2020 CI401
Introduction to programming

Week 2.02
ATM lab notes

15th February 2021

Roger Evans

Module leader

Introduction to ATM

Starter project - ATM

- The ATM project simulates a cashpoint (except that it doesn't give you any money )
- In this week's lab exercise, you are given a version of ATM that does everything except the actual bank functions (deposit, withdraw etc).
- The lab exercise is to turn it into a ATM (with help from tutors if you need it)
- The solution to this lab is the starting point for your independent project work, if you choose to do ATM.

Demonstrating ATM

- The solution system – an ATM system which lets you log into your account, check balance and deposit and withdraw money

- The lab exercise system – has no banking functionality! You can still type in your account number and password, and click buttons, but it won't ever find your account or carry out the banking commands

Try this!

- To enter an account number click on (don't type!) 1 0 0 0 1 Ent
- Then to enter a password click on 1 1 1 1 1 Ent
- In the lab system, you will get the message 'Unknown account/password'
- In the solution system you will be logged in and can try
  - Bal to display your balance
  - 1 0 Dep to deposit 10 pounds
  - 5 0 W/D
  - Fin to log out

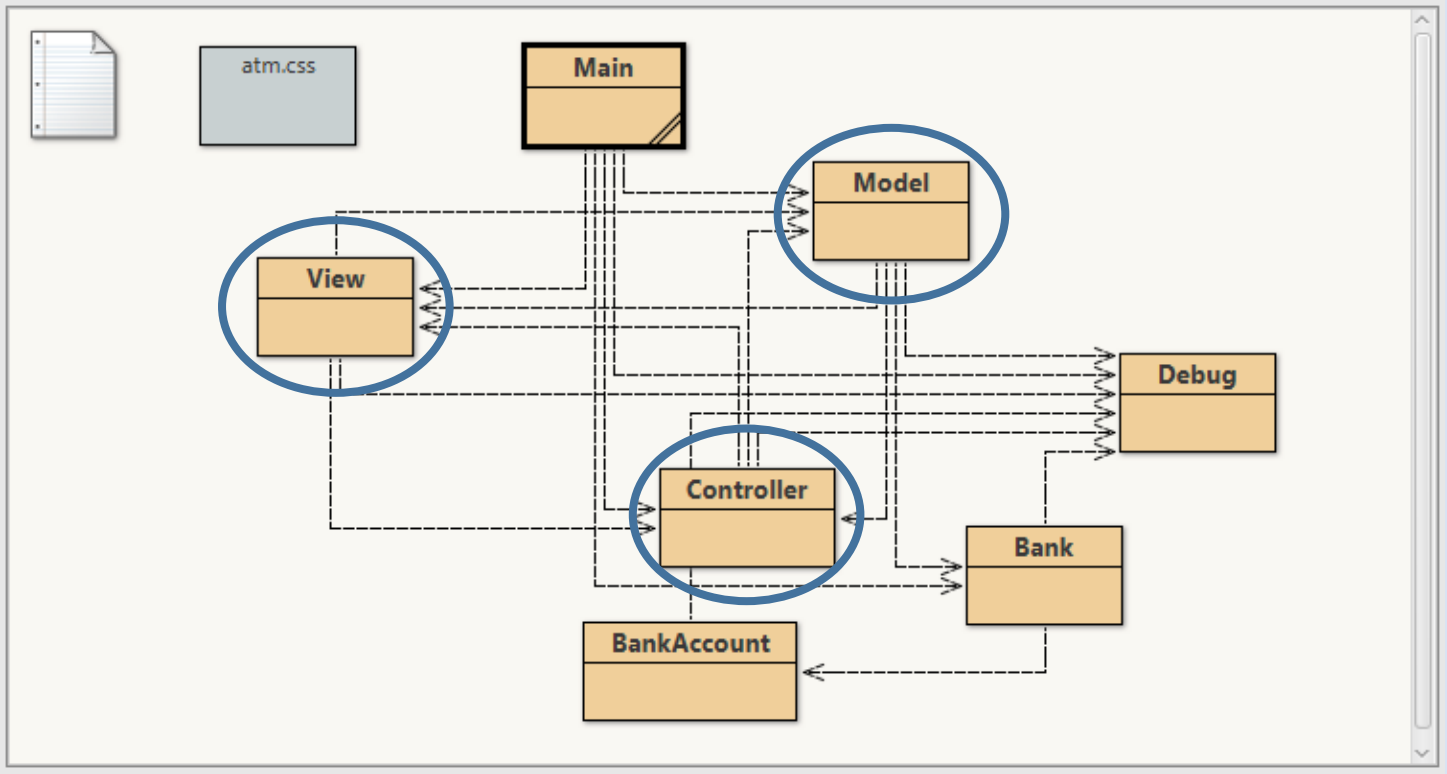
ATM organisation – MVC again!

- The idea of Model-View-Controller is to separate out three different parts of an application:
  - The Model class contains the core of the application – all the objects and methods that are needed for the application to 'do something'
  - The View class creates the actual GUI on the screen, with buttons, text boxes, colours etc. It doesn't know anything about what the application 'does', but it updates the display whenever the Model changes
  - The Controller links the View and the Model – when something happens in the View (eg a button gets clicked), the controller decides what the model should do.

MVC in the ATM

Here's the class diagram

- The Model contains the business logic – logging in, communicating with the bank etc.
- The View has all the buttons and layout features we see in the GUI. When the model changes, it updates its display
- The Controller accepts the button presses from the view, and turns them into instructions for the model



Benefits of MVC

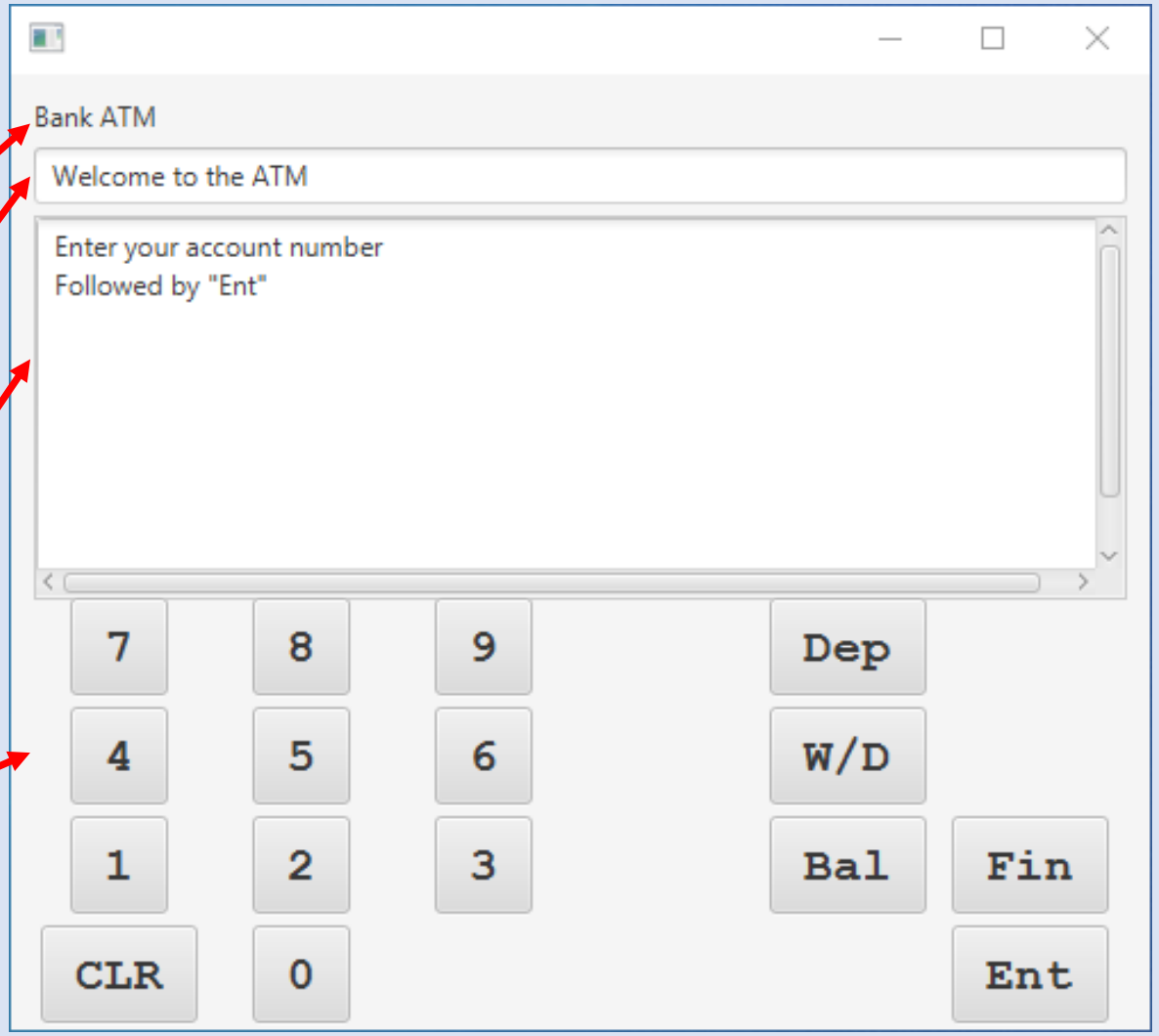
- Each component only has to worry about its own part of the task
- It’s easy to change the View (or have multiple views) without breaking the underlying application Model
- It’s easy to change the Model, make sure it is correct, and then update the View and the Controller to work with the new Model (eg adding new functions)
- The Controller can check that the user has typed sensible commands before passing them on to the Model (‘validate’ the user’s inputs)

The View

The View object is a fairly straightforward JavaFX UI.

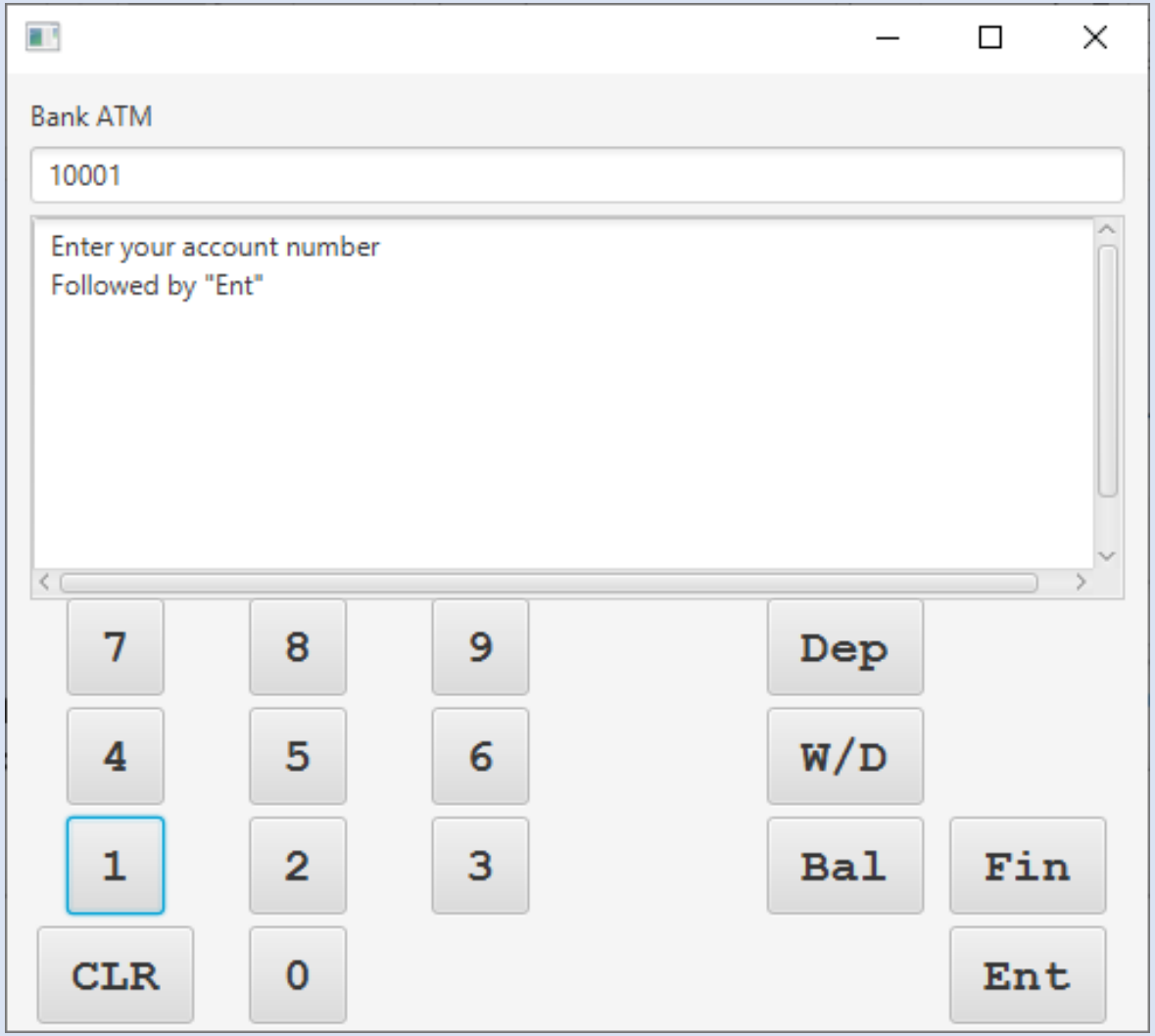
It is a GridPane containing:

- A Label for the title
- A Textbox for messages/typing
- A TextArea for the reply/info (actually wrapped in ScrollPane so it can be bigger than shows on screen)
- A TilePane full of Buttons



The View

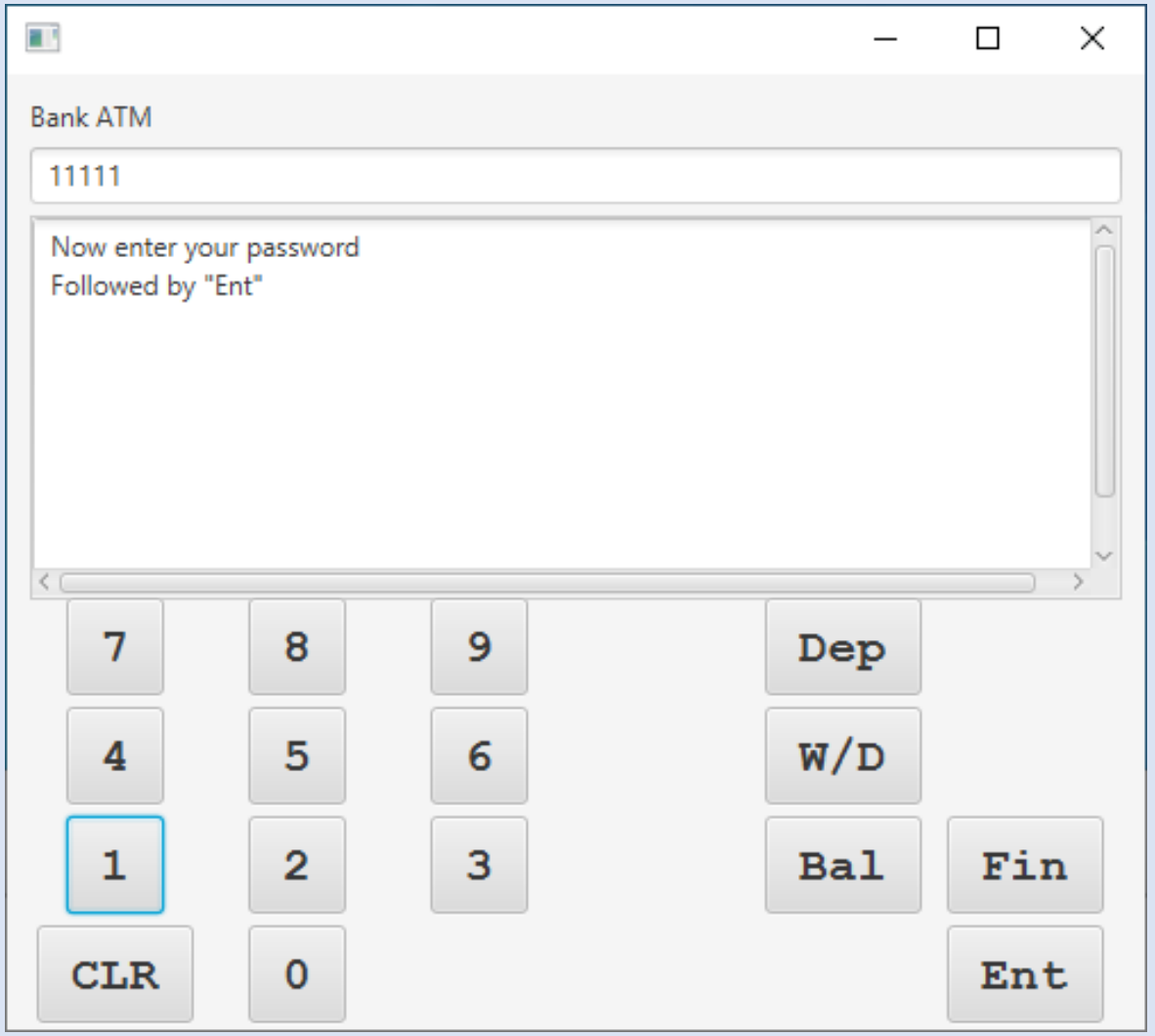
- When you click on numbers, they appear in the message area
- When you click on Ent (Enter) the reply/info area changes to ask for your password



The View

- When you click on more numbers, they appear in the message area
- When you click on Ent (Enter) the reply/info area, it tries to log you into your account
- (you will need to add some code to get past this point – see below)

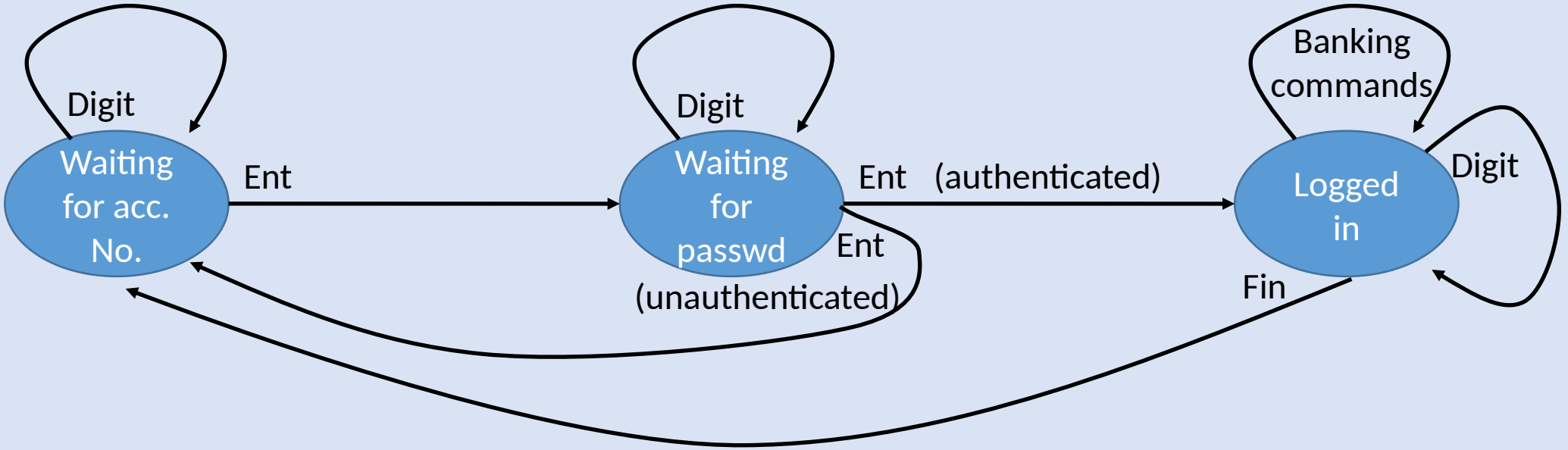
- Each button click generates a message to the Model (via the Controller), which executes the requested action and then tells the View to update the screen



The Model class

- The Model:
  - receives messages from the Controller when buttons are clicked on
  - manages user login by always being in one of three states:
    - waiting for account number
    - waiting for password
    - logged in
  - communicates with the Bank object to
    - authenticate account number and password and log in
    - execute banking instructions (when logged in)
  - updates the View with just three Strings:
    - the title message
    - the message area
    - the reply/info area

The Model as a 'finite state machine' (FSM)



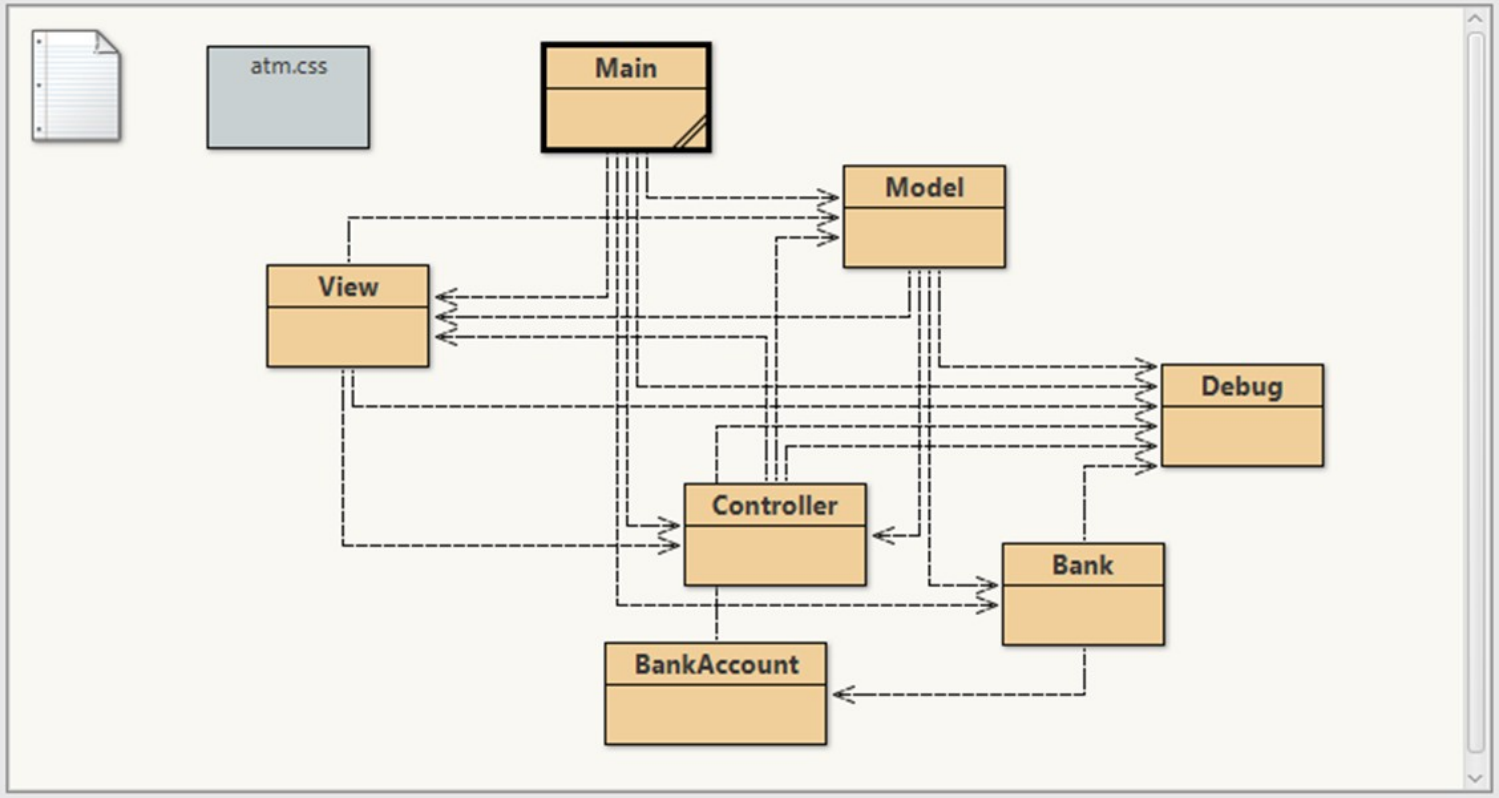
The Controller class

- As with Breakout, the controller is simplest of the three classes
- This provides the event handler function that the View uses
- Whenever the user presses a key, the controller decides what the game should do and tells the Model to do it
- It does this simply on the basis of the labels on each of the buttons. For example
  - When the user clicks a digit, it tells the Model that ‘a digit’ has been clicked (and of course, which one was clicked)
  - Each of the other functions generates a unique message to the Model (in other words, calls a specific method in the Model class)
- So it basically provides a mapping from View buttons to Model functions (Which you can change)

Other classes

- The Main class – starts the program, creates the Model, View, Controller and Bank objects and ‘joins them together’
- The Bank and BankAccount classes – the main data classes representing the banking information that the program manages
- The Debug class – prints out messages about what is going on to help you debug the program

The ATM classes



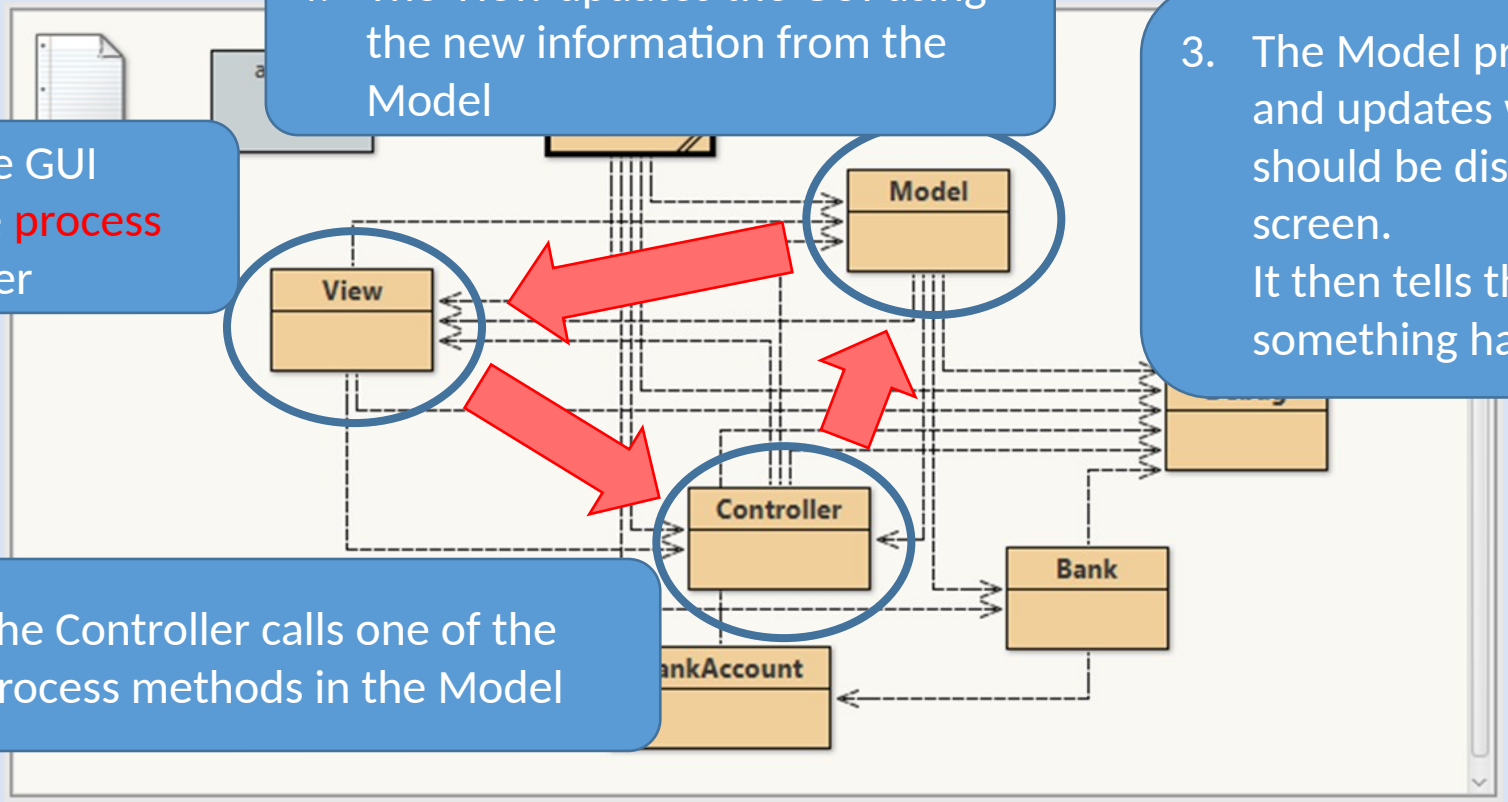
How the components communicate

1. Click on a button on the GUI
   The View class calls the process method in the Controller

2. The Controller calls one of the process methods in the Model

3. The Model processes the action and updates what it thinks should be displayed on the screen.
   It then tells the View that something has changed

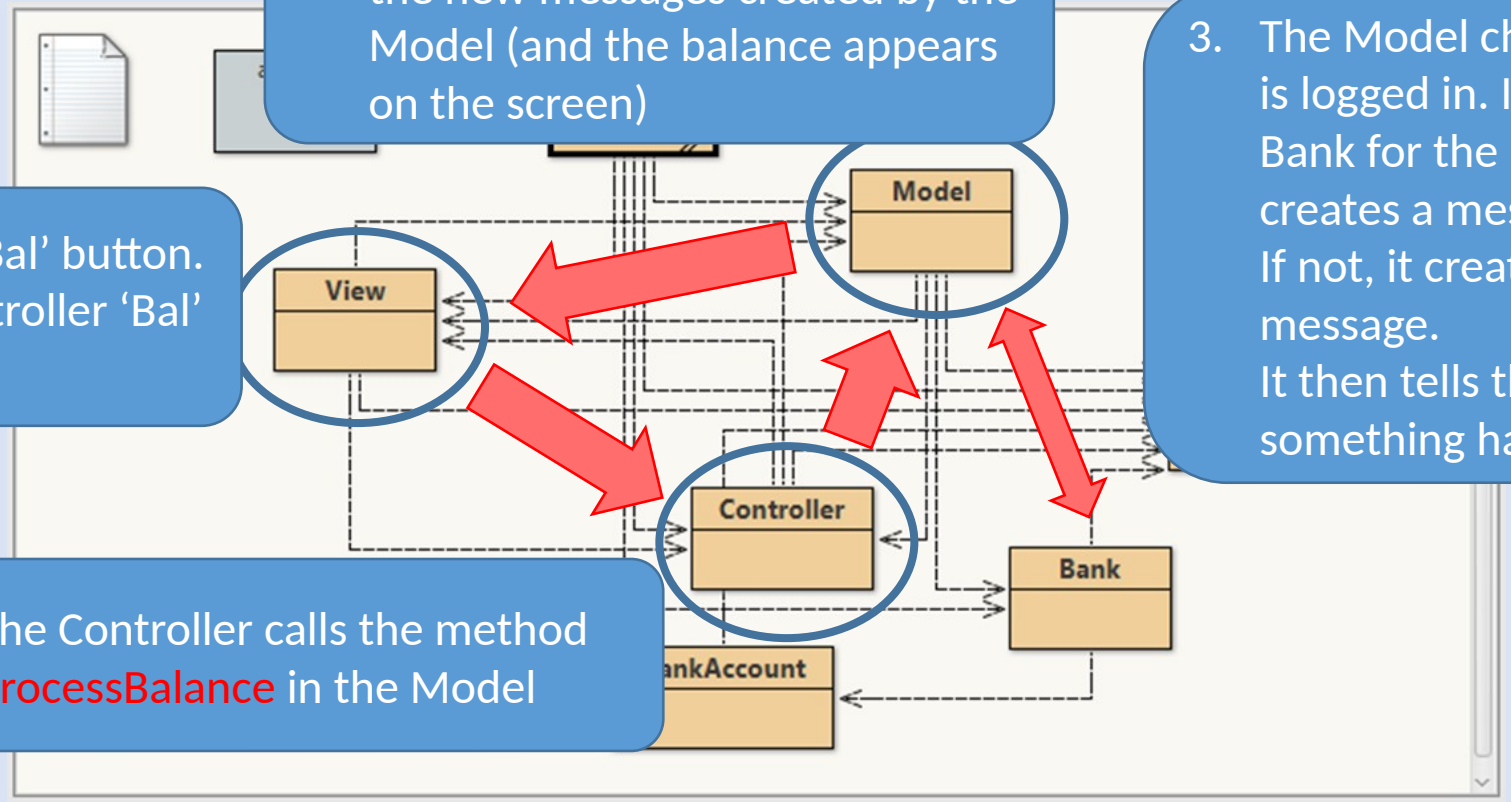
4. The View updates the GUI using the new information from the Model



ATM example – getting the balance

4. The View updates the GUI with the new messages created by the Model (and the balance appears on the screen)

1. The user presses the 'Bal' button. The View tells the Controller 'Bal' has been pressed.



3. The Model checks that the user is logged in. If so it asks the Bank for the balance and creates a message to display it. If not, it creates an error message.
It then tells the View that something has changed

2. The Controller calls the method processBalance in the Model

The ATM code

The Main class

Main class

- Create a Bank object, and add two test accounts to it
- Create Model, View and Controller objects – the Model needs the bank object as an argument to its constructor
- Join them together
- Start the view (user interface)
- Initialise the model and display it

```
21      public void start(Stage window)
22      {
23          // set up debugging and print initial debugging message
24          Debug.set(true);
25          Debug.trace("atm starting");
26          Debug.trace("Main::start");
27
28          // Create a Bank object for this ATM
29          Bank b = new Bank();
30          // add some test bank accounts
31          b.addBankAccount(10001, 11111, 100);
32          b.addBankAccount(10002, 22222, 50);
33
34          // Create the Model, View and Controller objects
35          Model model = new Model(b);   // the model needs the Bank object to 'talk to' the bank
36          View  view  = new View();
37          Controller controller  = new Controller();
38
39          // Link them together so they can talk to each other
40          // Each one has instances variable for the other two
41          model.view = view;
42          model.controller = controller;
43
44          controller.model = model;
45          controller.view = view;
46
47          view.model = model;
48          view.controller = controller;
49
50          // start up the GUI (view), and then tell the model to initialise and display itself
51          view.start(window);
52          model.initialise("Welcome to the ATM");
53          model.display();
54
55          // application is now running
56          Debug.trace("atm running");
57      }
58  }
```

The View class

View class start method

- Sets up the main JavaFX interface
- The GUI uses a GridPane layout manager with four children:
  - A Label for the title
  - A TextField for the message (number etc)
  - A ScrollPane containing a TextArea (a multi-line text box), for the reply
  - Lastly (next slide) a TilePlane for the buttons

```
44  public void start(Stage window)
45  {
46      Debug.trace("View::start");
47
48      // create the user interface component objects
49      // The ATM is a vertical grid of four components -
50      // label, two text boxes, and a tiled panel
51      // of buttons
52
53      // layout objects
54      grid = new GridPane();
55      grid.setId("Layout");           // assign an id to be used in css file
56      buttonPane = new TilePane();
57      buttonPane.setId("Buttons");    // assign an id to be used in css file
58
59      // controls
60      title  = new Label();           // Message bar at the top for the title
61      grid.add( title, 0, 0);         // Add to GUI at the top
62
63      message  = new TextField();     // text field for numbers and error messages
64      message.setEditable(false);     // Read only (user can't type in)
65      grid.add( message, 0, 1);       // Add to GUI on second row
66
67      reply  = new TextArea();        // multi-line text area for instructions
68      reply.setEditable(false);       // Read only (user can't type in)
69      scrollPane  = new ScrollPane(); // create a scrolling window
70      scrollPane.setContent( reply ); // put the text area 'inside' the scrolling wind
71      grid.add( scrollPane, 0, 2);    // add the scrolling window to GUI on third row
72
```

View class start method - buttons

- The Buttons are laid out on a TilePane – a layout manager that lays out its children evenly
- The buttons are all the same except for their labels
- We have a rectangular array of labels, and loop through it, making buttons and adding them to the TilePane
- We add the same method (buttonClicked) as the event handler for all of them
- For an empty string we use a dummy Text element to leave a space

```
72
73      // Buttons - these are laid out on a tiled pane, then
74      // the whole pane is added to the main grid as the fourth row
75
76      // Button labels - empty strings are for blank spaces
77      // The number of button per row should match what is set in
78      // the css file
79      String labels[][] = {
80              {"7",   "8",  "9",  "",  "Dep",  ""},
81              {"4",   "5",  "6",  "",  "W/D",  ""},
82              {"1",   "2",  "3",  "",  "Bal",  "Fin"},
83              {"CLR", "0",  "",   "",  "",     "Ent"} };
84
85      // loop through the array, making a Button object for each label
86      // (and an empty text label for each blank space) and adding them to the buttonP
87      // The number of button per row is set in the css file, not the array.
88      for ( String[] row: labels ) {
89          for (String label: row) {
90              if ( label.length() >= 1 ) {
91                  // non-empty string - make a button
92                  Button b = new Button( label );
93                  b.setOnAction( this::buttonClicked ); // set the method to call when
94                  buttonPane.getChildren().add( b );    // and add to tiled pane
95              } else {
96                  // empty string - add an empty text element as a spacer
97                  buttonPane.getChildren().add( new Text() );
98
99              }
100         }
101     }
102     grid.add(buttonPane,0,3); // add the tiled pane of buttons to the grid
103
104     // add the complete GUI to the window and display it
105     Scene scene = new Scene(grid, W, H);
106     scene.getStylesheets().add("atm.css"); // tell the app to use our css file
107     window.setScene(scene);
108     window.show();
```

View class communication methods

- The buttonClicked method gets called whenever any button is clicked
- It finds out which button from the event object it is given, and passes the button's label string to the Controller
- update is called by the Model whenever the Model changes
- Updating is easy – there are just three strings to get from the model and update our controls – the title, the message and the reply

```
111     // This is how the View talks to the Controller
112     // This method is called when a button is pressed
113     // It fetches the label on the button and passes it to the controller's process meth
114     public void buttonClicked(ActionEvent event) {
115         // this line asks the event to provide the actual Button object that was clicked
116         Button b = ((Button) event.getSource());
117         if ( controller != null )
118         {
119             String label = b.getText();   // get the button label
120             Debug.trace( "View::buttonClicked: label = "+ label );
121             // Try setting a breakpoint here
122             controller.process( label );  // Pass it to the controller's process method
123         }
124     }
125
126     // This is how the Model talks to the View
127     // This method gets called BY THE MODEL, whenever the model changes
128     // It fetches th title, display1 and display2 variables from the model
129     // and displays them in the GUI
130     public void update()
131     {
132         if (model != null) {
133             Debug.trace( "View::update" );
134             String message1 = model.title;          // get the new title from the model
135             title.setText(message1);                // set the message text to be the titl
136             String message2 = model.display1;       // get the new message1 from the model
137             message.setText( message2 );            // add it as text of GUI control outpu
138             String message3 = model.display2;       // get the new message2 from the model
139             reply.setText( message3 );              // add it as text of GUI control outpu
140         }
141     }
142 }
143
```

View class atm.css file

- The controls added in the View class were given css identifiers, using the setId method
- In atm.css we can add CSS rules to control their appearance
- Here you see rules for:
  - #Layout – the GridPane
  - #Buttons – the TilePane
  - .button – all the Buttons
- You could add colours etc.
- NB: we set the number of columns of buttons to 6, and we put 6 labels in each row of the array, but nothing ensures these are the same – the layout actually ignores the array rows and uses the number here to lay out its columns!

```
#Layout {
    -fx-grid-lines-visible: false;
    -fx-hgap: 5;
    -fx-vgap: 5;
    -fx-padding: 10;
}

#Buttons {
    -fx-pref-columns: 6;
    -fx-pref-tile-width: 75;
    -fx-hgap: 5;
    -fx-vgap: 5;
}

.button {
    -fx-font: bold 16pt "courier new";
}
```

The Controller class

Controller class

- The Controller has one method which is called by the View when a button is clicked
- It gets given the label on the button, and then uses a switch statement to tell the model what to do – process a number, clear, enter, withdraw deposit, balance or finish.

```
// This is how the View talks to the Controller
// AND how the Controller talks to the Model
// This method is called by the View to respond to some user interface event
// The controller's job is to decide what to do. In this case it uses a switch
// statement to select the right method in the Model
public void process( String action )
{
    Debug.trace("Controller::process: action = " + action);
    switch (action) {
    case "1" : case "2" : case "3" : case "4" : case "5" :
    case "6" : case "7" : case "8" : case "9" : case "0" :
        model.processNumber(action);
        break;
    case "CLR":
        model.processClear();
        break;
    case "Ent":
        model.processEnter();
        break;
    case "W/D":
        model.processWithdraw();
        break;
    case "Dep":
        model.processDeposit();
        break;
    case "Bal":
        model.processBalance();
        break;
    case "Fin":
        model.processFinish();
        break;
    default:
        model.processUnknownKey(action);
        break;
    }
}
```

The Model class

Model class

- The first three Strings are used as values for the variable state, which tells the model which state it is in
- We have a set of variables for the inner workings of the ATM model (the current number in the message box, the account number and password)
- A second set of variables are used just to update the View
- Notice the constructor at the bottom – it needs to be given a Bank object to talk to (by the Main class)

```
3  // For the ATM, it keeps track of the information shown in the display
4  // (the title and two message boxes), and the interaction with the bank, executes
5  // commands provided by the controller and tells the view to update when
6  // something changes
7  public class Model
8  {
9      // the ATM model is always in one of three states - waiting for an account number,
10     // waiting for a password, or logged in and processing account requests.
11     // We use string values to represent each state:
12     // (the word 'final' tells Java we won't ever change the values of these variables)
13     final String ACCOUNT_NO = "account_no";
14     final String PASSWORD = "password";
15     final String LOGGED_IN = "logged_in";
16
17     // variables representing the ATM model
18     String state = ACCOUNT_NO;      // the state it is currently in
19     int  number = 0;                // current number displayed in GUI (as a number, not
20     Bank  bank = null;              // The ATM talks to a bank, represented by the Bank
21     int accNumber = -1;             // Account number typed in
22     int accPasswd = -1;             // Password typed in
23     // These three are what are shown on the View display
24     String title = "Bank ATM";      // The contents of the title message
25     String display1 = null;         // The contents of the Message 1 box (a single line)
26     String display2 = null;         // The contents of the Message 2 box (may be multipl
27
28     // The other parts of the model-view-controller setup
29     public View view;
30     public Controller controller;
31
32     // Model constructor - we pass it a Bank object representing the bank we want to tal
33     public Model(Bank b)
34     {
35         Debug.trace("Model::<constructor>");
36         bank = b;
37     }
38
```

Model class utility methods

- initialise just sets up the key variables of the ATM and sets the given message to be displayed (it doesn't update the display itself though)
- setState is just a utility to change from one state to another, printing a debugging message as it does so

```
38
39    // Initialising the ATM (or resetting after an error or logout)
40    // set state to ACCOUNT_NO, number to zero, and display message
41    // provided as argument and standard instruction message
42    public void initialise(String message) {
43        setState(ACCOUNT_NO);
44        number = 0;
45        display1 = message;
46        display2 =  "Enter your account number\n" +
47        "Followed by \"Ent\"";
48    }
49
50    // use this method to change state - mainly so we print a debugging message whenever
51    //the state changes
52    public void setState(String newState)
53    {
54        if ( !state.equals(newState) )
55        {
56            String oldState = state;
57            state = newState;
58            Debug.trace("Model::setState: changed state from "+ oldState + " to " + newS
59        }
60    }
61
```

Model class number methods

- processNumber is called each time a digit is clicked. Its argument is the digit (as a string)
- It does some 'magic' to turn the string into the actual number (eg convert "5" to 5), and then it updates the whole number represented by all the digits so far – multiply the old value by 10 and add this one (so clicking "5" when the number is 32 gives 325)
- processClear just resets the number to zero.
- Both then call display to tell the View to update, showing the new value on the screen

```
64
65    // process a number key (the key is specified by the label argument)
66    public void processNumber(String label)
67    {
68        // a little magic to turn the first char of the label into an int
69        // and update the number variable with it
70        char c = label.charAt(0);
71        number = number * 10 + c-'0';           // Build number
72        // show the new number in the display
73        display1 = "" + number;
74        display();  // update the GUI
75    }
76
77    // process the Clear button - reset the number (and number display string)
78    public void processClear()
79    {
80        // clear the number stored in the model
81        number = 0;
82        display1 = "";
83        display();  // update the GUI
84    }
85
```

Model class Banking buttons

- These three methods handle the banking functions
- They only work if you are in the 'logged in' state
- They operate by calling the corresponding methods in the Bank object (which will then call methods in the BankAccount object, which you need to write 🏝)
- They then update the messages and call display to update the View

```
// Withdraw button - check we are logged in and if so try and withdraw some money from
// the bank (number is the amount showing in the interface display)
public void processWithdraw()
{
    if (state.equals(LOGGED_IN) ) {
        if ( bank.withdraw( number ) )
        {
            display2 =   "Withdrawn: " + number;
        } else {
            display2 =   "You do not have sufficient funds";
        }
        number = 0;
        display1 = "";
    } else {
        initialise("You are not logged in");
    }
    display();  // update the GUI
}

// Deposit button - check we are logged in and if so try and deposit some money into
// the bank (number is the amount showing in the interface display)
public void processDeposit()
{
    if (state.equals(LOGGED_IN) ) {
        bank.deposit( number );
        display1 = "";
        display2 = "Deposited: " + number;
        number = 0;
    } else {
        initialise("You are not logged in");
    }
    display();  // update the GUI
}

// Balance button - check we are logged in and if so access the current balance
// and display it
public void processBalance()
{
    if (state.equals(LOGGED_IN) ) {
        number = 0;
        display2 = "Your balance is: " + bank.getBalance();
    } else {
        initialise("You are not logged in");
    }
    display();  // update the GUI
}
```

Model class Last few methods

- processFinish logs you out of the ATM (if you are logged in) and changes state back to ACCOUNT_NO
- processUnknownKey is there just in case an unknown key is pressed
- Display is the method which tells the View to update, and redisplay the current state of the model

```
178     // Finish button - check we are logged in and if so log out
179     public void processFinish()
180     {
181         if (state.equals(LOGGED_IN) ) {
182             // go back to the log in state
183             setState(ACCOUNT_NO);
184             number = 0;
185             display2 = "Welcome: Enter your account number";
186             bank.logout();
187         } else {
188             initialise("You are not logged in");
189         }
190         display();  // update the GUI
191     }
192
193     // Any other key results in an error message and a reset of the GUI
194     public void processUnknownKey(String action)
195     {
196         // unknown button, or invalid for this state - reset everything
197         Debug.trace("Model::processUnknownKey: unknown button \"" + action + "\", re-initialising");
198         // go back to initial state
199         initialise("Invalid command");
200         display();
201     }
202
203     // This is where the Model talks to the View, by calling the View's update method
204     // The view will call back to the model to get new information to display on the screen
205     public void display()
206     {
207         Debug.trace("Model::display");
208         view.update();
209     }
210 }
```

Bank classes

Bank class
Set-up methods

- The Bank class maintains an array of accounts, and a ‘current’ account
- makeBankAccount is a method to be used to create a new account (instead of using ‘new BankAccount’ directly)
- addBankAccount adds a new account to the bank (if there is space)
- It has two forms, one where you have already made a BankAccount object, and one where you want it to make the BankAccount and add it all in one go (for convenience).

```
11 public class Bank
12 {
13     // Instance variables containing the bank information
14     int maxAccounts = 10;        // maximum number of accounts the bank can hold
15     int numAccounts = 0;         // the number of accounts currently in the bank
16     BankAccount[] accounts = new BankAccount[maxAccounts];  // array to hold the bank accounts
17     BankAccount account = null; // currently logged in acccount ('null' if no-one is logged in)
18
19     // Constructor method - this provides a couple of example bank accounts to work with
20     public Bank()
21     {
22         Debug.trace( "Bank::<constructor>");
23
24
25     }
26
27     // a method to create new BankAccounts - this is known as a 'factory method' and is a more
28     // flexible way to do it than just using the 'new' keyword directly.
29     public BankAccount makeBankAccount(int accNumber, int accPasswd, int balance)
30     {
31         return new BankAccount(accNumber, accPasswd, balance);
32     }
33
34     // a method to add a new bank account to the bank - it returns true if it succeeds
35     // or false if it fails (because the bank is 'full')
36     public boolean addBankAccount(BankAccount a)
37     {
38         if (numAccounts < maxAccounts) {
39             accounts[numAccounts] = a;
40             numAccounts++ ;
41             Debug.trace( "Bank::addBankAccount: added " +
42                          a.accNumber +" "+ a.accPasswd +" £"+ a.balance);
43             return true;
44         } else {
45             Debug.trace( "Bank::addBankAccount: can't add bank account - too many accounts");
46             return false;
47         }
48     }
49
50     // a variant of addBankAccount which makes the account and adds it all in one go.
51     // Using the same name for this method is called 'method overloading' - two methods
52     // can have the same name if they take different argument combinations
53     public boolean addBankAccount(int accNumber, int accPasswd, int balance)
54     {
55         return addBankAccount(makeBankAccount(accNumber, accPasswd, balance));
56     }
```

Bank class
Login methods

- login tries to log into the bank with the given account and password
- So it combines authentication (checking the account/password is valid) with logging in
- It should search the bank account array for an account matching the account number and password and if it finds one, it sets that as the current account and returns true. Otherwise it returns false
- You need to write the code to do this – that is the first part of the lab!
- logout logs the user out, and loggedIn tests whether anyone is logged in or not.

```
// Check whether the current saved account and password correspond to
// an actual bank account, and if so login to it (by setting 'account' to it)
// and return true. Otherwise, reset the account to null and return false
// YOU NEED TO ADD CODE TO THIS METHOD FOR THE LAB EXERCISE
public boolean login(int newAccNumber, int newAccPasswd)
{
    Debug.trace( "Bank::login: accNumber = " + newAccNumber);
    logout(); // logout of any previous account

    // search the array to find a bank account with matching account and password.
    // If you find it, store it in the variable currentAccount and return true.
    // If you don't find it, reset everything and return false

    // YOU NEED TO ADD CODE HERE TO FIND THE RIGHT ACCOUNT IN THE accounts ARRAY,
    // SET THE account VARIABLE AND RETURN true


    // not found - return false
    account = null;
    return false;
}

// Reset the bank to a 'logged out' state
public void logout()
{
    if (loggedIn())
    {
        Debug.trace( "Bank::logout: logging out, accNumber = " + account.accNumber);
        account = null;
    }
}

// test whether the bank is logged in to an account or not
public boolean loggedIn()
{
    if (account == null)
    {
        return false;
    } else {
        return true;
    }
}
```

BankAccount class

- BankAccount is a data class which stores an account number, password and balance
- It has a constructor which allows you to set all three
- It also has three methods for withdrawing, depositing and checking balance
- These do not work – you need to write the code for each of them as part of the lab.

```
public class BankAccount
{
    public int accNumber = 0;
    public int accPasswd = 0;
    public int balance = 0;

    public BankAccount()
    {
    }

    public BankAccount(int a, int p, int b)
    {
        accNumber = a;
        accPasswd = p;
        balance = b;
    }

    // withdraw money from the account. Return true if successful, or
    // false if the amount is negative, or less than the amount in the account
    public boolean withdraw( int amount )
    {
        Debug.trace( "BankAccount::withdraw: amount =" + amount );

        // CHANGE CODE HERE TO WITHDRAW MONEY FROM THE ACCOUNT
        return false;
    }

    // deposit the amount of money into the account. Return true if successful,
    // or false if the amount is negative
    public boolean deposit( int amount )
    {
        Debug.trace( "LocalBank::deposit: amount = " + amount );
        // CHANGE CODE HERE TO DEPOSIT MONEY INTO THE ACCOUNT
        return false;
    }

    // Return the current balance in the account
    public int getBalance()
    {
        Debug.trace( "LocalBank::getBalance" );

        // CHANGE CODE HERE TO RETURN THE BALANCE
        return 0;
    }
}
```

Lab exercises
Week 2.02

Week 2.02 lab work - ATM

- Run the ATM project and explore the code a little
- You can try and login using account number 10001 and password 11111. These credentials should work, but they don’t because there is code missing in the Bank class (in the login method – see slide 37, above)
- Once you have fixed that and can log in, you will find that none of the banking functions work. This is because you also need to add code in the BankAccount class (in the withdraw, deposit, and balance methods).
- Remember, we can help you with this lab work. Once the basic ATM is working, you are ready to try things on your own if you want to use the ATM for your project.