Waiter System – Practical Workshop 7

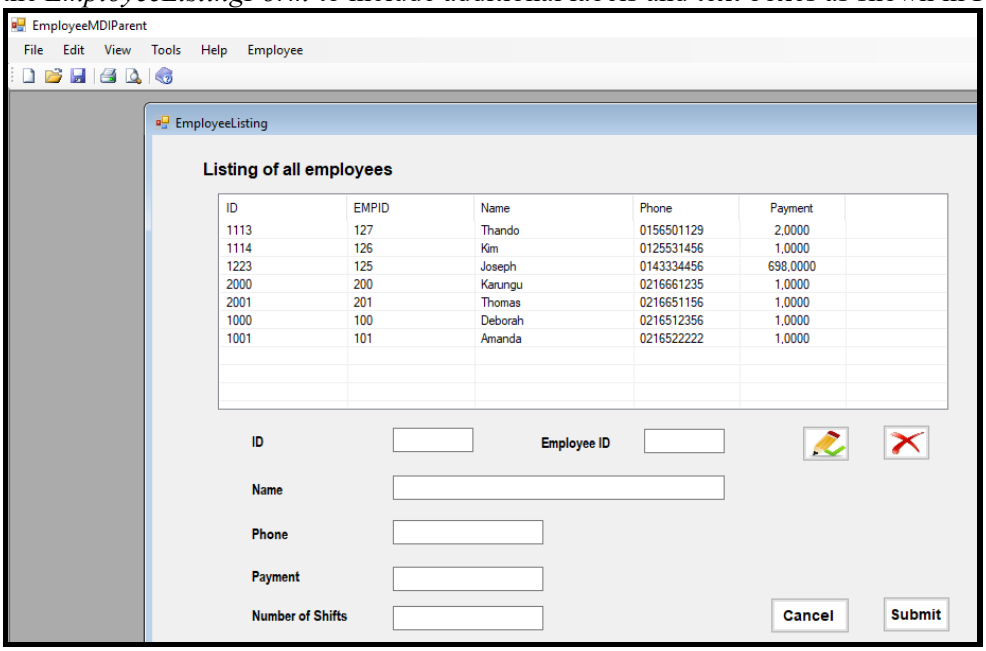
Objectives

In computer programming CRUD is the acronym for the basic functionality of **create** (add), **read** (list), **update** (edit) and **delete**. In the past two workshops we have seen how to add an employee and how tolist the current employees whose information are stored in three tables in a database. The main objective for Workshop 7 is to prepare our program to allow the user to perform the CRUD functionalities for the employee system by extending what was already done in the previous workshops sessions

Prepare the program to include the CRUD functionalities

1. We will make changes to the *EmployeeMDIParent* form to allow CRUD Functionalities. Please note that you cannot edit or delete an employee if you have not first selected that particular employee.

1.1 Adapt the *EmployeeListingForm* to include additional labels and text boxes as shown in Figure 1.



1.2 Select two appropriate icons to represent the Edit and Delete functionality. You can choose any suitable icon and save them as picture files in the Debug folder of your application. Choose two buttons and set the *BackGroundImage* property of each button to the respective icon. Use the *BackgroundImageLayout* ***Stretch*** property to size the icon to fit exactly on the button.

1.3 We would like to adapt the *EmployeeListingForm* to be in different states for view, edit and delete capability – For this it is necessary to add an **enum** list to represent the form states. The *FormStates* will start at View=0, Add= 1, etc. (we include the Add state as this could be used at a later stage).

- Create an enum *FormStates* that will capture the different states for view, Add, Edit and Delete capability. The *FormStates* will start at View = 0, Add = 1, etc.
- Declare a variable of this enum type, called *state*, at form level.
- Initialise this variable in the constructor to the View state.

2. We will add Utility methods to show or hide the textboxes and the labels for good presentation/user experience. You only want the textboxes and the labels to show, once you selected a particular employee that shouldbe displayed. Create a region titled *Utility methods* for this purpose.

2.1 Use the same function *ShowAll* as in the Utility section of the *EmployeeForm* - Adapt the code for the *EmployeeListForm,* not to include RadioButtons.

- If the form state is View, the Submit button and the Edit button should not be visible.
- Make sure, depending on the role, the *shiftslabel* control and the *Shifts* Textbox control are made visible

2.2 Use the same function *ClearAll* as in the Utility section of the *EmployeeForm* to clear the controls.

2.3 Add a private void *EnableEntries* method which receives a variable *value* of type boolean as parameter. This method can be used to ensure that certain fields (textboxes) cannot be accessed(changed) when the form is in the **Edit** state. For example, one would prevent the user from changing a textbox content that represents (displays)a primary key (or keys, like the ID and the EMPID). Implement the *EnableEntries* method with the following logic: Complete the code**.**

```
//2.3.1        write the code for the Method header
        {
          if ((//2.3.2 write the code to check if the form is in an Edit state) && value)
            {
             // 2.3.3  write the code to ensure that both the ID Textbox control and the employeeID text box control
are not enabled. The ID Textbox control is given as an example
                        idTextBox.Enabled = !value;
            }
            else
            {
                 //2.3.4 write the code to ensure both the ID Textbox control and the employeeID text box control
                  should be enabled
            }
                 //2.3.5 write the code to enable the following controls: phone, payment and shifts - an example is
given for the name Text box control
                  nameTextBox.Enabled = value;

            //2.3.6    write the code to check if the form is in a Delete state
            {
              //2.3.7 write the code to ensure the cancelButton control and the Submit Button control are not visible.

            }
            // 2.3.8   if the form is not in a delete state (- write the code for this)
            {
             //2.3.9   write the code to ensure the cancelButton control and the Submit Button control are visible.
            }
      } //end of the EnableEntries method
```

3. Now we will focus on the selection of the Employee. Once you click on the ID of an Employee in the *EmployeeListView* Control the *employeeListView_SelectedIndexChanged* event is fired. When this happens, you must display the details of this employee on the form in a **View** state. For this you need to find the specific employee object with this ID number in the employees collection.

3.1 In the *EmployeeController* class, add a **Find** method in the *Search Method* region of the *EmployeeController* class. The method received the ID of the employee you are looking for and returns the employee object. The code is given below but it is not complete. You are to complete the code.

```
public Employee Find(string ID)
  {
   3.1.1: write the code to declare a variable index and initialise it to 0
   3.1.2: write the code to declare a variable found of type Boolean. Initialise this variable to the following:
               (employees[index].ID == ID);  //checks if it is the first employee. The found variable will be
                                                 searching for an employee
   3.1.3: write the code to declare a variable count. Initialise this variable to the count of employees in the
          collection.

   while (!(found) && (index < employees.Count - 1)) //if you have not found the employee AND you have not
                                                        reached the end of the collection – write
          {
      3.1.4 Write the code to increment the index variable
      3.1.5 Write the code to assign found to (employees[index].ID == ID); // this will be TRUE if found
          }
          return the employee // we have found the employee
      }
```

3.2 In the *EmployeeListingForm* class**:** Add a PopulateTextboxes method that will receive an employee object and assign each attribute of this object to the appropriate textbox on the form. The code is given below but it is not complete. You are to complete the code.

```
private void PopulateTextBoxes(Employee employee)
    {
      HeadWaiter headW;
      3.2.1 do the same for the other employee roles

      idTextBox.Text = employee.ID;
      3.2.2 do the same for the other general fields
      switch (employee.role.getRoleValue)
      {
        case Role.RoleType.Headwaiter:
          headW = (HeadWaiter)(employee.role);
          paymentTextBox.Text = Convert.ToString(headW.SalaryAmount);
           break;
          3.2.3 do the same for the other employee roles
       }
   }
```

3.3 Double click on the *employeeListView* control to create the outline of an *employeeListView_SelectedIndexChanged* event. In the *employeeListView_SelectedIndexChanged* event:

3.3.1 Call the *ShowAll* method with the Boolean value set to true and the *roleValue*.

3.3.2 Set the form state to **View**.

3.3.3 Call the *EnableEntries* method with the boolean value set to **false**.

3.3.4 If an entry has been selected, call the *Find* method (of the EmployeeController class) and assign it to the employee object (declare at form level if you have not done so). The code is given below but it is not complete. You are to complete the code.

```
if (employeeListView.SelectedItems.Count > 0)  // if you selected an item
    {
       employee =
       employeeController.Find(employeeListView.SelectedItems[0].Text);
       //3.3.4.1: call the PopulateTextboxes method for this employee
    }
```

Run your program and you should be able to select an employee from the listview and this will automatically populate your controls. If there are any issues, debug your program.