**This tutorial will examine the process of proceeding from informal requirements, to representations of the behavioural aspects of the system, and finally to the implementation in a high level language. ©Roy Eagleson**

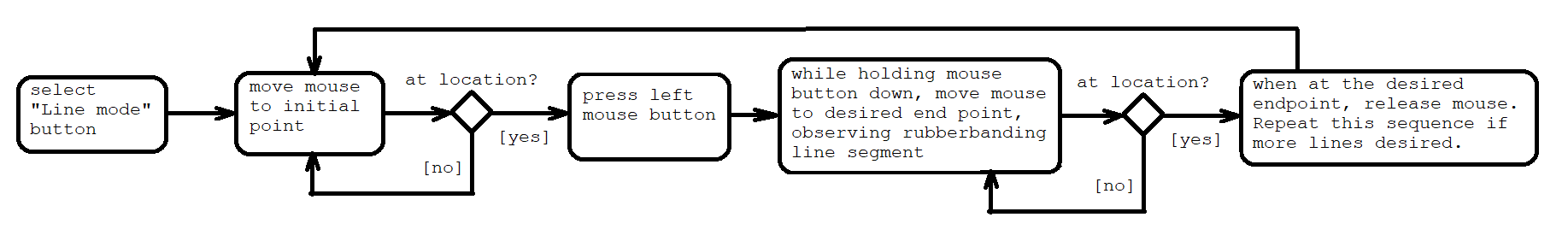
**Reprint with permission from the author: eagleson@uwo.ca**

The Sketchpad system will have a number of modes – these are the Use Cases – and so let’s begin with one of the simplest: *A mode to draw straight lines.* This will be our first Use Case. In other words, our first “case of usage” for our sketchpad system.

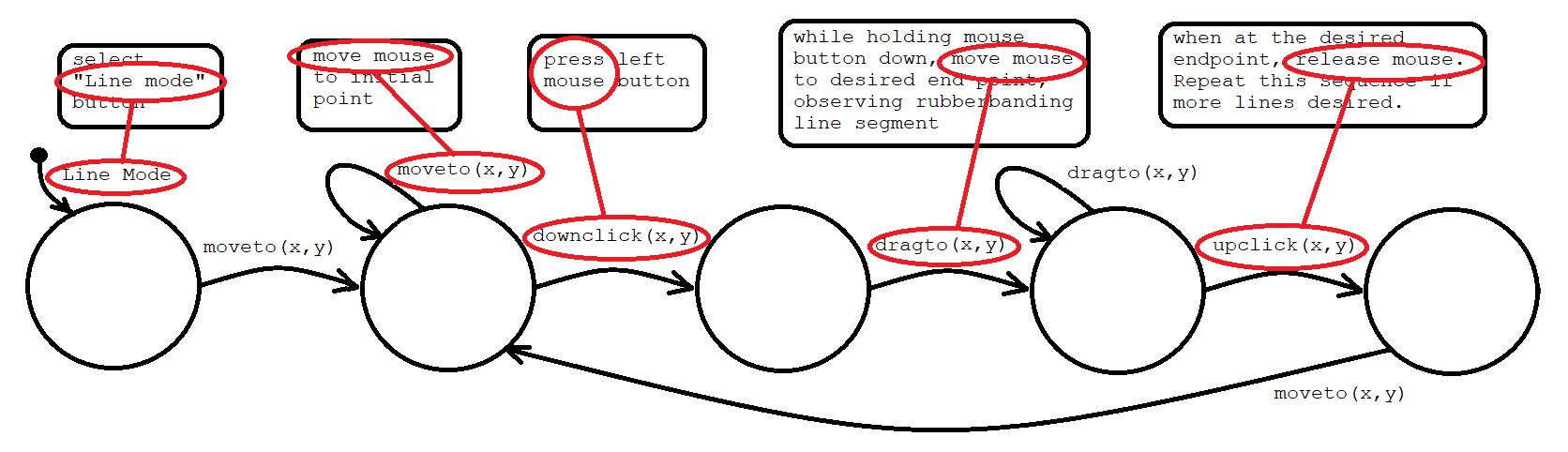
As covered in class, the user’s Activity in this mode can be written informally in English:

*“First, select the Line Mode button, and then move the mouse to the initial position. When you are there, press the left mouse button, and then while holding the mouse button down, drag the mouse to the desired endpoint (while observing the ‘rubberbanding’ line segment). When you are at the desired endpoint, release the mouse. Repeat this usage scenario if more lines are desired.”*

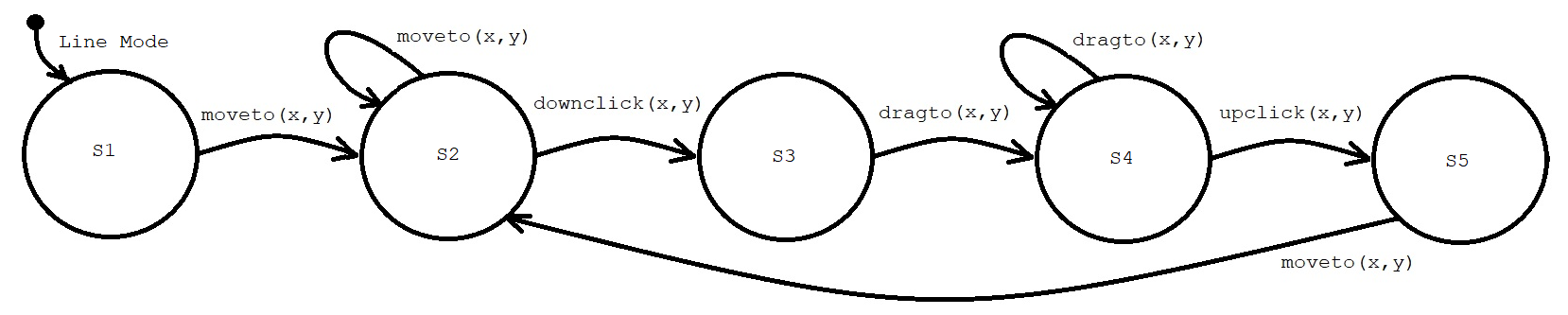
Or this can be broken into discrete states, where there is some positioning or selection output from the human, and represented using the following diagram:



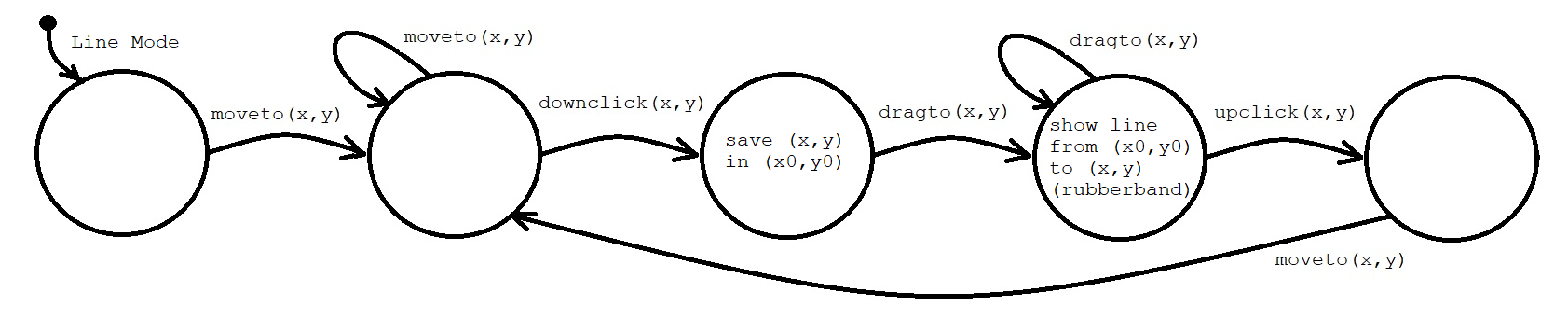
The original usage scenario was used to derive the user’s Activity Diagram, shown above.



The complement of this diagram is formed by realizing that each output from the human can be considered as an input event to the computer. This yields the following transition arcs for the system state diagram behavioural representation:



We can next consider representing the computer outputs (these are the inputs to the human). If we use a Moore machine, the individual states should represent the outputs from the computer. The output is provided by a ‘drawline’ function call, and also requires a pair of state variables (x0,y0) to save the initial point.

The semantics of this state diagram can be represented as a state table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Moore state Output: | Moveto(x,y) | Dragto(x,y) | Downclick(x,y) | Upclick(x,y) |
| S1: | Next state: S2 |  |  |  |
| S2: | Next state: S2 |  | Next state: S3 |  |
| S3: (x0,y0)🡨(x,y) |  | Next state: S4 |  |  |
| S4:drawline(x0,y0,x,y) |  | Next state: S4 |  | Next state: S5 |
| S5: | Next state: S2 |  |  |  |

This Moore State Diagram and state table can be converted to a Mealy State diagram by ‘backing up’ the outputs shown in the state bubble, to the incoming transition arcs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mealy  state: | Moveto(x,y)  { output } | Dragto(x,y) | Downclick(x,y) | Upclick(x,y) |
| S1: | Next state: S2 |  |  |  |
| S2: | Next state: S2 |  | { (x0,y0)🡨(x,y) }  Next state: S3 |  |
| S3: |  | {drawline(x0,y0,x,y)}  Next state: S4 |  |  |
| S4: |  | {drawline(x0,y0,x,y)}  Next state: S4 |  | Next state: S5 |
| S5: | Next state: S2 |  |  |  |

We can observe empty cells in this state table. And so we can analyse this representation to inquire how they should be filled. For example, if the system is in S1, there can be a Downclick(x,y) event (without any motion), and in that case, the system can proceed to state S3, and this transition should also “back up” the output that was associated with S3. Likewise, if the system is in S3, there can be an Upclick(x,y) event. And this enquiry leads to the suggestion that, in that case, the system should proceed to Next state:S5. We also can observe that in S1 or S2, there cannot be a “Drag” event (since there has not yet been a downclick), and therefore there cannot be an Upclick event. And likewise, in S3 or S4, there cannot be a downclick event (since the mouse button in these states is already down), and therefore, there cannot be a Moveto(x,y) event. And there cannot be an Upclick, nor a Drag. Accordingly, we can mark those five cells with “anything” since they are cells which alternately could be labelled as “don’t care” cells. And in S5, if there is a Downclick (without a move) then a new line is starting, so the system can go to S3 to handle it, “backing up” the output for S3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mealy state: | Moveto(x,y) | Dragto(x,y) | Downclick(x,y) | Upclick(x,y) |
| S1: | Next state: S2 | anything | { (x0,y0)🡨(x,y) }  Next state: S3 | anything |
| S2: | Next state: S2 | anything | { (x0,y0)🡨(x,y) }  Next state: S3 | anything |
| S3: | anything | {drawline(x0,y0,x,y)}  Next state: S4 | anything | Next state: S5 |
| S4: | anything | {drawline(x0,y0,x,y)}  Next state: S4 | anything | Next state: S5 |
| S5: | Next state: S2 | anything | { (x0,y0)🡨(x,y) }  Next state: S3 | anything |

We immediately notice that, by inspection, S1, S2 and S5, are equivalent states. And S3 and S4 are equivalent states. So our analysis allows us to synthesize the following simplified State Diagram:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mealy  state: | Moveto(x,y)  { output } | Dragto(x,y) | Downclick(x,y) | Upclick(x,y) |
| S2: | Next state: S2 | anything | { (x0,y0)🡨(x,y) }  Next state: S3 | anything |
| S3: | anything | { drawline(x0,y0,x,y) }  Next state: S3 | anything | Next state: S2 |

And since “anything” can be replaced with anything at all, including anything convenient, we can re-write as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mealy  state: | Moveto(x,y)  { output } | Dragto(x,y) | Downclick(x,y) | Upclick(x,y) |
| S2: | Next state: S2 | { drawline(x0,y0,x,y) }  Next state: S3 | { (x0,y0)🡨(x,y) }  Next state: S3 | Next state: S2 |
| S3: | Next state: S2 | { drawline(x0,y0,x,y) }  Next state: S3 | { (x0,y0)🡨(x,y) }  Next state: S3 | Next state: S2 |

And, so trivially, the behaviour of this system can be replaced by a single-state Mealy machine:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mealy  state: | Moveto(x,y)  { output } | Dragto(x,y) | Downclick(x,y) | Upclick(x,y) |
| S: | { }  Next state: S | { drawline(x0,y0,x,y) }  Next state: S | { (x0,y0)🡨(x,y) }  Next state: S | { }  Next state: S |

We can consider a single state, which needs to handle the following possible input events, and write in the language of *pseudocode*:

If Moveto(x,y) { } // “do nothing”

If Dragto(x,y) { “drawline from x0,y0, to x,y” }

If Downclick(x,y) { save x event in x0, save y event in y0 }

If Upclick(x,y) { } // “do nothing”

We can now take this pseudocode, and consider mapping it to the namespace of an existing API, for example, the Java AWT API. In that API, we could use inheritance and override the Frame methods named “mouseDown” and “mouseDrag”. Or we might consider the events delegating to awt.event listeners with methods named “mouseMoved”, “MouseDragged”, “MousePressed”, and “MouseReleased”.

We can also make use of the repaint() method to call paint(), which will clear the screen and ensure that the display is synchronized to the raster screen sweep, which is part of the AWT Frame component. Accordingly, consider the following transformation from pseudocode to Java AWT -- The cells with imperatives can be mapped directly onto the Frame methods:

import java.awt.\*;

public class sketch10 extends Frame {

int x0,y0,x,y;

public sketch10() { setSize(250,400); }

public boolean mouseDown(Event e,int ex,int ey) { x0=ex; y0=ey; return(true); }

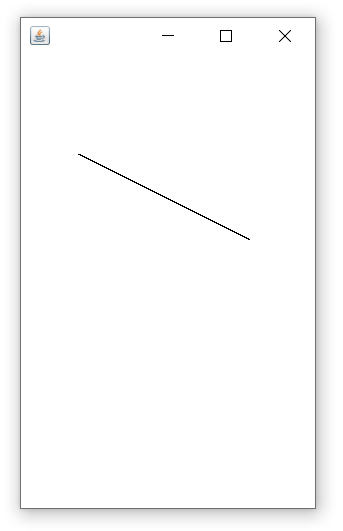
public boolean mouseDrag(Event e,int ex,int ey) { x=ex; y=ey; repaint(); return(true); }

public void paint(Graphics g) { g.drawLine(x0,y0,x,y); }

public static void main(String[] s){ new sketch10().setVisible(true); }

}

// "/cygdrive/c/Program Files/Java/jdk-14.0.2/bin/java" sketch10.java



This code implements an application which has the following view when the user has drawn a straight line:

Next, the cells with imperatives can be mapped directly onto awt.event listeners, by creating classes called “event handlers”

public void mouseMoved(MouseEvent e) { }

public void mouseDragged(MouseEvent e) { drawline(x0,y0,x=e.getX(),y=e.getY()); }

public void mousePressed(MouseEvent e) { x0=e.getX(); y0=e.getY(); }

public void mouseReleased(MouseEvent e) { }

Accordingly, we can have a different implementation which has exactly the same behaviour:

import java.awt.\*; import java.awt.event.\*;

public class sketch11 extends Frame {

public sketch11() {

setSize(250,400);

addMouseListener( new myMouseHandler());

addMouseMotionListener( new myMouseMotionHandler());

}

int x0,y0,x,y;

public class myMouseHandler extends MouseAdapter {

public void mousePressed(MouseEvent e){ x0=e.getX(); y0=e.getY(); }

public void mouseReleased(MouseEvent e) { }

}

public class myMouseMotionHandler extends MouseMotionAdapter {

public void mouseMoved(MouseEvent e) { }

public void mouseDragged(MouseEvent e){ x=e.getX(); y=e.getY(); repaint(); }

}

public void paint(Graphics g) { g.drawLine(x0,y0,x,y); }

public static void main(String[] s){ new sketch11().setVisible(true); }

}

// "/cygdrive/c/Program Files/Java/jdk-14.0.2/bin/javac" sketch11.java

Since there are many implementations which are behaviourally equivalent and functionally correct, we will next want to examine how they are structurally different, and what structural constraints might lead us to choose one over the other.

An important note: The source code is a functional implementation of the following informal requirements:

Usage Scenario:

*“Move the mouse to the initial position. When you are there, press the left mouse button, and then while holding the mouse button down, drag the mouse to the desired endpoint (while observing the ‘rubberbanding’ line segment). When you are at the desired endpoint, release the mouse. Repeat this usage scenario if more lines are desired.”* An important note: I challenge any of you to transform from one propositional representation to the other, without considering some of the intermediate steps on the transitions between concept and implementation!

**Is it really practical to consider transforming**

**the above *informal* behavioural specification**

**into the following *formal* implementation,**

**without considering some of the**

**intermediate transformations**

**covered in this tutorial?**

// implemented system:

import java.awt.\*; import java.awt.event.\*;

public class sketch11 extends Frame {

public sketch11() {

setSize(250,400);

addMouseListener( new myMouseHandler());

addMouseMotionListener( new myMouseMotionHandler());

}

int x0,y0,x,y;

public class myMouseHandler extends MouseAdapter {

public void mousePressed(MouseEvent e){ x0=e.getX(); y0=e.getY(); }

public void mouseReleased(MouseEvent e) { }

}

public class myMouseMotionHandler extends MouseMotionAdapter {

public void mouseMoved(MouseEvent e) { }

public void mouseDragged(MouseEvent e){ x=e.getX(); y=e.getY(); repaint(); }

}

public void paint(Graphics g) { g.drawLine(x0,y0,x,y); }

public static void main(String[] s){ new sketch11().setVisible(true); }

}

And consider this: Can you take this final block of source code, and transform into a description of the behaviour as a Usage Scenario? (in other words, to **analyse the implementation, and try to derive its behaviour in a way that can be described in terms of the user’s experience?** )