CSC320-A

Module 3

Implementing a Windows Forms Application Framework (Part 1)

1 Application & Shell

1.1 Project Setup

We begin by creating the project **SymBank** to build a GUI application. Choose the **Windows Forms Application** template and enter the project name and location as shown below. Once the project is created, we will perform some common operations to prepare the project that includes assigning application information, signing the assembly, assigning an application icon and changing the main form name and icon.

New Project Information

```
Project Name: SymBank
Project Type: Visual C# | Windows | Windows Forms Application
Location    : C:\CSC320-A\SRC\
Solution    : Module3
```

Assign the following application information to the target assembly. Then assign the icon file provided by the instructor to the application and then sign it with a new key file named **SymBank**. You may then choose to move the icon and the key file to the **Properties** folder. Change form class name to **ShellForm** and assign the application icon to the form as well. Then set the **Text** property to **SymBank** and **StartPosition** to **CenterScreen**.

Application information: SymBank\Properties\AssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("SymBank")]
[assembly: AssemblyDescription("Example smart client application.")]
[assembly: AssemblyCopyright("Copyright © Symbolicon Systems 2022")]
[assembly: AssemblyProduct("Symbolicon SymBank")]
[assembly: AssemblyCompany("Symbolicon Systems")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: Guid("C02F7865-1DB3-4CD8-9875-E74C7B337BC8")]
[assembly: ComVisible(false)]
```

Once an assembly is compiled and signed, any tampering on the executable or library binary file will be detected by .NET and it will not be loaded. An unsigned application can used signed or unsigned libraries but a signed application required all the libraries referenced to be signed. Copy **SymBank** key file to **SymBank.Banking** project and sign with the same key. Create a separate **Symbion** key file for signing **Symbion** and **Symbion.Loggers** assemblies. Then reference all other projects from the **SymBank** project.

1.2 Initialization & Finalization

The bootstrapping process is the minimal process to get the application into a running state. Basic application initialization is performed during this process. If the GUI is not required at this stage, you can implement the bootstrapping in the **Main** method of the **Program** class. If a GUI is required, you can implement the bootstrapping in the **Load** event handler of the main form. Open **ShellForm** and switch to events in the **Properties** window. Locate and double-click on **Load** event to generate the handler and enter the following code. The code requires the **Modules.xml** file so add the file to the project as a content file and set the *Copy to Output Directory* option to *Copy if newer*.

SymBank bootstrapping/initialization process: ShellForm.cs

```
private void ShellForm_Load(object sender, EventArgs e) {
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
    LoggerFactory.CreateInstance().Add();
    new PrincipalAuthorization().Add<IAuthorization>();
    ModuleLoader.Load("Modules.xml");
    ModuleLoader.Init();
}
```

We may also need to perform certain operations at the end of the application. We can detect the application is about to terminate when the main form is closed. Attach an event handler to the **FormClosed** event to perform your finalization operations.

SymBank finalization process

```
private void ShellForm_FormClosed(object sender, FormClosedEventArgs e) {
    ModuleLoader.Exit();
}
```

1.3 GUI Service

A form itself can be exposed as a service to provide GUI-related services to the rest of the application. We will now add an **IShell** interface to **Symbion** library and add in the members as shown below.

GUI-related IShell service interface: Symbion\IShell.cs

```
namespace Symbion {
    public interface IShell : IService {
        string Status { set; }
        void Success(string message);
        void Failure(string message);
        void Warning(string message);
        bool Confirm(string message);
        void Close();
    }
}
```

Implementing IShell in ShellForm: SymBank\ShellForm.cs

```
public partial class ShellForm : Form, IShell {
        :
```

Implement IShell methods to display messages

```
public void Success(string message) {
    MessageBox.Show(this, message, "Information",
        MessageBoxButtons.OK, MessageBoxIcon.Information);
}

public void Failure(string message) {
    MessageBox.Show(this, message, "Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}

public void Warning(string message) {
    MessageBox.Show(this, message, "Alert",
        MessageBoxButtons.OK, MessageBoxIcon.Warning);
}

public bool Confirm(string message) {
    return MessageBox.Show(this, message, "Confirm",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question)
        == DialogResult.Yes;
}
```

To display status messages in the main form of the **SymBank** application we can add a **StatusStrip** named sbrMain to contain a **StatusLabel** named **lblStatus**. Following are the properties of these controls. Then implement the **Status** property in the form to assign the message to the label if the form is visible.

Properties for sbrMain

```
RenderMode   : ManagerRenderMode
Text         : (blank)
```

Properties for lblStatus

```
Spring       : true
Text         : (blank)
TextAlign    : MiddleLeft
Padding      : 4,4,4,4
```

Implement the IShell Status property

```
public string Status {
    set {
        if (Visible) {
            lblStatus.Text = value ?? "Ready.";
            sbrMain.Refresh();
        }
    }
}
```

1.4 SplashScreen

If there are too many modules and services to be loaded and initialized, the shell form may take a long time to appear. We can implement a splash screen that is displayed before the shell form appears during the initialization process. Add a new form class named **SplashScreen** to **SymBank** project.

Properties for SplashScreen form: SymBank\SplashScreen.cs

```
BackgroundImage : C:\CSC300-A\RSC\SymBank.jpg
FormBorderStyle : FixedDialog
ControlBox      : false
MaximizeBox     : false
MinimizeBox     : false
StartPosition   : CenterScreen
Size            : 600,400
Text            : (blank)
```

The shell form can create and show the splash screen before initialization and closes it after initialization completes. Since currently we only have one module, initialization will be too fast for us to see the splash screen so we will use an **#if** directive to insert a delay using **Thread.Sleep** only for the *DEBUG* version so we can then clearly see it during debugging.

Using the splash screen in ShellForm: ShellForm.cs

```
private SplashScreen frmSplash;

private void ShellForm_Load(object sender, EventArgs e) {
    frmSplash = new SplashScreen(); frmSplash.Show();
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
    LoggerFactory.CreateInstance().Add();
    new PrincipalAuthorization().Add<IAuthorization>();
    ModuleLoader.Load("Modules.xml");
    ModuleLoader.Init();
#if DEBUG
    Thread.Sleep(2000);
#endif
    frmSplash.Close();
    frmSplash = null;
}
```

You may want to redirect all status messages to appear in the splash screen instead during initialization. Add a **Label** control named **lblStatus** to the splash screen with the following property settings. Expose a **Status** property to set the label. Modify the **Status** property in shell form to redirect messages to the splash screen if it iexists.

Properties for lblStatus label

```
Name            : lblStatus
BorderStyle     : Fixed Single
TextAlign       : MiddleCenter
Padding         : 8,8,8,8
Size            : 584, 35
Location        :  5, 352
Text            : (blank)
```

Status property to set the label: SplashScreen.cs

```
public string Status {
    set {
        lblStatus.Text = value ?? "Please wait...";
        lblStatus.Refresh();
    }
}
```

Update ShellForm Status property: ShellForm.cs

```
public string Status {
    set {
        if (frmSplash == null) {
            lblStatus.Text = value ?? "Ready.";
            sbrMain.Refresh();
        }
        else frmSplash.Status = value;
    }
}
```

Using **#if** DEBUG directive can result in ugly code, you can implement a **Conditional** method for *DEBUG* instead in **DebugHelper** class. Any code that calls this method is only compiled in the *DEBUG* version.

Adding a conditional method: Symbion\DebugHelper.cs

```
[Conditional("DEBUG")]
public static void Delay(int duration) {
    Thread.Sleep(duration);
}
```

Displaying messages during initialization: ShellForm.cs

```
private void ShellForm_Load(object sender, EventArgs e) {
    frmSplash = new SplashScreen(); frmSplash.Show();
    AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
    LoggerFactory.CreateInstance().Add();
    new PrincipalAuthorization().Add<IAuthorization>();
    Status = "Loading modules...";
    ModuleLoader.Load("Modules.xml");
    DebugHelper.Delay(2000);
    Status = "Initializing modules...";
    ModuleLoader.Init();
    DebugHelper.Delay(2000);
    frmSplash.Close(); frmSplash = null;
    Status = null;
}
```

1.5 Resources

A Windows Forms application already has a **Resources** file added. Each form also has its own local resources file. A resources file is mainly used to store a string table but can also be used to embed icons, images and other content files into the assembly. It also allows us to globalize our application later on by adding additional resource files for each language and country that we wish to support without touching the source code. A class library normally does not have a resources file added but you can easily add it in. You can now add a **Resources** file into **Symbion** project. You can choose to move the file into the **Properties** folder if you wish. A class will be generated for the **Resources** file to access the resources from your code. Notice that class is **internal**. If you want to share the resources across assemblies, open the **Properties** window on the resources file and change *Custom Tool* to *PublicResXFileCodeGenerator*.

Let us move the strings we used in the **Load** method of **ModuleLoader** class into the resources file instead. The following is the names and strings to store in the resources file.

ModuleLoader string resources

```
CannotFindClassInModule      : Cannot find class {0} in module {1}.
CannotLocateModule           : Cannot locate module {0}.
ErrorInstancingClassInModule : Error '{0}' instancing {1} in module {2}.
ErrorLoadingModule           : Error '{0}' occurred loading module {1}.
ModuleLoaded                 : Module {0} loaded successfully.
UserNotAuthorizedForModule   : User not authorized for module {0}.
```

Each string name becomes a property in the **Resources** class. Note that the resource manager uses **CultureInfo.CurrentUICulture** by default to determine the correct resources file to access. However, string formatting use **CultureInfo.CurrentCulture** to locate the correct formatter to format date, time and numeric information.

Using strings from resources file

```
if (item.Roles.Count > 0 && !auth.IsInAnyRoles(item.Roles)) {
    Debug.WriteLine(string.Format(CultureInfo.CurrentCulture,
        Resources.UserNotAuthorizedForModule, item.Path));
        continue;
}
```

You can move the messages in **SymBank** application initialization to **Resources** file. Also move the strings for the captions for **MessageBox** using in the **IShell** methods as well. Update the initialization and the message methods to use the resource strings instead.

Application initialization string resources

```
LoadingModules      Loading modules...
InitializingModules Initializing modules...
PleaseWait          Please wait...
Ready               Ready
Success             Information
Failure             Error
Warning             Warning
Confirm             Confirmation
```

Update messages in application initialization: ShellForm.cs

```
private void ShellForm_Load(object sender, EventArgs e) {
        :
    Status = Resources.LoadingModules;
    ModuleLoader.Load("Modules.xml");
    DebugHelper.Delay(2000);
    Status = Resources.InitializingModules;
    ModuleLoader.Init();
    DebugHelper.Delay(2000);
        :
}
```

Update IShell methods to use resource strings

```
public void Success(string message) {
    MessageBox.Show(this, message, Resources.Success,
    MessageBoxButtons.OK, MessageBoxIcon.Information);
}

public void Failure(string message) {
    MessageBox.Show(this, message, Resources.Failure,
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}

public void Warning(string message) {
    MessageBox.Show(this, message, Resources.Warning,
        MessageBoxButtons.OK, MessageBoxIcon.Warning);
}

public bool Confirm(string message) {
    return MessageBox.Show(this, message, Resources.Confirm,
        MessageBoxButtons.YesNo, MessageBoxIcon.Question)
        == DialogResult.Yes;
}

public string Status {
    set {
        if (frmSplash == null) {
            lblStatus.Text = value ?? Resources.Ready;
            sbrMain.Refresh();
        }
        else frmSplash.Status = value;
    }
}
```

Update SplashScreen status: SplashScreen.cs

```
public string Status {
    set {
        lblStatus.Text = value ?? Resources.PleaseWait;
        lblStatus.Refresh();
    }
}
```

1.6 Shell Singleton

To make it easier for all services and modules to get access to **IShell** service, you can expose a singleton for it in both the **BaseService** and **BaseModule**. In this way all services and modules have access to the GUI service that we will enhance with more features as we continue implementing the application. Add the following code to both **BaseService** class and **BaseModule** class.

Exposing IShell service to services and modules

```
private static IShell _shell;

public static IShell Shell {
    get { return _shell ?? (_shell = ServiceRepository.Get<IShell>());
}
```

2 Commands

At the current moment **Symbion** is not dependent on any specific GUI framework. It will work with Windows Forms application or a WPF application. However we will need to implement classes that will reference types from a GUI framework. If you still wish to make **Symbion** non-GUI specific, you can always create a separate class library for classes that depends on a GUI framework. In this way you can support multiple GUIs by implementing multiple libraries.

Supporting multiple GUI frameworks

```
Symbion.dll          Non-GUI specific class library
Symbion.Forms.dll    Windows Forms specific extended class library
Symbion.WPF.dll      WPF specific extended class library
```

So Windows Forms applications will reference both **Symbion** and **Symbion.Forms** while a WPF application will reference **Symbion** and **Symbion.WPF** instead. However to save time, we will use **Symbion** library to target Windows Forms applications only so there is no need to add additional libraries as we will not provide WPF support at the moment. We will need to reference the following assemblies in both **Symbion** and **SymBank.Banking** to get access to all Windows Forms resource types, controls and components in those assemblies.

Assemblies to reference: Symbion

```
System.Windows.Forms
System.Drawing
```

2.1 Commands

By default a GUI uses control events to trigger event handlers to run application code. This means that we need to directly create controls in the GUI, add event handlers in the GUI class and then write code for the event handlers. Everything is encapsulated within the GUI itself. However for a modular application, the code to execute may not be in the GUI application but in an external module that can be dynamically loaded. If the module is not loaded then the control used to trigger the code should also not be available as well.

We will now implement a **Command** class in **Symbion** library that provides enough information to support the dynamic generation of a control and also expose a way to execute code that can be anywhere inside or outside of the GUI application. Once the control is created for the command, we need to make sure updating of the command should also update the control. We use a **CommandChanged** event to inform a GUI that to update the command's associated control.

Implementing an executable command class: Symbion\Command.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Symbion {
    public class Command {
        public event EventHandler CommandChanged;
        public Image Icon { get; set; }
        public string Caption { get; set; } = string.Empty;
        public string Description { get; set; } = string.Empty;
        public bool Enabled { get; set; } = true;
        public bool Checked { get; set; };
        public Keys Keys { set; get; } = Keys.None;
        public Action<Command> Action { get; set; }
        public object Parameter { get; set; }

        public void Update() {
            CommandChanged?.Invoke(
                this, EventArgs.Empty);
        }

        public void Execute() {
            if (Enabled) Action?.Invoke(this);
        }
    }
}
```

We will now declare an **IActionSite** interface to be implemented by components that can accept command objects to generate controls that can then be used to execute the code attached to the command object. We also support updating and removal of the command.

Interface to support commands: IActionSite.cs

```
namespace Symbion {
    public interface IActionSite {
        void Add(Command command);
        void Update(Command command);
        void Remove(Command command);
        void AddSeparator();
    }
}
```

The areas where commands can be accepted is determined by the application. We will create and expose a dictionary of **IActionSites** that will support commands.

Exposing action sites from the main form: SymBank\ShellForm.cs

```
public partial class ShellForm : Form, IShell {
    private Dictionary<string, IActionSite> _sites;
    public ShellForm() {
        InitializeComponent();
        _sites = new Dictionary<string, IActionSite>();
    }
    public Dictionary<string, IActionSite> Sites { get { return _sites; }}
            :
}
```

However services and modules will not be able to access the dictionary unless we also expose it through our GUI shell service.

Allow access to action sites through IShell interface: Symbion\IShell.cs

```
namespace Symbion {
    public interface IShell : IService {
        Dictionary<string, IActionSite> Sites { get; }
        string Status { set; }
        void Success(string message);
        void Failure(string message);
        void Warning(string message);
        void Close();
    }
}
```

2.2 Implementing an Action Site

We can now implement one or more action site classes to support the different type of GUI elements. For example if we want to use commands to create the menu items in a menu control, we need to implement an action site for menu.

Implementing an action site for menu: Symbion\MenuActionSite.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Symbion {
    public class MenuActionSite : IActionSite {

    }
}
```

We will declare a **_menu** field to assign an actual **ToolStripMenuItem** representing the parent of the controls that we will dynamically add from information stored inside a command object. We will also declare a dictionary to associate a command with the control created from that command.

Fields added to store a parent menu and controls to be added to the menu

```
private ToolStripMenuItem _menu;
private Dictionary<Command, ToolStripMenuItem> _items;

public MenuActionSite(ToolStripMenuItem menu) {
    _items = new Dictionary<Command, ToolStripMenuItem>();
    _menu = menu;
}
```

Event handlers to execute a command and update a control from its command

```
private void OnItemClick(object sender, EventArgs e) {
    var item = (ToolStripMenuItem)sender; ((Command)item.Tag).Execute();
}

private void OnCommandChanged(object sender, EventArgs e) { Update((Command)sender); }
```

The add command method

```
public void Add(Command command) {
    command.CommandChanged += OnCommandChanged;
    ToolStripMenuItem item = new ToolStripMenuItem(
        command.Caption, command.Icon, OnItemClick, command.Keys);
    _items.Add(command, item);
    _menu.DropDownItems.Add(item);
    item.ToolTipText = command.Description;
    item.Enabled = command.Enabled;
    item.Checked = command.Checked;
    item.Tag = command;
}
```

The **Add** method will create a menu item control with data from the command object. If the command object is changed, we will need to update the menu item control as well. The control will be added to the menu that this site represents. The command is stored in the **Tag** property of the control so you can easily locate the command object from the control. The command is associated to the control using a dictionary. We will then be able to fetch the control from the dictionary based on the command. We can then update the control from data in the command object.

The update command method

```
public void Update(Command command) {
    ToolStripMenuItem item = _items[command];
    if (item != null) {
        item.Text = command.Caption;
        item.Image = command.Icon;
        item.ShortcutKeys = command.Keys;
        item.ToolTipText = command.Description;
        item.Enabled = command.Enabled;
        item.Checked = command.Checked;
    }
}
```

If for some reason you no longer want to use the command, call the **Remove** method that will remove the control associated with the command from a UI. To segregate a group of menu items from another group in the UI, can call **AddSeparator** to add a separator to a menu between items.

The remove methods

```
public void Remove(Command command) {
    ToolStripMenuItem item = _items[command];
    if (item != null) {
        command.CommandChanged -= OnCommandChanged;
        _menu.DropDownItems.Remove(item);
        _items.Remove(command);
    }
}
```

Add separator method

```
public void AddSeparator() {
    _menu.DropDownItems.Add( new ToolStripSeparator());
}
```

Using the Form Designer to add a **MenuStrip** control to the main form and change its name to **mbrMain**. In the **MenuStrip**, add two **ToolStripMenuItem**; one called as **mnuFile** and the other **mnuTools**. Then set the following control properties.

MenuStrip control properties: SymBank\ShellForm.cs

```
(Name)          : mbrMain
ShowItemToolTips: true
```

ToolStripMenuItem controls and properties

```
Name: mnuFile    Text: &File
Name: mnuTools   Text: Too&ls
```

Since there are two separate menus, you need to create two **MenuActionSite**, one for each menu. Add them to the **_sites** dictionary to allow access to the action sites exposed throught the **Sites** property from anywhere.

Registering MenuActionSite for each menu

```
public ShellForm() {
    InitializeComponent();
    _sites = new Dictionary<string, IActionSite>();
    _sites.Add("FileMenu", new MenuActionSite(mnuFile));
    _sites.Add("ToolsMenu", new MenuActionSite(mnuTools));
}
```

2.3 Implementing a Command

We will now demonstrate how to create a **Command** object and add to an action site. Before we write code, let us add a image resource to the **Resources** file. This image will be in for the command.

Image to add to the Resources file: SymBank\Properties\Resources.resx

```
Cancel  C:\CSC300-A\RSC\icons\dialog\cancel.png
```

We will add a method to process the command, which in this case will terminate the application. We can then retrieve the correct **IActionSite** and then attach a command that calls the method to terminate the application.

Method that terminates the application

```
public void onExitAction(Command comand) {
    Application.Exit();
}
```

Create and attach an inline command object

```
public ShellForm() {
            :
    var site = _sites["FileMenu"];
    site.Add(new Command { Caption = "E&xit", Icon = Resources.Cancel,
        Description = "Close SymBank application", Action = onExitAction,
        Keys = Keys.Alt | Keys.X });
}
```

If you create an inline command object but did not store the reference to this object, you will not be able to remove the object later. Following shows implementing a static method and creating a static command assigned to field or property that is accessible from anywhere in the application. You can easily update or remove the command.

Pre-create Command objects

```
private static void onExitAction(Command command) {
    Application.Exit();
}

public static Command ExitCommand = new Command {
    Caption = "E&xit", Icon = Resources.Cancel,
    Description = "Close SymBank application",
    Action = onExitAction, Keys = Keys.Alt | Keys.X
};
```

Registering an existing command

```
public ShellForm() {
            :
    var site = _sites["FileMenu"];
    site.Add(ExitCommand);
}
```

You can also create classes for commands and if the commands are reusable you can compile the classes into a separate shared assembly. Add an **ApplicationCommands** class into **Symbion** library. Add the *Cancel* image icon into **Symbion** resources file as well and remove it from **SymBank** application resources.

Implementing a shared Commands class: Symbion\ApplicationCommands.cs

```
namespace Symbion {
    public class ApplicationCommands {
        public static Command Exit = new Command {
            Caption = "E&xit",
            Icon = Resources.Cancel,
            Description = "Closes application",
            Action = ExitAction,
            Keys = Keys.Alt | Keys.X
        };

        private static void ExitAction(Command command) {
            Application.Exit();
        }
    }
}
```

Using a shared Command: SymBank\ShellForm.cs

```
public ShellForm() {
            :
    _sites["FileMenu"].Add(
        ApplicationCommands.Exit);
}
```

2.4 Asynchronous Commands

You can use the **Command** object to execute asynchronous code as well. Use **async** and **await** to run asynchronous methods from the action method.

Asynchronous commands

```
private static async void onExitAction(Command command) {
    await Task.Delay(2000);
    Application.Exit();
}
```

You do not need to implement a separate named method for the action as you use a lambda expression to contain the code to execute. Lambda expressions can also run asynchronously by using **async** and **await**.

Executing lambda expressions from Command

```
public static Command Exit = new Command {
    Caption = "E&xit",
    Icon = Resources.Cancel,
    Description = "Closes application",
    Action = command => Application.Exit(),
    Keys = Keys.Alt | Keys.X
};
```

Executing asynchronous lambda expressions from Command

```
public static Command Exit = new Command {
    Caption = "E&xit",
    Icon = Resources.Cancel,
    Description = "Closes application",
    Action = async command => {
        await Task.Delay(2000);
        Application.Exit();
    },
    Keys = Keys.Alt | Keys.X
};
```

2.5 ToolBar Action Site

The same command can also be assigned to multiple sites. Here we will implement a new action site that encapsulates a toolbar. A **ToolStripButton** will be created for each command contained within a **ToolStrip** control.

Implementing action site for toolbars: ToolbarActionSite.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Symbion {
    public class ToolbarActionSite : IActionSite {

    }
}
```

Fields for the toolbar and toolbar buttons

```
private Dictionary<Command, ToolStripButton> _items;
private ToolStrip _toolbar;

public ToolbarActionSite(ToolStrip toolbar) {
    _items = new Dictionary<Command, ToolStripButton>();
    _toolbar = toolbar;
}
```

Execute command when toolbar button is clicked

```
private void OnItemClick(object sender, EventArgs e) {
    var item = (ToolStripButton)sender;
    ((Command)item.Tag).Execute();
}
```

Update button when command is changed

```
private void OnCommandChanged(object sender, EventArgs e) {
    Update((Command)sender);
}
```

Adding a button for each command

```
public void Add(Command command) {
    command.CommandChanged += OnCommandChanged;
    ToolStripButton item = new ToolStripButton(
        string.Empty, command.Icon, OnItemClick);
    _items.Add(command, item);
    _toolbar.Items.Add(item);
    item.ToolTipText = command.Description;
    item.Enabled = command.Enabled;
    item.Checked = command.Checked;
    item.Tag = command;
}
```

Adding separators between button groups

```
public void AddSeparator() {
    _toolbar.Items.Add(new ToolStripSeparator());
}
```

Removing button associated with command

```
public void Remove(Command command) {
    ToolStripButton item = _items[command];
    if (item != null) {
        command.CommandChanged -= OnCommandChanged;
        _toolbar.Items.Remove(item);
        _items.Remove(command);
    }
}
```

Updating button from command

```
public void Update(Command command) {
    ToolStripButton item = _items[command];
    if (item != null) {
        item.Text = command.Caption;
        item.Image = command.Icon;
        item.ToolTipText = command.Description;
        item.Enabled = command.Enabled;
        item.Checked = command.Checked;
    }
}
```

Add a **ToolStrip** control into the main form and named it as **tbrMain**. In code, add a **ToolBarActionSite** for the **ToolStrip**. Then add **Exit** command also to the new site. Run the application to see the **Exit** command available from the menu as well as on a toolbar.

Register a new action site and command: SymBank\ShellForm.cs

```
public ShellForm() {
    InitializeComponent();
    _sites = new Dictionary<string, IActionSite>();
    _sites.Add("FileMenu", new MenuActionSite(mnuFile));
    _sites.Add("ToolsMenu", new MenuActionSite(mnuTools));
    _sites.Add("Toolbar", new ToolbarActionSite(tbrMain));
    _sites["FileMenu"].Add(ApplicationCommands.Exit);
    _sites["Toolbar"].Add(ApplicationCommands.Exit);
}
```