the user from the console window. The `Ask.string` method outputs the prompt "`Name ?`" after a new line and the user response is then placed into the variable `thing` {16}.

The Consumer process remains unaltered from the version shown in Listing 2-2. Similarly, the script to run the processes is the same as Listing 2-3 except that the name of the producer process has been changed to `ProduceHN`. A typical output from the execution of the script is shown in Output 2-2, where user typed input is shown in *italics*. This also shows how easy it is to reuse a process in another network.

```
Name ? Jon
Hello Jon!
```

Output 2-2 Output from Hello Name Network

2.3 Processing Simple Streams of Data

The final example in this chapter requires the user to type in a stream of integers into a producer process that writes them to a consumer process, which then prints them. The specification of the `Producer` process is given in Listing 2-5.

```
class Producer implements CSProcess {

  def ChannelOutput outChannel

  void run() {
    def i = 1000
    while ( i > 0 ) {
      i = Ask.Int ("next: ", -100, 100)
      outChannel.write (i)
    }
  }
}
```

Listing 2-5 The Producer Process

The `run()` {14} method is formulated as a `while` loop {16–19}, which is terminated as soon as the user inputs zero or negative number. The input integer value is obtained using the `Ask.Int` (from phw.util) method that will ensure that any input lies between -100 to 100 {17}. The `while` loop has been structured to ensure the final zero is also output to the `Consumer` process.

```
10 class Consumer implements CSProcess {
11
12   def ChannelInput inChannel
13
14   void run() {
15     def i = 1000
16     while ( i > 0 ) {
17       i = inChannel.read()
18       println "the input was : $i"
19     }
20     println "Finished"
21   }
22 }
```

Listing 2-6 The Consumer Process

The Consumer process is shown in Listing 2-6. The `Consumer` {10} process reads data {17} from its input channel, `inChannel` {12}, which is then printed {18}. Once a zero is read the `while` loop {16–19} terminates resulting in the printing of the "`Finished`" message {20}.

The script, called `RunProducerConsumer` that causes the execution of the network of processes is shown in Listing 2-7 which is very similar to the previous script shown in Listing 2-3, the only change being, the names of the processes that make up the list of processes {12, 13}.

```
10  def connect = Channel.one2one()
11
12  def processList = [ new Producer ( outChannel: connect.out() ),
13                      new Consumer ( inChannel: connect.in() )
14                    ]
15  new PAR (processList).run()
```

Listing 2-7 The Producer Consumer System Script

Output from a typical execution of the processes is given in Output 2-3.

```
next: 1
next: the input was : 1
2
the input was : 2
next: 3
the input was : 3
next: 0
the input was : 0
Finished
```

Output 2-3 Typical Results from Producer Consumer System

The output, especially when executed from within Eclipse, can be seen to be correct but the output is somewhat confused as we have two processes writing concurrently to a single console at the same time; both processes use `println` statements. We therefore have no control of the order in which outputs appear and these often become interleaved. A process called GConsole is available in the package `org.jcsp.groovy.plugAndPlay` that creates a console window with both an input and output area. This process can be used to provide a specific console facility for a given process. Many such GConsole processes can be executed in a network of processes as required. Its use will be demonstrated in later chapters.

2.4 Summary

This chapter has introduced the basic use of many simple JCSP concepts. A set of simple Producer – Consumer based systems have been implemented and output from these systems has also been given. These basic building blocks of processes and channels, with their simple semantics are the basis for all the concurrent and parallel systems we shall be building throughout the rest of this book.

2.5 Exercises

Exercise 2-1

Using Listing 2-7 as a basic design implement and test a new process called `Multiply` that is inserted into the network between the `Producer` and `Consumer` processes which takes an input value from the `Producer` process and multiplies it by some constant `factor` before outputting it to the `Consumer` process. The multiplication `factor` should be one of the properties of the `Multiply` process. To make the output more meaningful you may want to change the output text in the `Consumer` process. You should reuse the `Producer` process from the `ChapterExamples` project in `src` package `c2`.

Exercise 2-2

A system inputs data objects that contain three integers; it is required to output the data in objects that contain eight integers. Write and test a system that undertakes this operation. The process `ChapterExercises/src/c2.GenerateSetsOfThree` outputs a sequence of `List`s, each of which contains three positive integers. The sequence is to be terminated by a `List` comprising [-1, -1, -1].

What change is required to output objects containing six integers? How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12)? What happens if the number of integers required in the output stream is not a factor of the total number of integers in the input stream (e.g. 5 or 7)?