CSC320-A

Module 5

Implementing an MVC-Based Application

1 Views & Controllers

1.1 Designing the View

We will implement a view named **AccountsView** in **SymBank.Banking** project. Add a separate folder named **Views** for organizing the views. Set the default **Caption** in the constructor.

Creating a basic view: Views\AccountsView.cs

```
namespace SymBank.Banking.Views {
    public partial class AccountsView : BaseView {
        public AccountsView() {
            InitializeComponent();
            Caption = "Accounts";
        }
    }
}
```

Add a **Resources** file to store the following icon. The icon is used in a **Command** to open **AccountsView** view. The **Init** method will register the command into the **File** menu. You should now to be able to access the accounts view.

Adding icon for command to resources

```
RSC/icons/application/application_form.png
```

Creating and registering command to open view: BankingModule.cs

```
public static Command Accounts = new Command {
    Caption = "&Accounts",
    Description = "Manage accounts",
    Icon = Resources.ApplicationForm,
    Action = command => {
        var view = new AccountsView();
        view.Show("TabSpace");
    }
};

public override void Init() {
    Shell.Sites["FileMenu"].Add(Accounts);
}
```

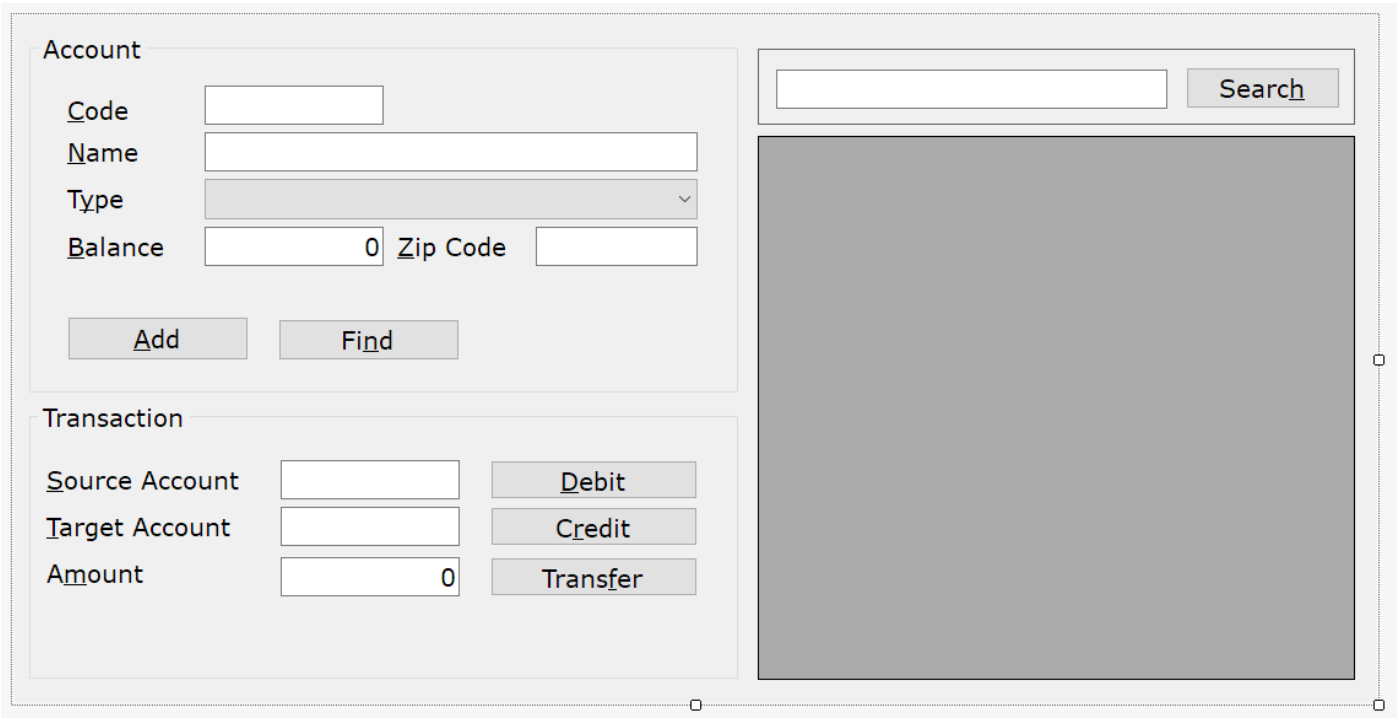
Before we start adding controls to the form, you can setup the properties of the form first. Select the form and then use the **Properties** window to setup the properties for the form. If the window is not visible, you can press **F4** to open it. The following is the basic list of properties to configure for the form.

View properties

```
Font            :   Verdana, 12 pt
StartPosition   :   CenterScreen
```

Once the form has been setup, add the controls from the **Toolbox** window and setup their properties using the following information. Note that you should use the **Tab Order** option from **View** menu to set the **TabIndex** property rather than using the **Properties** window directly. Add two **GroupBox** and one **Panel** controls to separate the form into 3 sections; one section to add and view an account, while the other to perform account transactions and the last one to search for accounts by name.

Visual appearance of the view



Frame controls

```
grpAccount      Text: Account           TabIndex: 0
grpTransaction  Text: Transaction       TabIndex: 1
pnlSearch                               TabIndex: 2
```

In the first group, place 5 **Label**, 4 **TextBox** controls, a **ComboBox** and 2 **Button** controls as shown in the visual representation above. Then configure the controls as shown in the next page.

Label controls

```
lblCode     Text: &Code
lblName     Text: &Name
lblType     Text: T&ype
lblBalance  Text: 0
lblZipCode  Text: &Zip Code
lblBalance  Text: &Balance
```

TextBox controls

```
txtCode     MaxLength    :  4
txtName     MaxLength    : 30
txtBalance  MaxLength    : 10    Text: 0 TextAlign    : Right
txtZipCode  MaxLength    : 5
```

ComboBox control

```
cbxType     DropDownStyle    : DropDownList
            Items            :
             –   Savings
             –   Checking
             –   Loan
```

Button controls

```
btnAdd  Text: &Add
btnFind Text: &Find
```

In the second group box, add and configure the following controls. Ensure that the shortcut key assigned using the **&** in the Text property for Labels and Buttons do not clash with the ones already assigned in the above controls.

Label controls

```
lblSource    Text:   &Source Account
lblTarget    Text:   &Target Account
lblAmount    Text:   A&mount
```

TextBox controls

```
txtSource MaxLength :  4
txtTarget MaxLength :  4
txtAmount MaxLength : 10    Text : 0 TextAlign : Right
```

Button controls

```
btnDebit    Text:   &Debit
btnCredit   Text:   C&redit
btnTransfer Text:   Trans&fer
```

Add the following controls into tha panel. Finally, you can add a **DataGridView** and an additional Button control on the right side of the form. The grid is to display the list of accounts when the button is clicked.

Other Controls

```
txtSearch   Text : {blank}
btnSearch   Text : Searc&h
```

DataGridView control

```
grdAccounts ReadOnly                : True
            AllowUsersToAddRows     : False
            AllowUsersToDeleteRows  : False
```

If you allow the form to be resized, controls have to be anchored or docked in order to be automatically resized to fit the view. If the view is expected to be large, you can enable **AutoScroll** property so that scrollbars will appear if all the controls cannot fit into the visible area of the view.

Anchoring for controls

```
grpAccount      Anchor : Top + Left
grpTransaction  Anchor : Top + Left + Bottom
pnlSearch       Anchor : Top + Left + Right
grdAccounts     Anchor : Top + Left + Right + Bottom
txtSearch       Anchor : Top + Left + Right
btnSearch       Anchor : Top + Right
```

1.2 Accessing Services

The view will use property injection to get access to the required services. Following shows the view requesting the injection of **IAuthorization**, **IAccountController** and **ITransactionController** services.

Getting access to services

```
[Inject]public IAuthorization Authorization { get; set; }
[Inject]public IAccountController AccountController { get; set; }
[Inject]public ITransactionController TransactionController { get; set; }

public AccountsView() {
    InitializeComponent();
    Caption = "Accounts";
    this.Inject();
}
```

While you can automatically close the view after performing an operation, if you wish to remain open you may sometimes want to clear or reset part of the view. Following are two methods to clear controls in each **GroupBox**.

Helper methods to clear input

```
private void ClearAccount() {
    txtCode.Text = string.Empty;
    txtName.Text = string.Empty;
    cbxType.SelectedIndex = 0;
    txtZipCode.Text = string.Empty;
    txtBalance.Text = "0";
    txtCode.Focus();
}

private void ClearTransaction() {
    txtSource.Text = string.Empty;
    txtTarget.Text = string.Empty;
    txtAmount.Text = "0";
    txtSource.Focus();
}
```

Add event handlers for all the buttons in the view and implement them as shown by the following code. UI is the frontend so it is always responsible to catch and handle runtime errors. We can use the **Status** property or **MessageBox** methods exposed through **IShell** interface to display non-instrusive and instrusive messages.

Adding a new account

```
private void btnAdd_Click(object sender, System.EventArgs e) {
    try {
        var item = new Account {
            Code = int.Parse(txtCode.Text),
            Name = txtName.Text,
            Type = cbxType.SelectedIndex,
            ZipCode = txtZipCode.Text,
            Balance = decimal.Parse(txtBalance.Text)
        };
        AccountController.Add(item);
        var message = $"Account {item.Code} added.";
        Shell.Status = message;
        Shell.Success(message);
        ClearAccount();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot add account. " + ex.Message);
    }
}
```

Locating an existing account

```
private void btnFind_Click(object sender, System.EventArgs e) {
    try {
        var item = AccountController.GetAccount(int.Parse(txtCode.Text));
        if (item == null) throw new Exception("Invalid account ID.");
        txtCode.Text = item.Code.ToString();
        txtName.Text = item.Name;
        cbxType.SelectedIndex = item.Type;
        txtZipCode.Text = item.ZipCode;
        txtBalance.Text = item.Balance.ToString();
        Shell.Status = $"Found account {item.Code}.";
        txtCode.Focus();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot find account. " + ex.Message);
        ClearAccount();
    }
}
```

Search for accounts by name

```
private void btnSearch_Click(object sender, System.EventArgs e) {
    try {
        var name = txtSearch.Text.Trim();
        var results = AccountController.GetAccountsForName(name);
        grdAccounts.DataSource = results;
    }
    catch (Exception ex) {
        Shell.Failure("Search failed. " + ex.Message);
    }
}
```

Debiting an account

```
private void btnDebit_Click(object sender, System.EventArgs e) {
    try {
        var source = int.Parse(txtSource.Text);
        var amount = decimal.Parse(txtAmount.Text);
        TransactionController.Debit(source, amount);
        var message = $"Account {source} debited with {amount:N2}";
        Shell.Success(message);
        Shell.Status = message;
        ClearTransaction();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot debit account. " + ex.Message);
    }
}
```

Crediting an account

```
private void btnCredit_Click(object sender, System.EventArgs e) {
    try {
        var source = int.Parse(txtSource.Text);
        var amount = decimal.Parse(txtAmount.Text);
        TransactionController.Credit(source, amount);
        var message = $"Account {source} credit with {amount:N2}";
        Shell.Success(message);
        Shell.Status = message;
        ClearTransaction();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot credit account. " + ex.Message);
    }
}
```

Transfer money between accounts

```
private void btnTransfer_Click(object sender, System.EventArgs e) {
    try {
        var source = int.Parse(txtSource.Text);
        var target = int.Parse(txtTarget.Text);
        var amount = decimal.Parse(txtAmount.Text);
        TransactionController.Credit(source, amount);
        var message = $"Transfered {amount:N2} from account {source} to {target}.";
        Shell.Success(message);
        Shell.Status = message;
        ClearTransaction();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot transfer amount. " + ex.Message);
    }
}
```

1.3 Binding Source

It may be a bit cumbersome to transfer data between an **Account** object and controls in the view. You can add a **BindingSource** component to the view and then map the **DataSource** property to a data model. You can then bind properties of controls to the model properties.

Add a **BindingSource** named **accountSource** and set **DataSource** to the **Account** model. Then go to the **TextBox** controls in the first group box and use the *Advanced* setting in *DataBindings* of each control to bind **Text** property to the relevant property in an **Account** model. However, there are limitations, not all properties are supported like you cannot bind to **SelectedIndex** property of **ComboBox**, thus we still have to manually set the control property directly. Once the bindings is done, declare a field in the view to be used to hold on to the bound object.

Field for the binding data source

```
private Account _account;
```

Update **ClearAccount** method to create a new object and assign it to binding source instead. We do not need to update controls directly except for control properties that are not supported in data-binding. You can call this method from the constructor to create a new object for the initial binding.

Clear account creates a new binding source object

```
private void ClearAccount() {
    _account = new Account();
    accountSource.DataSource = _account;
    cbxType.SelectedIndex = _account.Type;
    txtCode.Focus();
}
```

Create the initial data source object

```
public AccountsView() {
    InitializeComponent();
    Caption = "Accounts";
    this.Inject();
    ClearAccount();
}
```

Event handlers for **Add** and **Find** buttons are simplified since for most of the controls the data transfer between controls and the **Account** object is done for you so you can simply just pass the object to the services for processing.

Simplified Add button event handler

```
private void btnAdd_Click(object sender, System.EventArgs e) {
    try {
        _account.Type = cbxType.SelectedIndex;
        AccountController.Add(_account);
        var message = $"Account {_account.Code} added.";
        Shell.Status = message;
        Shell.Success(message);
        ClearAccount();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot add account. " + ex.Message);
    }
}
```

Simplified Find button event handler

```
private void btnFind_Click(object sender, System.EventArgs e) {
    try {
        var item = AccountController.GetAccount(int.Parse(txtCode.Text));
        if (item == null) throw new Exception("Invalid account ID.");
        cbxType.SelectedIndex = item.Type;
        accountSource.DataSource = _account = item;
        Shell.Status = $"Found account {item.Code}.";
        txtCode.Focus();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot find account. " + ex.Message);
        ClearAccount();
    }
}
```

1.4 Validation

Input controls has a **CausesValidation** property. When set to **true**, when the value in the control is changed, a **Validating** event and a **Validated** event will occur. You can add an **ErrorProvider** component to display error messages next to controls. For this to work, you must ensure that the default value is valid.

Default account name

```
private void ClearAccount() {
    _account = new Account { Name = "New Account" };
    accountSource.DataSource = _account;
    cbxType.SelectedIndex = _account.Type;
    txtCode.Focus();
}
```

When validation fails, set **Cancel** event argument to true. This will cancel any button **Click** event that causes validation. However, you should turn off **CausesValidation** on buttons that doesn't require validation. For example, the **Add** button would require validation but not the **Find** button.

Validating account name changed

```
private void txtName_Validating(object sender, CancelEventArgs e) {
    var text = txtName.Text.Trim();
    if (text.Length <= 3) {
        errorProvider.SetError(txtName,
            "Account name must be more the 3 characters.");
        e.Cancel = true;
    }
}
```

Remove error message on validated

```
private void txtName_Validated(object sender, EventArgs e) {
    errorProvider.SetError(txtName, null);
}
```

1.5 Custom Exceptions

Generally an **Exception** contains an error message and possibly an inner exception. If you want an exception to contains more information, such as an error code or the actual control that failed input validation, you can always create your own exception class. In the following example, we implemented a new exception class specifically for input validation errors.

Custom exception class: Symbion\ValidationException.cs

```
namespace Symbion {
    public class ValidationException : Exception {
        private Control _control;

        public ValidationException(Control control, string message) : base(message) {
            _control = control;
        }

        public void Refocus() {
            if (_control != null) _control.Focus();
        }
    }
}
```

We will implement a **Validation** class that contain methods to help us do validation and a **ValidationException** will be thrown upon validation failure.

Beginning a validation class: Symbion\Validation.cs

```
namespace Symbion {
    public static class Validation {
        public const string CannotBeNull = "{0} cannot be null.";
        public const string CannotBeEmpty = "{0} cannot be empty.";
        public const string OutOfRange = "{0} is must be between {1} and {2}.";
        public const string IsNotValid = "{0} is not valid.";
    }
}
```

Must not be null or empty

```
public static void NotNull(this Control control, string name) {
    if (control.Text == null) throw
        new ValidationException( control,
            string.Format(CannotBeNull, name));
}

public static void NotNullOrEmpty(this Control control, string name) {
    NotNull(control, name);
    if (control.Text.Length == 0) throw
        new ValidationException(control,
            string.Format(CannotBeEmpty, name));
}
```

Checking number within a range

```
public static void InRange(this Control control,
    string name, int minValue, int maxValue) {
    int value;
    if (!int.TryParse(control.Text, out value)) throw
            new ValidationException(control, string.Format(IsNotValid, name));
    if (value < minValue || value > maxValue) throw new ValidationException(
        control, string.Format(OutOfRange, name, minValue, maxValue));
}

public static void InRange(this Control control,
    string name, decimal minValue, decimal maxValue) {
    decimal value;
    if (!decimal.TryParse(control.Text, out value)) throw
            new ValidationException(control, string.Format(IsNotValid, name));
    if (value < minValue || value > maxValue) throw new ValidationException(
        control, string.Format(OutOfRange, name, minValue, maxValue));
}
```

Using Regular Expressions

```
public static void Matches(this Control control, string name, string pattern) {
        if (!new Regex(pattern).IsMatch(control.Text))
                throw new ValidationException(control, string.Format(
                    IsNotValid, name));
        }
    }
}
```

Using **Validation** and **ValidationException**

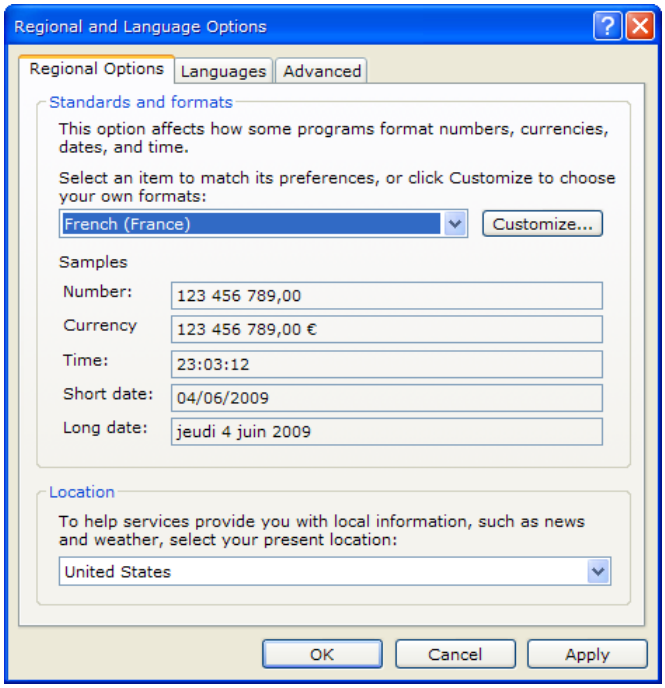
```
private void btnAdd_Click(object sender, System.EventArgs e) {
    try {
        txtBalance.InRange("Balance", 100m, decimal.MaxValue);
        _account.Type = cbxType.SelectedIndex;
        AccountController.Add(_account);
        var message = $"Account {_account.Code} added.";
        Shell.Status = message;
        Shell.Success(message);
        ClearAccount();
    }
    catch (ValidationException ex) {
        Shell.Failure("Validation failed. " + ex.Message);
        ex.Refocus();
    }
    catch (Exception ex) {
        Shell.Failure("Cannot add account. " + ex.Message);
    }
}
```

2 Globalization

2.1 CultureInfo

Each instance of the **Thread** class has two properties named **CurrentCulture** and **CurrentUICulture**. The **CurrentCulture** property is used to determine what default culture to use for formatting dates and numbers. Initially **CurrentCulture** will follow the same settings as the Regional and Language Options in the Control Panel. In the following example, we have set the default culture to French for France. If you wish to access and manipulate culture inside the application import **System.Globalization** library namespace and use **CultureInfo** type to store and access culture information.

Changing CurrentCulture



Display the names for the current culture: Culture1\Program.cs

```
Thread t = Thread.CurrentThread;
CultureInfo c1 = t.CurrentCulture;
MessageBox.Show(
    c1.Name + '\n' +
    c1.DisplayName + '\n' +
    c1.NativeName + '\n' +
    c1.EnglishName);
```

Formatting automatically uses CurrentCulture

```
decimal v1 = 2987.5m;
DateTime v2 = DateTime.Now;
MessageBox.Show(
    string.Format(
    "{0:N2}\n{0:C2}\n" +
    "{1:ddd dd,mmm yyyy}\n" +
    "{1:dddd dd,mmmm yyyy}",v1, v2));
```

It is possible to support multiple cultures. Even though the default is already provided through the **CurrentCulture** property, additional cultures can be retrieved and used explicitly for formatting. Note that both the language and country is important when the culture is used for formatting.

Accessing and using explicit cultures

```
CultureInfo c2 = new CultureInfo("en-GB");
CultureInfo c3 = CultureInfo.CreateSpecificCulture("ms-MY");
MessageBox.Show(
    string.Format(c2,
        "{0:N2}\n{0:C2}\n" +
        "{1:ddd dd,MMM yyyy}\n" +
        "{1:dddd dd,MMMM yyyy}", v1, v2));
MessageBox.Show(
    string.Format(c3,
        "{0:N2}\n{0:C2}\n" +
        "{1:ddd dd,MMM yyyy}\n" +
        "{1:dddd dd,MMMM yyyy}", v1, v2));
```

The **CultureInfo** object stores all the required information for formatting dates and numbers. If you wish, you can access the information directly.

Accessing culture information

```
MessageBox.Show(
    string.Format(
        "Currency symbol:{0}\n" +
        "First day of week:{1}\n" +
        "First month of year:{2}",
        c3.NumberFormat.CurrencySymbol,
        c3.DateTimeFormat.DayNames[0],
        c3.DateTimeFormat.MonthNames[0]));
```

2.2 UI Culture

Controls and resource manager does not use **CurrentCulture** property for rendering content and retrieving resources. **CurrentUICulture** property will be used instead for these operations. However, this property follow the culture of the Windows operating system rather then the regional settings in the Control Panel. It would be difficult for users from different countries to use the same operating system if it is localized to only one language. Microsoft provides a **Multilingual UI Pack** then can be installed on the operating system and allow individual users to select their own language on the same system. The UI would adapt to the selected language.

Even if you only have a localized version of the operating system without MUI you can still make the controls and resource manager use the regional settings by setting **CurrentCUICulture** to **CurrentCulture** as shown below.

Making UI Culture follow regional settings: SymBank\Program.cs

```
static void Main() {
    var thread = Thread.CurrentThread;
    thread.CurrentUICulture = thread.CurrentCulture;
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new ShellForm());
}
```

2.3 Localizing Resources

To localize resources in an assembly you need to make a copy of the resources file. The file can be added to the same project or compiled in a different project. The most important part is the filename. The name of the file must be the same as the original file except that the culture code of the localized version needs to be attached to the end of the filename. Language code is more important since some countries speak the same language and thus can use back the same file. If you attach the country code, you would need a separate file for each country. The following shows how to name localized resources files. Two letter ISO codes will be used. You can easily find these codes on the Internet by searching for two letter ISO language code and two letter ISO country code.

Resources source filenames

```
Default     BaseFileName.resx                   Resources.resx
Language    BaseFileName.language.resx          Resources.fr.resx
Country     BaseFileName.language-COUNTRY.resx  Resources.fr-CA.resx
```

When you build the project, all default resources are compiled together in the same assembly. However, localized resources for different languages and different countries are compiled as separate assemblies that we call as satellite assemblies. They are normally compiled as libraries and have the same name as the target assembly but extended with the word resources. Each file is placed in a subfolder named according to culture. These files and folders can be removed. If the localized file is not found, the default resource will be used.

Resources target assemblies

```
Default     TargetName.dll/.exe
Language    Language\TargetName.resources.dll/.exe
Country     Language-COUNTRY\TargetName.resources.dll/.exe
```

Example of resources for Symbion project

```
Default     Symbion.dll
Language    fr\Symbion.resources.dll
Country     fr-CA\Symbion.resources.dll
```

To test localization of resources, make a copy of the **Resources** file in **SymBank**. Delete the designer generated code file. Open the **Properties** window for the file and remove the *ResXFileCodeGenerator* setting in the *Custom Tool* property so that Visual Studio does not generate any code for the localized file since the code generated for the default will work with the localized resources as long as the correct UI culture is set. We wish to localize this file to French so change filename to **Resources.fr.resx** and then translate the messages to the correct language.

Resources localized to French: SymBank\Properties\Resources.fr.resx

```
Confirm             Confirmation
Failure             Erreur
InitializingModules Initialisation des modules...
LoadingModules      Chargement des modules...
PleaseWait          S'il vous plaît, attendez..
Ready               Prêt.
Success             Information
Warning             Alerte
```

2.4 UI Localization

Forms and user controls can be localized very easily from Visual Studio. Prepare the form and the user control in the default language. When completed, select form or user control and open the **Properties** window. You will be able to see a **Localizable** property. Set the property to true and switch over to a language using the **Language** property before changing the UI. Note to reset back to **Default** once completed. The following shows the view localized to **French**.

Localizing the user-interface to French