**Lab Book – Glenn Wilkie-Sullivan (40208762)**

***Exercise 2-1:***

(Multiplier Class)

// write i \* factor to outChannel

outChannel.write(i \* factor);

// read in the next value of i

i = inChannel.read();

(Consumer Class)

//insert a modified println statement

println ("New Value: " + i)

i = inChannel.read()

(RunMultiplier Class)

//insert here an instance of multiplier with a multiplication factor of 4

**new** Multiplier ( inChannel : connect1.in(),

outChannel : connect2.out() ),

Pipeline Diagram:



Output:



Network Operation Commentary:

Using this network is fairly trivial – upon running, the user can enter numbers into the console, and the program will return that number multiplied by a factor, ready for another input.

***Exercise 2-2:***

(GenerateSetsOfThree Class)

//write the terminating List as per exercise definition

outChannel.write([-1, -1, -1])

(ListToStream Class)

// hint: output list elements as single integers

**for**(i **in** 0 ..< inList.size)outChannel.write(inList[i])

inList = inChannel.read()

(CreateSetsOfEight Class)

**for** ( i **in** 0 .. 7 ) {

// put v into outList and read next input

outList.add(v);

v = inChannel.read();

}

println " Eight Object is ${outList}"

outList.clear();

Pipeline Diagram:



Output:



**Question 1 (What change is required to output objects containing six integers?)**

Within our “CreateSetsOfEight” class, we read each element of the incoming tuples and add them to the outlist, which has the given range 0-7. Simply changing this to 0-5 generates list of six.

**Question 2 (How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12) ?)**

As before, we change the range parameter to whatever size of tuple we require.

**Question 3 (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) ?)**

If a new list of full size cannot be created, the remainder of integers are added to the previous list.

***Exercise 3-2:***

(GSCopy Class)

// output the input value in sequence to each output channel

outChannel0.write(i)

outChannel1.write(i)

(GSquares Class)

**new** GSPairsA (inChannel: I2P.in(),

outChannel: outChannel)

**new** GSPairsB (inChannel: I2P.in(),

outChannel: outChannel)

Piepline Diagram (FOR GSPairsA & GSPairsB):



Output (FOR GSPairsB):



**Question 1 (Replace GPairsB with GPairsA and determine the effect of the change. Why does this happen?)**

GTail only outputs what is read into it. As channels a/b are switched, GPlus then expects an input of channels b/c, but is instead fed an input of a/c.

***Exercise 3-3:***

**Question 1 (Why was it considered easier to build GParPrint as a new process rather than using multiple instances of GPrint to output a table of results?)**

GParPrint contains logic to do all the printing and formatting at once, which is considerably faster than instantiating two GPrints, which will have to be formatted into columns, spaced, etc. which takes a lot of time.

***Exercise 3-4:***

(Minus Class)

// output one value subtracted from the other

// be certain you know which way round you are doing the subtraction!!

outChannel.write(read1.value - read0.value)

(Negator Class)

//output the negative of the input value

**def** i = inChannel.read()

outChannel.write(i \*= -1)

(Differentiate Class)

// insert a constructor for Minus

**new** Minus ( inChannel0: c.in(),

inChannel1: a.in(),

outChannel: outChannel)

(DifferentiateNeg Class)

//insert a constructor for Negator

**new** Negator ( inChannel: c.in(),

outChannel: d.out()),

Pipeline Diagram (Minus):



Pipeline Diagram (Negator):



Output (Both Functions):



***Exercise 4-1:***

If one reset value is input, the program will use that value but doesn’t increment it. Thus, the reset value and prefix value are output side by side. However, if the user inputs multiple values, the program deadlocks and stops the output.

***Exercise 4-2:***

(ResetSuccessor Class)

// deal with inputs from resetChannel and inChannel

// use a priSelect

**def** i = alt.priSelect()

**if** (i.value == 0) {

**def** j = resetChannel.read()

inChannel.read()

outChannel.write(j + 1)

} **else** {

**def** k = inChannel.read()

outChannel.write(k + 1)

}

(ResetNumbers Class)

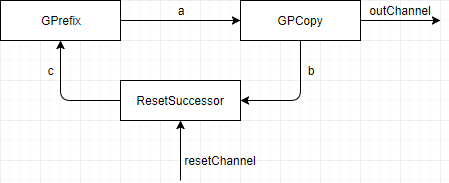
// requires a constructor for ResetSuccessor

**new** ResetSuccessor ( inChannel: b.in(),

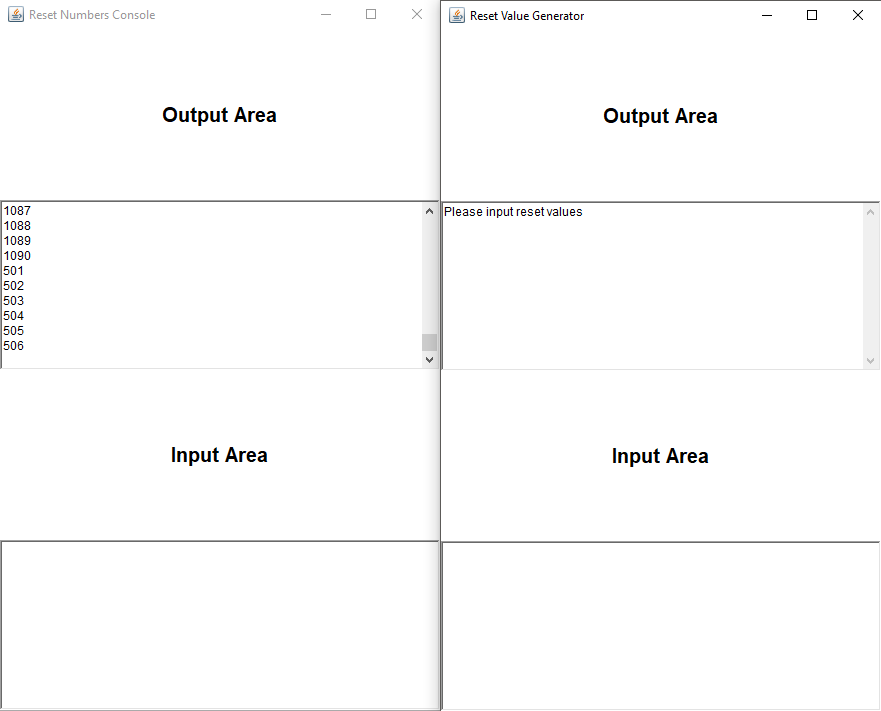
outChannel: c.out(),

resetChannel: resetChannel)

Pipeline Diagram:



Output:



**Associated Question (Does it overcome the problem identified in Exercise 1? If not, why not?)**

It doesn’t overcome the problem in exercise 1, as the functionality is almost identical except for the prefix functions are now in successor.

***Exercise 5-1:***

The delays do in fact work on the QProducer/Consumer process, as the program runs in variable time according to the value of the delay – the larger the delay, the longer the program takes to execute. It appears only one delay needs to be modified for the delay to work – as expected. This means there can either be a delay from values being written, or a delay from values being read. Upon further examination, I believe the delay is scaled from 1000 to 1. This means for each 1000 units added to the delay value, the producer/consumer will take an extra second to process a read or write signal. It also appears there is no limit on the delay – I tested from a 0 to 500,000,000-unit delay, but the program still ran (may be delimited by the size of primitive variables).

***Exercise 5-2:***

(Scale Class)

**case** SUSPEND:

// deal with Suspend input

preCon[SUSPEND] = **false**

preCon[INJECT] = **true**

preCon[INPUT] = **false**

suspend.read()

factor.write(scaling)

suspended = **true**

println "Stream Suspended"

**def** inValue = inChannel.read()

**def** result = **new** ScaledData()

result.original = inValue

result.scaled = inValue

outChannel.write(result)

**break**

**case** INJECT:

// deal with Inject input

preCon[SUSPEND] = **true**

preCon[INJECT] = **false**

preCon[INPUT] = **true**

scaling = injector.read()

println "Injected scaling is $scaling"

suspended = **false**

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm(timeout)

**break**

**case** TIMER:

// deal with Timer input

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm(timeout)

scaling = scaling \* 2

println "Normal Timer: new scaling is ${scaling}"

**break**

**case** INPUT:

// deal with Input channel

**def** inValue = inChannel.read()

**def** result = **new** ScaledData()

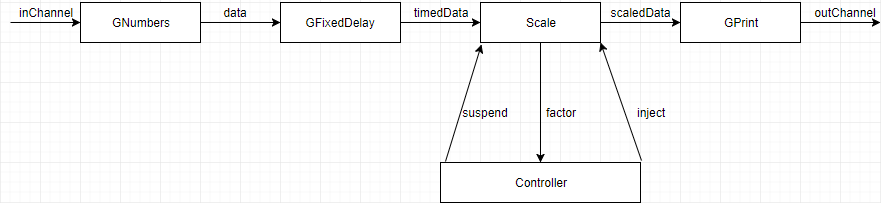
result.original = inValue

result.scaled = inValue \* scaling

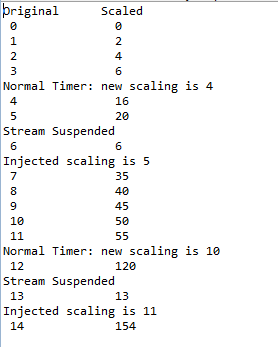
outChannel.write(result)

**break**

Pipeline Diagram:



Output:



**Associated Question (Which is the more elegant formulation? Why?)**

The precondition formulation is naturally more robust, or elegant, as it specifies which behaviours can be dealt with at which specific times during the course of the program running. It stops any unexpected input from happening, which could break the first case-statement formulation if the input is not dealt with correctly. The second formulation also allows more control over the variables effecting the eventual output, as there is more insight into the suspension and injection of the values.

***Exercise 6-1:***

(Amended ListToStream Class)

**def** ChannelInput inChannel

**def** ChannelOutput outChannel

**def** testList = []

**for**(i **in** 0 ..< inList.size) {

outChannel.write(inList[i])

testList = testList << inList[i];

}

(Amended CreateSetsOfEight Class)

**def** ChannelInput inChannel

**def** testList = []

**for** ( i **in** 0 .. 7 ) {

// put v into outList and read next input

outList.add(v);

testList = testList << v

v = inChannel.read();

}

(TestThreeToEight Class)

**class** TestThreeToEight **extends** GroovyTestCase {

**void** testMethod() {

One2OneChannel connect1 = Channel.*createOne2One*()

One2OneChannel connect2 = Channel.*createOne2One*()

**def** genSetOfThree = **new** GenerateSetsOfThree ( outChannel: connect1.out() )

**def** listStream = **new** ListToStream ( inChannel: connect1.in(), outChannel: connect2.out() )

**def** createSetOfEight = **new** CreateSetsOfEight ( inChannel: connect2.in() )

**def** testList = [genSetOfThree, listStream, createSetOfEight]

**new** PAR (testList).run()

**def** expected = listStream.testList

**def** actual = createSetOfEight.testList

**for** (i **in** 0..22) {

*assertTrue*(expected == actual)

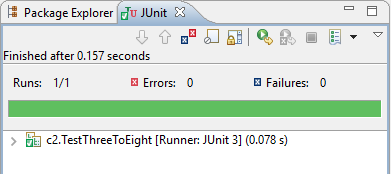
*assertFalse*(expected != actual)

}

}

}

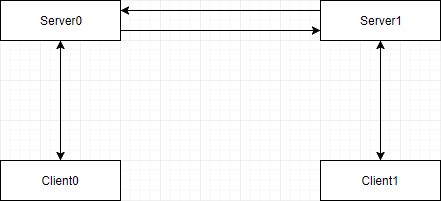
Output:



***Exercise 7-1:***

The deadlock lies when two pairs of identical numbers fall in parallel between the two clients. The first working system can run smoothly because it robustly deals with one pair of identical numbers being sent at the same time, and can rely on the next input, but upon receiving two pairs of identical numbers (the server tries to rely on another identical pair) the system deadlocks as the server has to go between requesting identical numbers for two pairs instead of one, which can be dealt with.

Pipeline Diagram:



***Exercise 8-1:***

(Amended Client Class)

**class** Client **implements** CSProcess {

**def** ChannelInput receiveChannel

**def** ChannelOutput requestChannel

**def** clientNumber

**def** selectList = [ ]

**def** propertyOrder = **false**

**void** run () {

**def** iterations = selectList.size

println "Client $clientNumber has $iterations values in $selectList"

**for** ( i **in** 0 ..< iterations) {

**def** key = selectList[i]

requestChannel.write(key)

**def** v = receiveChannel.read()

println "Key: " + key

println "V: " + v

**if** (v == key \* 10) {

propertyOrder = **true**;

}

}

println "Client $clientNumber has finished"

**if** (propertyOrder == **true**) {

println "Client $clientNumber in order"

}

**else** {

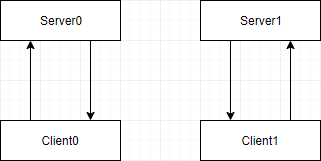
println "Client $clientNumber not in order"

}

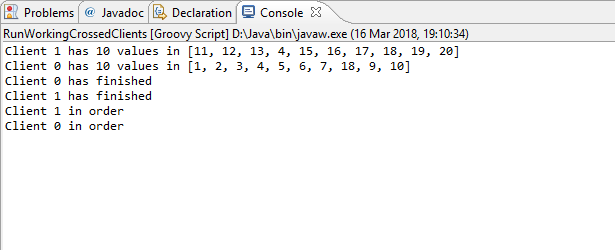
}

}

Pipeline Diagram:



Output:



***Exercise 9-1:***

(EventFix Class)

**class** EventFix **implements** CSProcess {

**boolean** passed = **true**

**def** ChannelInput inChannel

**def** ChannelOutput outChannel

**void** run() {

**while** (**true**) {

**def** eventData = inChannel.read().copy()

**if** (eventData.data != 100 && eventData.data != eventData.prev +

eventData.missed + 1)

{

println "Missed"

}

outChannel.write(eventData)

}

}

}

(New EventData Class)

**class** EventData **implements** Serializable, JCSPCopy {

**def** **int** source = 0

**def** **int** data = 0

**def** **int** missed = -1

**def** **int** prev

**def** copy() {

**def** e = **new** EventData ( source: **this**.source,

data: **this**.data,

missed: **this**.missed,

prev: **this**.prev )

**return** e

}

**def** String toString() {

**def** s = "EventData -> [source: "

s = s + source + ", data: "

s = s + data + ", missed: "

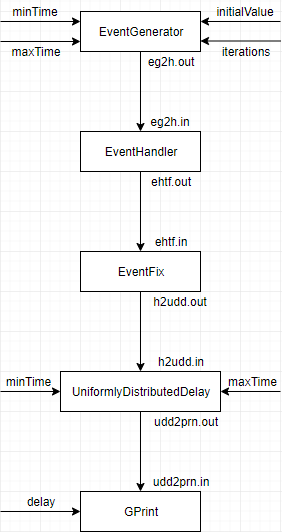
s = s + missed + "]"

**return** s

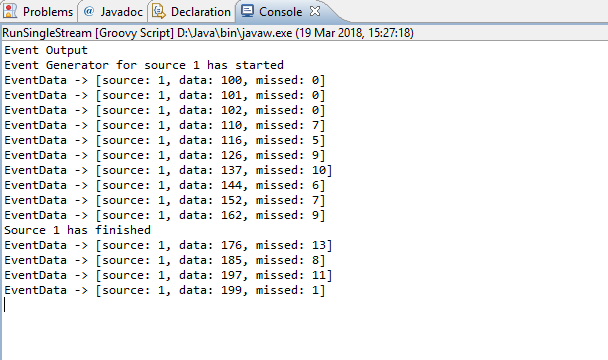
}

}

Pipeline Diagram:



Output:



***Exercise 9-2:***

The times associated with the event generation stream seem almost directly proportional to the number of missed events. i.e, if the minimum and maximum times are increased, more events are missed – but if the times are decreased, less events are missed. By modifying the times of the processing system, increasing the times doesn’t seem to have much effect, but if the times are decreased, the system misses more events as time goes on, and the sources take much longer to finish. From this, we can assume that decreasing the time on the processing system is directly linked to how long the sources take to send/receive data.

***Exercise 9-3:***

Out of the three multiplexer, the basic multiplexer seems to be the slowest – it has an effect on each source, and seems to miss a lot of data (albeit the least of the three). The ‘PriMultiplex’ isn’t much faster, but seems to miss massive chunks of data at a time each time, possibly to compensate for a lack of processing speed. The ‘FairMultiplex’ doesn’t miss nearly as much data as the PriMultiplex, and seems to be a bit faster than the basic multiplexer.