11 Graphical User Interfaces: Brownian Motion

This chapter introduces the concept of active user interface components

- active components are implemented by a process
- they have channel interfaces
- an alternative can be used to determine which component is ready for interaction
- the user interface has a declarative style with little or no coding for the interactions
- the application demonstrates how animation can be achieved
- the use of any2one and one2any channels is explained and justified

Previously, a simple user interface (`GConsole`) has been used that enables easier interpretation of the output from process networks. This chapter explores more complex user interfaces in conjunction with a relatively simple graphical application based upon particle movement.

The JCSP package contains an active implementation of the Java AWT (Abstract Windows Toolkit). The term active is here used to mean that each AWT component, for example, button, scrollbar and canvas, has been wrapped in a process so that component events and configuration are undertaken by channel communications. This means that the active components can be connected to any process. Furthermore, the programmer does not have to write any event handling or listener methods as these are contained within the active process wrapper. The active components inherit capabilities from the basic AWT components, thus the methods and fields associated with the component can be reused and active and ordinary, non-active, components can be used in the same interface.

The primary benefit of the active AWT components is that processes that access the user interface can utilise their non-deterministic capabilities, thereby reflecting the unpredictable behaviour of user interfaces. The user interface has no knowledge of when, for example, a button is going to be pressed and thus either a channel communication or an alternative provides a simple method for capturing that non-deterministic behaviour.

11.1 Active AWT Widgets

The fundamental process diagram for an active widget is shown in Figure 11-1. A widget is any component available in the `java.awt` package for which an active version has been constructed. Some active widgets have been constructed that simplify the construction of user interfaces. Specific widgets may have more or less channels depending upon the functionality of the widget.

All widgets have a configure input channel which enables the configuration of the widget at run-time. In most cases the configuration of a new widget can be defined when it is constructed, unless of course the content of the user interface is to be altered by changing the configuration of one of its widgets. For example, when a button has its associated text changed to reflect the state of the user interface. Each of the active component output channels produces data values that are related to the underlying AWT specification of that event and is specified in the `java.awt` documentation. The role of the configuration and event channels is specified in the `org.jcsp.awt` documentation and depends upon the specific component. For example, if the event arises from the pressing of an `ActiveButton` then the message communicated is the text string associated with the button. Similarly, a configuration channel message could be a text string that is to replace the current text associated with the button.

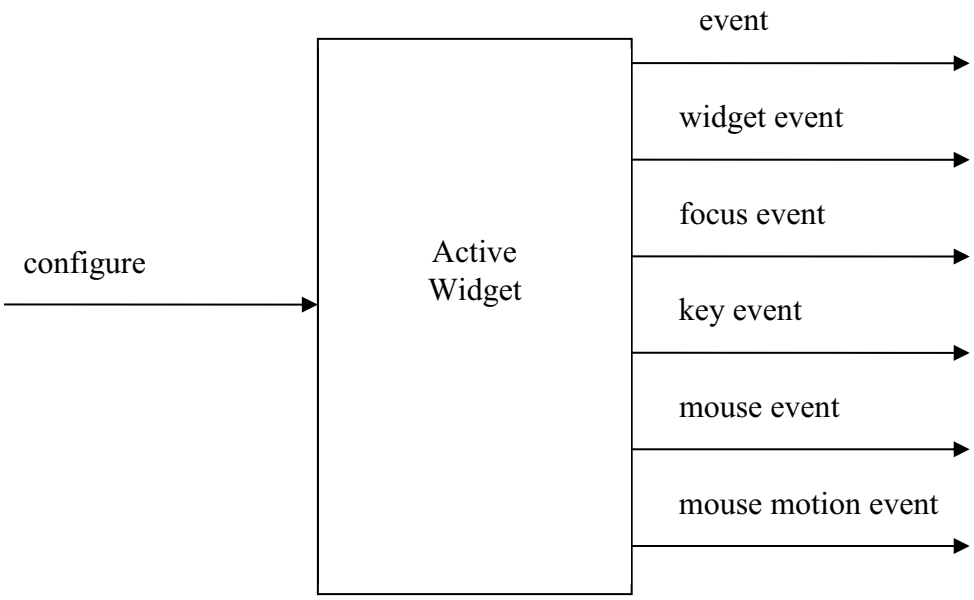


Figure 11-1 Generic Active Widget Process Diagram

11.2 The Particle System – Brownian Motion

A particle motion system (Lea, 2003) comprises a number of particles that move around at random. Their position is shown on a `Canvas`. Using Java `thread`s and a `Canvas` results in a somewhat cumbersome representation of the solution because a `Canvas` executes in its own thread of control, which has the effect of distributing the particle control, random movement and the graphical representation throughout the classes that make up the solution.

In the parallel solution that follows (see Figure 11-2) these drawbacks are eliminated and the fact that a `Canvas` has to execute in its own thread of control is hidden from the programmer. Furthermore in this solution we shall introduce some additional capabilities. The particles will bounce off the side of the bounding `Canvas`. The user will be given control of the application with a button that allows them to initially start the system and then subsequently to pause and resume its operation. In addition two buttons are provided which modify the 'temperature' of the system. The higher the temperature the greater the random movement exhibited by the particles. The particles do not bounce off each other and that is left as an additional exercise for the interested reader.

A number of particles (`Particle` 0 to n) are connected to the `ParticleInterface`. This utilises a new form of channel called `any2one`. An `any2one` channel enables the connection of any number of writer processes to a single reader process. The point-to-point nature of channel communication is, however, still maintained because only one communication can proceed at a time. Communications on an `any2one` are such that communication from one writer to the single reader is completed before the next writer can commence its communication. The converse is true of `one2any` channels. The JCSP library also includes `any2any` channels where yet again once a communication has started it behaves like a one-to-one communication.

`Particle`s are not aware of their position relative to the sides of the bounding `Canvas` and thus the particle may move to a position that is out with the bounding `Canvas`. In this case the particle's position is updated within the `ParticleInterface`. The updated position together with any change of temperature is returned to the `Particle` using the update channel. The update channel is a `one2any` channel that permits one writer to write to any number of readers. This is not a broadcast communication because the writer can only write to one of the reader processes at any one time. Furthermore, once one of the many reader processes has committed to a communication no other reader will be able to start a communication until the writer has written to that reader process.

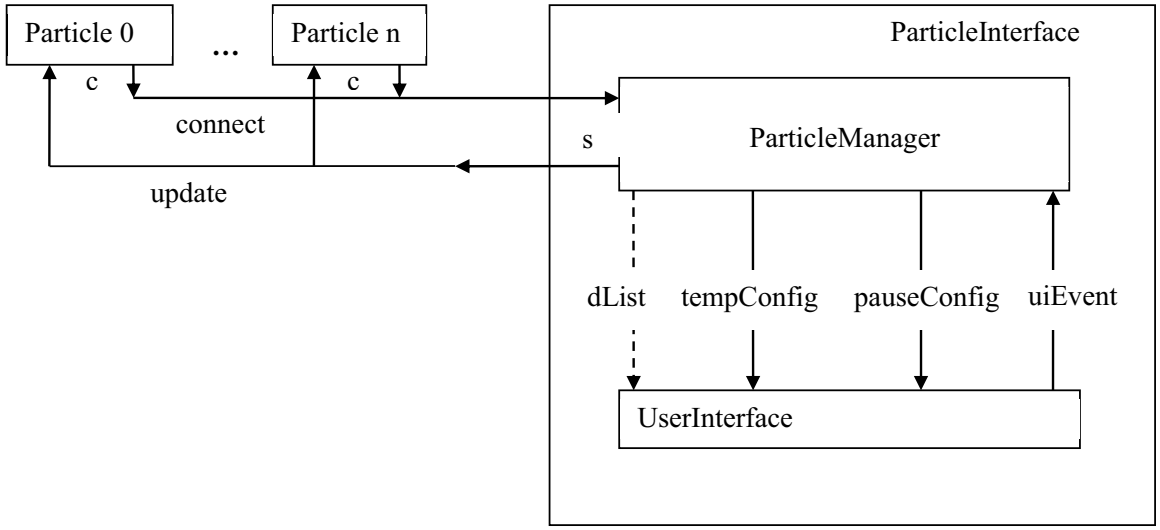


Figure 11-2 Brownian Motion Process Network

The `ParticleManager` is responsible for receiving inputs from the `Particle` processes; modifying their position, should the indicated position lie outside the bounding canvas; and then causing the display of the particle's position on the canvas. The `ParticleManager` is also responsible for dealing with button events from the `UserInterface` and configuring the buttons and labels within the `UserInterface`. Data is passed between the `Particle` processes and the `ParticleManager` by means of a data object that contains both positional information as well as any change to the temperature.

The `UserInterface` contains the display canvas, together with a button that is used to initially start and then subsequently used to pause and restart the system. Two further buttons are provided that are used to increase or decrease the temperature together with a `Label` that shows the current temperature value with an indication of whether the last change was up or down. The channels used between the `ParticleManager` and the `UserInterface` will be described more fully in a later section.

11.2.1 The Position Data Object

The `Position` data object, see Listing 11-1, is used to communicate data between the Particles and the ParticleInterface.

`Position` implements the interface `JCSPCopy` {10}, which is defined within the `org.jcsp.groovy` package. It should be recalled that objects are passed between processes running on the same machine by means of an object reference. In some situations this could lead to the creation of a large number of newly created short-lived objects, which could lead to the calling of the automatic Java garbage collector. The calling of the garbage collector during a graphical display would interfere with the presentation. The abstract interface `JCSPCopy` defines a method called `copy()`, which can be used to generate a deep copy of an object.

Lines {12–17} define the properties of `Position`. The property `id` is the number of the `Particle`. The properties `lx` and `ly` are the newly calculated [x, y] position co-ordinates of the `Particle`. These co-ordinates may lie outside the display area. The properties `px` and `py` are the co-ordinates of the previous position of the particle. The property `temperature` maintains the current value of the temperature within the system. All the properties, apart from `id` can be altered within the `ParticleInterface`.

```
10 class Position implements JCSPCopy {
11
12   def int id // particle number
13   def int lx // current x location of particle
14   def int ly // current y location of particle
15   def int px // previous x location of particle
16   def int py // previous y location of particle
17   def int temperature // current working temperature
18
19   def copy() {
20     def p = new Position ( id: this.id,
21                            lx: this.lx, ly: this.ly,
22                            px: this.px, py: this.py,
23                            temperature : this.temperature )
24     return p
25   }
26
27   def String toString () {
28     def s = "[Position-> " + id + ", " + lx + ", " + ly
29     s = s + ", " + px + ", " + py
30     s = s + ", " + temperature + " ]"
31     return s
32   }
33 }
```

Listing 11-1 The Position Data Object

Lines {19–25} define the method `copy` required for the implementation of the interface `JCSPCopy`. For completeness, a `toString` method is defined {27–32} that can be used to output the contents of a `Position` object.

11.2.2 The Particle Process

The definition of the `Particle` process is shown in Listing 11-2.

```
10 class Particle implements CSProcess {
11
12   def ChannelOutput sendPosition
13   def ChannelInput getPosition
14   def int x = 100 // initial x location
15   def int y = 100 // initial y location
16   def long delay = 200 // delay between movements
17   def int id
18   def int temperature = 25 // in range 10 to 50
19
20   void run() {
21     def timer = new CSTimer()
22     def rng = new Random()
23     def p = new Position ( id: id, px: x, py: y, temperature: temperature )
24     while (true) {
25       p.lx = p.px + rng.nextInt(p.temperature) - ( p.temperature / 2 )
```

```
26        p.ly = p.py + rng.nextInt(p.temperature) - ( p.temperature / 2 )
27        sendPosition.write ( p )
28        p = ( (Position)getPosition.read() ).copy()
29        timer.sleep ( delay )
30      }
31    }
32 }
```

Listing 11-2 The Particle Process

A `Particle` has two channels one {12}, `sendPosition`, to output its `Position` to, and the other {13}, `getPosition`, to receive updated `Positions` from the `ParticleInterface`. It should be noted that even though these channels will eventually be implemented as `any2one` and `one2any` channels as far as the process is concerned these are just a `ChannelOutput` and `ChannelInput` respectively. The properties `x` {14} and `y` {15} hold the initial position of the particle. A default display area of 200 pixels is presumed and thus all particles start their movement from the centre of that area. The position of the particles will be recalculated after the interval specified by `delay` {16}, which is initially set to 200 milliseconds. Each `Particle` is given a unique identification `id` {17}. The initial `temperature` of the system is set at 25 {18} and can range from 10 to 50.

The `run` method defines a `CSTimer` called `timer` {21} and uses the Java provided random number generator mechanism, `Random ()` {22}. The variable `p` holds the `Position` of the particle and is constructed using the initial values held within the properties passed to the process {23}.

The main loop of the process {24-30} requires the calculation of the new position of the particle `lx` and `ly` that are stored in the variable object p {25, 26}. The calculation ensures that the particle moves in a space that surrounds the current location [`px`, `py`] by a square with a side of size `temperature`. The position `p` is then written to the `ParticleInterface` {27}. This is a `write` operation that is implemented on a shared `any2one` channel and thus the process will have to wait until any other outstanding communications have completed. An `any2one` channel is essentially fair in that the communications are placed in a queue of such communications.

The `Particle` process behaves like a client to the `ParticleInterface`'s server. As soon as it has written its position to the `ParticleInterface` it `read`s the updated position information {28} from the `getPosition` channel. The `getPosition` channel is implemented by means of a `one2any` channel and thus this client – server interaction has to be carefully considered. When the `sendPosition.write(p)` {27} communication is completed only this `Particle` process can be in that state because only one communication is permitted on an `any2one` channel. Hence the only process that will be in a position to undertake a `read` on the `getPosition` channel is this process. Hence we are assured that a `Particle` process that writes it's position to `ParticleInterface` will be the one to receive its response, even though we are using shared `any2one` and `one2any` channels.

Finally, the Particle process sleeps for the delay period {29} after which the loop is repeated until the user stops the application through the user interface. The user interface will cause the Particle processes to stop even though they are implemented using a non-terminating while-loop.

11.2.3 The Particle Interface Process

This process, shown in Listing 11-3 is typical of any application that uses a graphical user interface in that it comprises a process that undertakes both the interaction with the user interface and the rest of the system and the process that implements the user interface itself. These two processes are always run in parallel using communication channels to pass events and configuration information between the processes.

The channels `inChannel` {12} and `outChannel` {13} are used to connect this process to the Particle processes. Yet again this process definition does not need to be aware of the specific implementation of the channels actually used to connect the processes together. The property `canvasSize` {14} provides a default size for the display area. Similarly, properties are defined for the number of `particles` {15}, the `centre` of the display area {16} and the `intialtemp`(erature) {17} of the system.

The variable `dList` {20} is of type `DisplayList`, defined within `org.jcsp.awt`. The use of `dList` will be described later. It is sufficient to note, at this stage, that it is passed as a property to the `ParticleManager` process {28}. An `ActiveCanvas`, `particleCanvas` is defined {21} and then a call to its `setPaintable()` method is made that associates it with `dList` {22}. In this manner both `ParticleManager` and `UserInterface` can access `dList`, the former directly as a property and the other indirectly through `particleCanvas` {36}. Essentially, `dList` is a shared object between the processes but the user can only modify the `dList` in `ParticleManager` directly. Therefore a `DisplayList` object has to be defined before either of the processes that access it are defined. A DisplayList is the mechanism by which animation can be more easily achieved.

```
10 class ParticleInterface implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14   def int canvasSize = 100
15   def int particles
16   def int centre
17   def int initialTemp
18
19   void run() {
20     def dList = new DisplayList()
21     def particleCanvas = new ActiveCanvas()
22     particleCanvas.setPaintable (dList)
23     def tempConfig = Channel.one2one()
```

```
24      def pauseConfig = Channel.one2one()
25      def uiEvents = Channel.any2one( new OverWriteOldestBuffer(5) )
26      def network = [ new ParticleManager ( fromParticles: inChannel,
27                                            toParticles: outChannel,
28                                            toUI: dList,
29                                            fromUIButtons: uiEvents.in(),
30                                            toUIPause: pauseConfig.out(),
31                                            toUILabel: tempConfig.out(),
32                                            CANVASSIZE: canvasSize,
33                                            PARTICLES: particles,
34                                            CENTRE: centre,
35                                            START_TEMP: initialTemp ),
36                      new UserInterface ( particleCanvas: particleCanvas,
37                                          canvasSize: canvasSize,
38                                          tempValueConfig: tempConfig.in(),
39                                          pauseButtonConfig: pauseConfig.in(),
40                                          buttonEvent: uiEvents.out() )
41                      ]
42      new PAR ( network ).run()
43    }
44 }
```

Listing 11-3 The ParticleInterface Process

The `tempConfig` channel {23} is used to update the temperature display in the interface. The `pauseConfig` {24} channel is used to set the text in the `START/PAUSE/RESTART` button.

The `uiEvents` channel {25} passes button events from the `UserInterface` to the `ParticleManager` process. It is not possible to press two buttons at the same time hence we can use an `any2one` channel, which simplifies processing within the `ParticleManager` process. The parameter `OverWriteOldestBuffer (5)` specifies that this channel will use a buffer of 5 elements in which, should it become full the oldest element in the buffer will be overwritten. This buffer is required because it is essential that events on this channel are always read otherwise the underlying Java event thread may block, which would also have the effect of stopping the rest of the user interface. The specified buffer will always read an input, hence ensuring that the Java event thread will not block and that another process will always be able to read the last few events, five in this case, even if the reading process is slow.

The `network` {26–41} simply comprises the `ParticleManager` and `UserInterface` processes with parameters and variables passed as parameters as required to construct the process network as shown in Figure 11-2.

11.2.4 The ParticleManager Process

The properties of the `ParticleManager` process are shown in Listing 11-4. The channel connections with `Particle` processes are provided by the channels `fromParticles` {12} and `toParticles` {13}. When the system is instantiated these will be passed shared channels of type `any2one` and `one2any` respectively. The constant properties {15–18} respectively contain the size of the square display area (`CANVASSIZE`), number of particles (`PARTICLES`), the centre co-ordinate of the display area (`CENTRE`) and the initial value of the system temperature (`START_TEMP`). The `DisplayList` property {14}, `toUI`, provides the graphical connection between the `ParticleManager` and `UserInterface` processes. The `ChannelInput` {19}, `fromUIButtons`, is the channel by which button events from the user interface are communicated to `ParticleManager`. Finally, the `ChannelOutputs` `toUILabel` {20} and `toUIPause` {21} provide the means by which the temperature value and the `START`, `PAUSE` and `RESTART` button have their values changed.

```
10 class ParticleManager implements CSProcess {
11
12   def ChannelInput fromParticles
13   def ChannelOutput toParticles
14   def DisplayList toUI
15   def int CANVASSIZE
16   def int PARTICLES
17   def int CENTRE
18   def int START_TEMP
19   def ChannelInput fromUIButtons
20   def ChannelOutput toUILabel
21   def ChannelOutput toUIPause
22
```

Listing 11-4 ParticleManager Properties

The initialisation of the `ParticleManager` is shown in Listing 11-5. The variable `colourList` {24–26} contains a list of `java.awt.colors` that is used to colour the particles once they start moving. The variable `temperature` {28} is assigned the value of property `START_TEMP`.

The next part {30–46} initialises the variables that will be used by the `DisplayList` mechanism. The variable, `particleGraphics` {30}, used to `set()` a `DisplayList` comprises an array of `GraphicsCommands`. The initial element of `particleGraphics` {32} contains a `GraphicsCommand` that clears the display area. The remainder of `particleGraphics` comprises two elements per particle. The first element of which is a command to set the colour of the particle and the second will draw a circle of that colour with a radius of 10 pixels at the position of the particle. However for initialisation, each particle is set to the colour `BLACK` {36} and placed at the `CENTRE` {37} of the display area. This is captured in the variable `initialGraphic` {34}. The nested for loops {39–44} copy the `initialGraphic` into the array `particleGraphics`. Thus `particleGraphics` comprises a first command to clear the display followed by as many pairs of `GraphicsCommands` as there are particles needing to be drawn. The `DisplayList`, `toUI` is then `set()` to `particleGraphics` {46}. The manner in which the `DisplayList` is manipulated will be described later.

The two element array `positionGraphic` {47–51} will subsequently be used to update the `DisplayList` to reflect the movement of particles. It is initialised to sensible values that will be overwritten. However it can be observed that the first element of the array contains a command to set the colour and the second causes the drawing of a circle of that colour. The `ParticleManager` process alternates over inputs from the user interface buttons, `fromUIButtons` and from the particles on channel `fromParticles` {53}. The `String initTemp` is defined to hold the initial value of `temperature` {55} surrounded by spaces. This `String` is then written to the label that displays this value using the channel `toUILabel` {56}.

```
23    void run() {
24      def colourList = [ Color.BLUE, Color.GREEN,
25                         Color.RED, Color.MAGENTA,
26                         Color.CYAN, Color.YELLOW]
27
28      def temperature = START_TEMP
29
30      GraphicsCommand[] particleGraphics = new GraphicsCommand[1+(PARTICLES*2)]
31
32      particleGraphics[0] = new GraphicsCommand.ClearRect(0, 0,
        CANVASSIZE,CANVASSIZE)
33
34      GraphicsCommand [] initialGraphic = new GraphicsCommand [ 2 ]
35
36      initialGraphic[0] = new GraphicsCommand.SetColor (Color.BLACK)
37      initialGraphic[1] = new GraphicsCommand.FillOval (CENTRE, CENTRE, 10, 10)
38
```

```
39      for ( i in 0 ..< PARTICLES ) {
40        def p = (i * 2) + 1
41        for ( j in 0 ..< 2) {
42          particleGraphics [p+j] = initialGraphic[j]
43        }
44      }
45
46      toUI.set (particleGraphics)
47      GraphicsCommand [] positionGraphic = new GraphicsCommand [ 2 ]
48      positionGraphic =
49        [ new GraphicsCommand.SetColor (Color.WHITE),
50          new GraphicsCommand.FillOval (CENTRE, CENTRE, 10, 10)
51        ]
52
53      def pmAlt = new ALT ( [fromUIButtons, fromParticles] )
54
55      def initTemp = " " + temperature + " "
56      toUILabel.write ( initTemp )
57
58      def direction = fromUIButtons.read()
59      while ( direction != "START" ) {
60        direction = fromUIButtons.read()
61      }
62      toUIPause.write("PAUSE")
63
```

Listing 11-5 ParticleManager Initialisation

The variable `direction` is read from the channel `fromUIButtons` {58}. A user interface button signals a button event by communicating the `String` that is currently displayed by the button. Recall that all the user interface buttons are connected to the same channel, `fromUIButtons`. Only the `START`/`PAUSE`/`RESTART` button has the initial value `START` and thus the process will wait until the button labelled `START` is pressed. This behaviour is captured in the `while` loop {59–61}, which ignores any other button events. Once `START` has been read, the button's text value is changed to `PAUSE` {62} by writing to the `toUIPause` channel. The operation of the system now commences and this is shown in Listing 11-6.

The `index` of the selected alternative is obtained, with priority being given to button events {65}. If the value `read` from the channel `fromUIButtons` is `PAUSE` {68} then it is immediately overwritten with `RESTART` {69}. The process then waits for the button event `RESTART` ignoring all other button events {71–73}. Once the system has been restarted the button is overwritten with the value `PAUSE` {74}.

```
64      while (true) {
65        def index = pmAlt.priSelect()
66        if ( index == 0 ) { // dealing with a button event
67          direction = fromUIButtons.read()
68          if (direction == "PAUSE" ) {
69            toUIPause.write("RESTART")
70            direction = fromUIButtons.read()
71            while ( direction != "RESTART" ) {
72              direction = fromUIButtons.read()
73            }
74            toUIPause.write("PAUSE")
75          }
76          else {
77            if (( direction == "Up" ) && ( temperature < 50 )) {
78                 temperature = temperature + 5
79                 def s = "+" + temperature + "+"
80                 toUILabel.write ( s )
81            }
82            else {
83              if ( (direction == "Down" ) && ( temperature > 10 ) ) {
84                 temperature = temperature - 5
85                 def s = "-" + temperature + "-"
86                 toUILabel.write ( s )
```

```
87                }
88                else {
89                }
90              }
91            }
92          }
```

Listing 11-6 ParticleManager Button Event Processing

If the value `read` into `direction` is not `PAUSE` then it must either be `Up` or `Down` which are the text strings associated with the buttons that manipulate the `temperature` of the system. If the `Up` button is pressed and provided the current value of `temperature` is less than 50 {70} then the `temperature` is raised by 5 {78} and the new value of `temperature` is written to the interface using the channel `toUILabel` surrounded by + symbols {79–80}. Similarly if the `Down` button is pressed then the `temperature` is reduced by 5 provided its current value is greater than 10 and is output surrounded by – symbols {83–89}.

Listing 11-7 shows the processing that deals with the movement of particles.

```
93        else { // index is 1 particle movement
94          def p = (Position) fromParticles.read()
95          if ( p.lx > CANVASSIZE ) { p.lx = (2 * CANVASSIZE) - p.lx }
96          if ( p.ly > CANVASSIZE ) { p.ly = (2 * CANVASSIZE) - p.ly }
97          if ( p.lx < 0 ) { p.lx = 0 - p.lx }
98          if ( p.ly < 0 ) { p.ly = 0 - p.ly }
99          positionGraphic [0] = new GraphicsCommand.SetColor( colourList.
            getAt(p.id%6))
100         positionGraphic [1] = new GraphicsCommand.FillOval (p.lx, p.ly, 10, 10)
101         toUI.change ( positionGraphic, 1 + ( p.id * 2) )
102         p.px = p.lx
103         p.py = p.ly
104         p.temperature = temperature
105         toParticles.write(p)
106       } // index test
107     } // while
108   } // run
109 }
```

Listing 11-7 ParticleManager Particle Movement Processing

Recall that ParticleManager is behaving as a server process. Hence we would expect to see it read a client request {94}, undertake some processing and then respond with the return value {105}. The `Position` data object is `read` into the variable `p` from the channel `fromParticles` {94}. The proposed location `[lx, ly]` of the particle is then assessed as to whether it still remains within the display area {95–98} and if not, its position is adjusted assuming that the reflection from the side of the display area involves no friction or elastic compression of the particle. The value of the `PositionGraphic` array is then modified to reflect the particle's colour by taking the modulus 6 remainder of the particle's id {99} and then setting the centre of the circle to `[lx, ly]` {100}. This is then used to overwrite the data for this particle in the DisplayList parameter using the `toUI.change()` method {101}.

The position of the particle can now be updated {102, 103}. The current value of temperature is assigned to the corresponding property of object `p` {104} and the updated object `p` is then written back to the waiting `Particle` process {105}, as described in Section 11.2.2.

The description of the operation of a `DisplayList` can now be completed. An `ActiveCanvas` takes the `DisplayList` object as a parameter. Internally, the ActiveCanvas constructs two copies of the associated DisplayList array of Graphics commands. These copies are used to provide a double buffering mechanism; this however is hidden from the programmer. At a specified period the `ActiveCanvas` draws the current buffer on the display, while other changes are recorded in the other copy. This mechanism is repeated displaying the first buffer and recording changes in the second and then displaying the second buffer while recording changes in the first copy.

The `DisplayList` is initialised by a set method {46}. Thereafter specific elements of the `DisplayList` can be altered using the change method {101}. Thus the programmer generates the effect of continually updating the display, which in fact is using a double buffering technique to smooth the repainting of the display. The user is not concerned with the repainting of the display as this handled within the `ActiveCanvas` process. Thus the `DisplayList` array of `GrahicsCommand`s has an initial element that clears the display area, which is then overwritten by the sequence of `GraphicsCommands` in the array. In this manner sophisticated animation can be achieved, without having to overwrite each particle individually.

11.2.5 The UserInterface Process

The `UserInterface` process is shown in Listing 11-8. The properties of the process include the `particleCanvas`, `fromPM` {12}, the size of the canvas {13}, the two input channels, `tempValueConfig` {14} and `pauseButtonConfig` {15} used to configure the temperature value and the start button. Finally, the `buttonEvent` channel is used to output button events to the `ParticleManager` process {16}.

The `run` method of this process comprises a declarative style list of definitions and associated method calls that instantiates the graphical user interface. First, `root` {19}, an `ActiveClosingFrame` is defined that will be used to hold the rest of the interface components. An `ActiveClosiongFrame` is defined with the frame's title as a parameter and is not introduced by a property name because these processes are defined as Java classes and thus are constructed using the normal Java mechanism. `ActiveClosingFrame` is a specialisation of `ActiveFrame` that permits the closing of the frame using the normal window based controls. Interface components have to be added to the enclosed frame which is accessed by means of the `getActiveFrameMethod()` call {20}. The next part of the Listing shows the definition of the interface widgets both active and ordinary AWT non-active ones which can be mixed as required. The `Label`, `tempLabel`, which displays the text '`Temperature`' is constructed {21}. An `ActiveLabel` called `tempValue` is then defined {22} with the channel `tempValueConfig` as its parameter. Typically, an active widget has a constructor that comprises the configuration and event channels, together with any other appropriate parameter. The alignment of the label is also specified {23}. After this the required `ActiveButton`s are defined {24–26}, in which the `null` parameter is a placeholder for the not needed configuration channel. The additional parameter specifies the initial text associated with the button. The `pauseButton` requires a configuration channel {26} because the value of the text `String` associated with the button changes as the application progresses.

```
10 class UserInterface implements CSProcess {
11
12   def ActiveCanvas particleCanvas
13   def int canvasSize
14   def ChannelInput tempValueConfig
15   def ChannelInput pauseButtonConfig
16   def ChannelOutput buttonEvent
17
18   void run() {
19     def root = new ActiveClosingFrame ("Brownian Motion Particle System")
20     def mainFrame = root.getActiveFrame()
21     def tempLabel = new Label ("Temperature")
22     def tempValue = new ActiveLabel (tempValueConfig)
23     tempValue.setAlignment( Label.CENTER)
24     def upButton = new ActiveButton (null, buttonEvent, "Up")
25     def downButton = new ActiveButton (null, buttonEvent, "Down")
26     def pauseButton = new ActiveButton(pauseButtonConfig, buttonEvent, "START" )
27     def tempContainer = new Container()
28     tempContainer.setLayout ( new GridLayout ( 1, 5 ) )
29     tempContainer.add ( pauseButton )
30     tempContainer.add ( tempLabel )
31     tempContainer.add ( upButton )
32     tempContainer.add ( tempValue )
33     tempContainer.add ( downButton )
34     particleCanvas.setSize (canvasSize, canvasSize)
35     mainFrame.setLayout( new BorderLayout() )
36     mainFrame.add (particleCanvas, BorderLayout.CENTER)
37     mainFrame.add (tempContainer, BorderLayout.SOUTH)
38     mainFrame.pack()
39     mainFrame.setVisible ( true )
40     def network = [root, particleCanvas, tempValue, upButton,
       downButton, pauseButton]
41     new PAR (network).run()
42   }
43 }
```

Listing 11-8 The User Interface Process

Next a `Container`, `tempContainer` is defined {27} that holds all the components associated with the manipulation of temperature together with the `pauseButton`. The `Container` uses a `GridLayout` {28}. The previously defined buttons and labels are then added to `the tempContainer` {29–33}. The size of `particleCanvas` is specified {34}.

The `mainframe` can now be created {35–39} by specifying it to be a `BorderLayout` {35}. The `particleCanvas` and `tempContainer` are then `added` to the `mainframe` in the `CENTER` and `SOUTH` of the layout {36, 37}. The `mainframe` is then `packed` and `setVisible` {38, 39}, in the manner normally required by AWT interfaces.

Finally, a process `network` is constructed that comprises the root and the remaining active widgets {40}. The `network` is then run {41} and that is all that needs to be specified for the user interface requirements of this application. The event handler and listener methods normally required do not have to be written as these have been encapsulated within the active widgets, thereby simplifying the construction of the user interface.

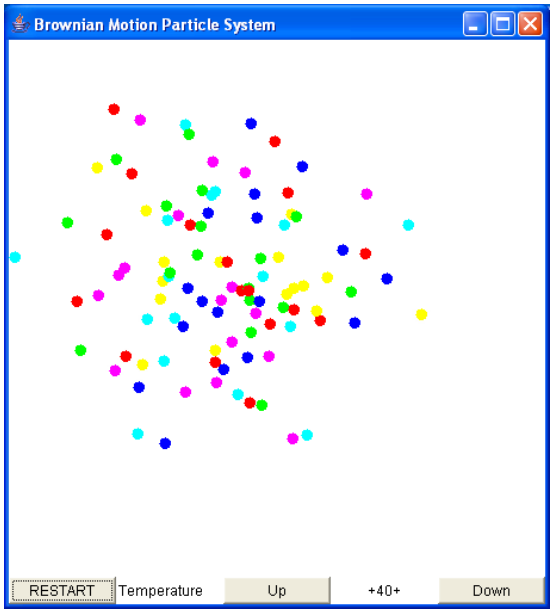
11.2.6 Invoking the Brownian Motion System

Listing 11-9 gives the script that is required to invoke the Brownian motion system. The `any2one` channel `connect` and the `one2any` channel `update` are defined {10, 11}. The fundamental constants of the system are either obtained from a user interaction or defined as constants {13-16}. The empty `List network` is defined {18} to which is appended each of the `Particle` processes {20–26}. The `ParticleInterface` process is finally appended to `network` {28–33}. The system is then executed by `run`ning `PAR` {35}.

```
10 def connect = Channel.any2one()
11 def update = Channel.one2any()
12
13 def CSIZE = Ask.Int ("Size of Canvas (200, 600)?: ", 200, 600)
14 def CENTRE = CSIZE / 2
15 def PARTICLES = Ask.Int ("Number of Particles (10, 200)?: ", 10, 200)
16 def INIT_TEMP = 20
17
18 def network = []
19 for ( i in 0..< PARTICLES ) {
20   network << new Particle ( id: i,
21                             sendPosition: connect.out(),
22                             getPosition: update.in(),
23                             x: CENTRE,
24                             y: CENTRE,
25                             temperature: INIT_TEMP )
26 }
27
28 network << ( new ParticleInterface ( inChannel: connect.in(),
29                                      outChannel: update.out(),
30                                      canvasSize: CSIZE,
31                                      particles: PARTICLES,
32                                      centre: CENTRE,
33                                      initialTemp: INIT_TEMP ) )
34 println "Starting Particle System"
35 new PAR ( network ).run()
```

Listing 11-9 The Script To Invoke the Brownian Motion System

A typical screen capture of the system, when it has been PAUSEd is shown in Output 11-1. We can observe that the control button has been set to RESTART. The temperature is currently set at 40 and the last operation was to increase its value because it is surrounded by + symbols. The Up and Down buttons are clearly visible. The screen is derived from a system that has a canvas size of 450 pixels running 100 particles.



Output 11-1 Screen Capture of the Brownian Motion System

11.3 Summary

This chapter has described how user interfaces can be constructed very simply using the active widget concept. Of most significance is the relative simplicity of the user interface definition as it does not require the programmer to implement the event and listener methods normally required. It has introduced a standard design pattern for user interface applications in which there is a process that undertakes the processing `ParticleManager` and its associated `UserInterface` process that are executed in parallel.

The concept of a `DisplayList` has been introduced which simplifies the programming of animated user interfaces based upon drawing in an `ActiveCanvas`. This in itself typifies the ease with which user interfaces can be constructed using active widgets because the programmer can use the parallel programming constructs to implement the interaction between user and application processes.

The design and implementation of user interfaces has become a much easier task because the user is no longer concerned with the writing of event handler and listener methods. Furthermore, the encapsulation of interface components, which run in their own thread and their associated event handler thread into a single process, makes it much easier to build the system that interacts with the interface.

11.4 Exercises

Exercise 111

The Control process in the Scaling system (Chapter 5) currently updates the scaling factor according to an automatic system. Replace this with a user interface that issues the suspend communication, obtains the current scaling factor and then asks the user for the new scaling factor that is then injected into the `Scaler`. The original and scaled values should also be output to the user interface. There is a widget called ActiveTextEnterField that may be useful (see the JCSP documentation).