Programmer to Programmer ™

Dissecting A C# Application

Inside SharpDevelop

Christian Holm, Mike Kruger, Bernhard Spuida

Wrox technical support at: support@wrox.com
Updates and source code at: www.wrox.com
Peer discussion at: p2p.wrox.com

What You Need to Use This Book

The following is the recommended system requirements for compiling and running SharpDevelop:

- Windows 2000 Professional or later
- The .NET Framework SDK (Freely available from Microsoft)

In addition, this book assumes the following knowledge:

- Sound knowledge of .NET Framework fundamentals
- A good understanding of the C# Language

Summary of Contents

Introduction		**1**
Chapter 1:	Features at a Glance	**7**
Chapter 2:	Designing the Architecture	**23**
Chapter 3:	Implementing the Core	**51**
Chapter 4:	Building the Application with Add-ins	**81**
Chapter 5:	Providing Functionality with Workspace Services	**107**
Chapter 6:	The User Interface	**135**
Chapter 7:	Internationalization	**169**
Chapter 8:	Document Management	**189**
Chapter 9:	Syntax Highlighting	**219**
Chapter 10:	Search and Replace	**235**
Chapter 11:	Writing the Editor Control	**263**
Chapter 12:	Writing the Parser	**291**
Chapter 13:	Code Completion and Method Insight	**329**
Chapter 14:	Navigating Code with the Class Scout and the Assembly Scout	**369**
Chapter 15:	The Designer Infrastructure	**413**
Chapter 16:	mplementing a Windows Forms Designer	**437**
Chapter 17:	Code Generation	**465**
Index		**499**

16

Implementing a Windows Forms Designer

In this chapter, we will examine all the services that are necessary for implementing a Windows Forms designer. We will look at how they are implemented in SharpDevelop, and how they interact with each other and the designer framework to support common uses.

We will be looking at:

- The services that are required for making the basic forms designer work
- Some designer services of minor importance, which are implemented but not required for a basic forms designer; these provide some useful functionalities.

Designer Services

In this section, we will look at the basic services required for making the Forms designer work, which are:

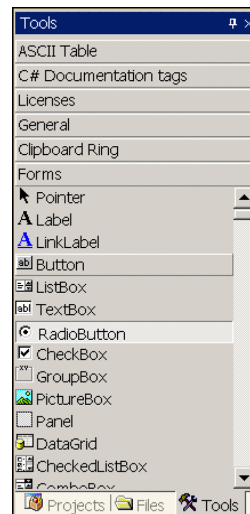
- **Toolbox Service**
  This wraps some toolbox functionality for the designer infrastructure
- **Menu Command Service**
  This service displays context menus and is used for calling commands, which affect the components in the design area
- **Selection Service**
  This handles the selection of components and keeps track of the selected components

Toolbox Service

The Toolbox service represents the Toolbox in the IDE. The Toolbox contains components that we can place onto our forms. The interface implemented by this service is the `System.Drawing.Design.IToolboxService` interface, which is a part of the .NET Framework.

The main purpose of the Toolbox service is to store `ToolBoxItem` objects. Toolbox items simply contain a type, bitmap, and text. The Design area uses the Toolbox service to create new components. When the Design area is clicked, the type of the component is stored in the `ToolBoxItem` object.

The following screenshot shows the SharpDevelop Toolbox:



In this Toolbox, the Forms category is active and it contains various components. These items represent the Toolbox items in the service. Toolbox items have their individual images, which are defined by the component implementer.

How does the designer know which Toolbox item should be used for creating a component?

The Toolbox service contains the functionality for selecting a component. After a Toolbox item is selected, clicking on the Design area will create a component of the Toolbox type in the Design area. The Design area does this automatically and there is not much code to understand.

The service is implemented in the `src\SharpDevelop\FormDesigner\FormDesigner\Services\ToolboxService.cs` file.

We will look at the interface to get an understanding of what the service does. We won't go through the full implementation. As the interface mainly wraps containers, the implementation won't provide us with any valuable insights. The interface describes just a container for Toolbox items:

```
public interface IToolboxService
{
  void AddToolboxItem(ToolboxItem toolboxItem, string category);
  void AddToolboxItem(ToolboxItem toolboxItem);
```

```
void AddLinkedToolboxItem(ToolboxItem toolboxItem, string category,
                          IDesignerHost host);
void AddLinkedToolboxItem(ToolboxItem toolboxItem, IDesignerHost host);

void RemoveToolboxItem(ToolboxItem toolboxItem, string category);
void RemoveToolboxItem(ToolboxItem toolboxItem);

ToolboxItemCollection GetToolboxItems(string category,
                                      IDesignerHost host);
ToolboxItemCollection GetToolboxItems(string category);
ToolboxItemCollection GetToolboxItems(IDesignerHost host);
ToolboxItemCollection GetToolboxItems();
```

A `category` is a group of Toolbox items. In the current implementation, there is only one global category, which contains all Toolbox items (it is represented under the Forms group inside the SharpDevelop Toolbox implementation).

The first part of the interface manages the Toolbox items. Toolbox items may be linked to a designer host. Currently this functionality is not used in SharpDevelop; when we use it the designer host will contain the selected item and there will only be one Global Selection service, which handles all designer areas; however, currently we have a Toolbox service for each designer host.

Now we will look at how to select a Toolbox item:

```
void SetSelectedToolboxItem(ToolboxItem toolboxItem);

ToolboxItem GetSelectedToolboxItem(IDesignerHost host);
ToolboxItem GetSelectedToolboxItem();

void SelectedToolboxItemUsed();
```

The Toolbox implementation of SharpDevelop calls `SetSelectedToolboxItem`, and the Designer area calls `SelectedToolboxItemUsed` when the item is used (when it's placed into the Design area). After that, we set the selected item to `null`, otherwise, the Toolbox item remains selected.

The service has functionality to serialize and deserialize Toolbox items:

```
object SerializeToolboxItem(ToolboxItem toolboxItem);

ToolboxItem DeserializeToolboxItem(object serializedObject,
                                   IDesignerHost host);
ToolboxItem DeserializeToolboxItem(object serializedObject);
```

Currently the implementation doesn't make use of serializing Toolbox items; therefore, it is unimplemented.

But how are the items serialized and deserialized? We can put (and remove) custom Toolbox item creators into the service with these four functions:

```
void AddCreator(ToolboxItemCreatorCallback creator, string format,
                IDesignerHost host);
void AddCreator(ToolboxItemCreatorCallback creator, string format);
void RemoveCreator(string format, IDesignerHost host);
void RemoveCreator(string format);
```

A `ToolboxItemCreatorCallback` delegate has the following signature:

```
public delegate ToolboxItem ToolboxItemCreatorCallback(
   object serializedObject,
   string format
);
```

How are items serialized using a custom method? The answer is that they won't be serialized by this service, but rather through a registered callback. A user-defined method, which is not part of this service, serializes them. But, the deserialization can be customized using the `ToolboxItemCreatorCallback`.

The SharpDevelop `IToolboxService` implementation does not implement the serialization functionality, as currently it is not needed. There are some more helper functions for the serialization parts (which are not implemented either):

```
bool IsToolboxItem(object serializedObject, IDesignerHost host);
bool IsToolboxItem(object serializedObject);

bool IsSupported(object serializedObject, ICollection filterAttributes);
bool IsSupported(object serializedObject, IDesignerHost host);
```

The `SetCursor` method changes the current application mouse cursor to a cursor that represents the selected toolbox item:

```
bool SetCursor();
```

It returns `true` when the cursor is set to a cursor representing the current selected Toolbox item. It returns `false` when no item is selected and the cursor is set to the standard cursor.

The `Refresh` method redraws the toolbox:

```
void Refresh();
```

In the current implementation, this method does nothing because the service only wraps the real SharpDevelop toolbar to this interface! Refreshing is done by a mechanism different from this service.

Finally properties that represent the Toolbox categories are listed:

```
  CategoryNameCollection CategoryNames {
    get;
  }

  string SelectedCategory {
    get;
    set;
  }
}
```

We have finished looking at the entire interface, but we still do not know how the Toolbox is filled with Toolbox items. For this very purpose there is a method defined in `src\SharpDevelop\FormDesigner\FormDesigner\FormDesignerDisplayBindingBase.cs`:

```
ArrayList BuildToolboxFromAssembly(Assembly assembly)
{
  ArrayList toolBoxItems = new ArrayList();

  Hashtable images = new Hashtable();

  // try to load resource icons
  foreach (string name in assembly.GetManifestResourceNames()) {
    try {
      Bitmap bitmap = newBitmap(Image.
        FromStream(assembly.GetManifestResourceStream(name)));
      images[name] = bitmap;
    } catch {}
  }

  foreach (Module module in assembly.GetModules(false)) {
    foreach (Type type in module.GetTypes()) {
      if (type.IsDefined(typeof(ToolboxItemFilterAttribute), true)) {
        object[] attributes = type.
          GetCustomAttributes(typeof(ToolboxItemFilterAttribute), true);

        // here we should enumerate all attributes ...
        ToolboxItemFilterAttribute attr =
          (ToolboxItemFilterAttribute) attributes[0];

        // get only the components
        //that require 'Windows Forms' toolbox filter
        if (attr.FilterType == ToolboxItemFilterType.Allow &&
          attr.FilterString == "System.Windows.Forms") {
          if (images[type.FullName + ".bmp"] != null) {
            ToolboxItem item = new ToolboxItem(type);
            // filter out UserControl manually
            if (item.DisplayName != "UserControl") {
              toolBoxItems.Add(item);
  } } } } } } //closing braces collapsed for brevity

  toolBoxItems.Sort(new ToolboxItemSorter());
  return toolBoxItems;
}
```

This method iterates through all types in the assembly and inserts all types having their `ToolboxItemFilterAttribute` set to `System.Windows.Forms`, and an icon defined in the assembly resources.

With the toolbox filled, we will move on to context menus, which are an important feature in a modern-day application.

Menu Command Service

The Menu Command service is used to display all context menus in the Design area. When the user right-clicks in the Design area (which is defined in the .NET Framework), a method from the Menu Command service is called. Therefore, this service must be implemented to make all context menus work.

Besides displaying menus, the Menu Command service also acts as a command execution service. The Designer Framework puts `MenuCommands` into the service. A client can execute the commands using a `CommandID` object that identifies a specific command, which is used for making the menu commands in the IDE. The menu commands simply call the Menu Command service.

There are many commands defined and implemented in the framework infrastructure. We will look at them at the end of this section.

The service is defined in the `src\SharpDevelop\FormDesigner\FormDesigner\Services\MenuCommandService.cs` file:

```
class MenuCommandService : IMenuCommandService
{
  IDesignerHost host;
  ArrayList     commands = new ArrayList();
  ArrayList     verbs    = new ArrayList();

  public DesignerVerbCollection Verbs {
    get {
      DesignerVerbCollection verbCollection =
        CreateDesignerVerbCollection();
      verbCollection.AddRange((DesignerVerb[])verbs.
        ToArray(typeof(DesignerVerb)));
      return verbCollection;
    }
  }

  public MenuCommandService(IDesignerHost host)
  {
    this.host = host;
  }

  public void AddCommand(MenuCommand command)
  {

    Debug.Assert(command != null);
    Debug.Assert(command.CommandID != null);
    Debug.Assert(!commands.Contains(command));
    this.commands.Add(command);
  }

  public void AddVerb(DesignerVerb verb)
  {
    Debug.Assert(verb != null);
    this.verbs.Add(verb);
  }
```

The Menu Command service contains commands and verbs. The commands are provided by the framework. verbs are like commands but they are user-defined. Verbs can be added by anyone but usually the designers of the components define verbs for their components.

The `CreateDesignerVerbCollection` method creates a verb collection that contains all verbs defined by the designers that are attached to the selected components (for more information about designers, refer to Chapter 15). It gets the selected component, and uses the verb collection of its attached designer to create a verb collection that is specific for this component.

If the selected component is the root component, the default verbs that are stored in the menu command are attached to the returned collection also. One framework component that has verbs attached to is the Tab control, which allows us to add or remove Tab pages in the context menu.

```
public DesignerVerbCollection CreateDesignerVerbCollection()
{
  DesignerVerbCollection designerVerbCollection = new
    DesignerVerbCollection();

  ISelectionService selectionService = (ISelectionService)
    host.GetService(typeof(ISelectionService));

  if (selectionService != null && selectionService.SelectionCount == 1) {
    IComponent selectedComponent =
      selectionService.PrimarySelection as Component;
    if (selectedComponent != null) {
      IDesigner designer = host.GetDesigner
        ((IComponent)selectedComponent);
      if (designer != null) {
        designerVerbCollection.AddRange(designer.Verbs);
      }
    }

    if (selectedComponent == host.RootComponent) {
      designerVerbCollection.AddRange(this.verbs);
    }
  }
  return designerVerbCollection;
}
```

Also the service allows commands and verbs to be removed:

```
public void RemoveCommand(MenuCommand command)
{
  Debug.Assert(command != null);
  commands.Remove(command.CommandID);

}

public void RemoveVerb(DesignerVerb verb)
{
  Debug.Assert(verb != null);
  verbs.Remove(verb);

}
```

Now we will look at the methods used to invoke and find a command:

```
public bool GlobalInvoke(CommandID commandID)
{
  MenuCommand menuCommand = FindCommand(commandID);
  if (menuCommand == null) {
    MessageBox.Show("Can't find command " + commandID, "Error");
    return false;
  }

  menuCommand.Invoke();
  return true;
}

public MenuCommand FindCommand(CommandID commandID)
{
  foreach (MenuCommand menuCommand in commands) {
    if (menuCommand.CommandID == commandID) {
      return menuCommand;
    }
  }

  foreach (DesignerVerb verb in Verbs) {
    if (verb.CommandID == commandID) {
      return verb;
    }
  }
  return null;
}
```

Before any kind of invocation can happen, we have to display the menu that contains the commands. The `ShowContextMenu` method takes the menu `command` and `verbs` and constructs a context menu using the Magic library, and displays it. It has a `CommandID`, which specifies the menu ID to be displayed. The menu ID is different, according to what the user had clicked (for example, the Component tray has a different menu ID as compared to a button in the root component). The menu ID is used for displaying context-sensitive menus. It gets the screen position, which determines where to show the context menu:

```
public void ShowContextMenu(CommandID menuID, int x, int y)
{
  Crownwood.Magic.Menus.PopupMenu popup =
    new Crownwood.Magic.Menus.PopupMenu();

  PropertyService propertyService =
    (PropertyService) ICSharpCode.Core.Services.ServiceManager.Services.
                      GetService(typeof(PropertyService));

  popup.Style =
    (Crownwood.Magic.Common.VisualStyle) propertyService.
    GetProperty("ICSharpCode.SharpDevelop.Gui.VisualStyle",
    Crownwood.Magic.Common.VisualStyle.IDE);
  ArrayList menuCommands = new ArrayList();
```

```
      foreach (MenuCommand command in commands) {
        if (command.Visible && command.Enabled) {
          if (ContextMenuCommand.GetStandardCommandName(command.CommandID) !=
            null) {
            menuCommands.Add(new ContextMenuCommand(command));
          }
        }
      }

      foreach (DesignerVerb verb in Verbs) {
        if (verb.CommandID.Guid  == menuID.Guid ) {
          menuCommands.Add(new ContextMenuCommand(verb));
        }
      }

      popup.MenuCommands.AddRange((Crownwood.Magic.Menus.MenuCommand[])
        menuCommands.ToArray(typeof(Crownwood.Magic.Menus.MenuCommand)));
      popup.TrackPopup(new Point(x, y));
    }
```

The `ContextMenuCommand` class is a wrapper class for the `Crownwood.Magic.Menus.MenuCommand` class. We don't look at it here because the implementation of this class will not bring us any new insights. It just wraps a name (which is displayed) to the menu command using the field name from the `System.ComponentModel.Design.StandardCommands` class.

The `StandardCommands` class contains public static `CommandID` fields and a method that converts the `CommandID` to a screen name. This will be replaced in future versions with SharpDevelop standard menus, which allow localization and real customization of these menus.

The commands defined by the standard commands class are:

```
AlignBottom, AlignHorizontalCenters, AlignLeft, AlignRight,
AlignToGrid, AlignTop, AlignVerticalCenters,

ArrangeBottom, ArrangeIcons, ArrangeRight, LineupIcons,

BringForward, BringToFront, SendBackward, SendToBack,

CenterHorizontally, CenterVertically,
SizeToControl, SizeToControlHeight, SizeToControlWidth, SizeToFit,
SizeToGrid, SnapToGrid,

Copy, Cut, Delete, Paste, Replace, SelectAll,

F1Help, Properties, PropertiesWindow, TabOrder, VerbFirst, VerbLast,

Group, Ungroup,

HorizSpaceConcatenate, HorizSpaceDecrease, HorizSpaceIncrease,
HorizSpaceMakeEqual,
VertSpaceConcatenate, VertSpaceDecrease, VertSpaceIncrease, VertSpaceMakeEqual,
```

```
MultiLevelRedo, MultiLevelUndo, Redo, Undo,

LockControls, ShowGrid, ShowLargeIcons, ViewGrid
```

Additional commands are defined in the `System.Windows.Forms.Design.MenuCommands` class:

```
DesignerProperties,

KeyCancel, KeyDefaultAction,
KeyMoveDown, KeyMoveLeft, KeyMoveRight, KeyMoveUp,
KeyNudgeDown,  KeyNudgeLeft, KeyNudgeRight, KeyNudgeUp,
KeyNudgeHeightDecrease, KeyNudgeHeightIncrease, KeyNudgeWidthDecrease,
KeyNudgeWidthIncrease,

KeyReverseCancel, KeySelectNext, KeySelectPrevious,
KeySizeHeightDecrease, KeySizeHeightIncrease,
KeySizeWidthDecrease, KeySizeWidthIncrease,
KeyTabOrderSelect
```

The class is called `MenuCommands` but most commands shown are key actions (they start with the prefix `Key`). So why is this class called `MenuCommands` and not `KeyboardCommands`?

The `MenuCommands` class contains IDs for menus. Remember the `ShowContextMenu` method from the Menu Command service has a `CommandID` as a parameter. The correct menu to display can be determined by comparing the given ID with the IDs from the `MenuCommands` class.
The `MenuCommands` class defines the following menus by `CommandID`:

```
ComponentTrayMenu, ContainerMenu, SelectionMenu, TraySelectionMenu,
```

Armed with this knowledge about available commands, let's look at how to execute those stock commands.

Executing the Standard Commands

Executing the standard commands is really simple. All commands that are used inside SharpDevelop are defined inside the
`src\SharpDevelop\FormDesigner\Commands\FormCommands.cs` file.

The commands extend this class:

```
public abstract class AbstractFormDesignerCommand : AbstractMenuCommand
{
  public abstract CommandID CommandID {
    get;
  }

  public override void Run()
  {
    IWorkbenchWindow window =
      WorkbenchSingleton.Workbench.ActiveWorkbenchWindow;
    if (window == null) {
```

```
        return;
      }

      FormDesignerDisplayBindingBase formDesigner =
        window.ViewContent as FormDesignerDisplayBindingBase;

      if (formDesigner != null) {
        IMenuCommandService menuCommandService =
          (IMenuCommandService)formDesigner.DesignerHost.
          GetService(typeof(IMenuCommandService));
        menuCommandService.GlobalInvoke(CommandID);
      }
  } }
```

The `AbstractFormDesignerCommand` just gets the Menu Command service and executes the `CommandID`, which must be given by a subclass. For example, the bring-to-front command looks like:

```
public class BringToFront : AbstractFormDesignerCommand
{
  public override CommandID CommandID {
    get {
      return StandardCommands.BringToFront;
    }
  }
}
```

Many standard commands are used inside the Format menu of SharpDevelop to execute specific tasks.

Now we will look at something a bit more difficult – the execution of keyboard commands.

Implementing a Key Event Handler

The problem with the key handler was that no existing service could be found that can take a key and perform the appropriate action. Instead, a class was implemented that extended the `System.Windows.Forms.IMessageFilter` interface. `IMessageFilter` objects can be used in Windows Forms to filter a specific window event.

This interface declares only one method:

```
bool PreFilterMessage(ref Message m);
```

If this method returns `true`, the message will be filtered out and not passed any further in the event filter chain.

An object of the `IMessageFilter` class must be put into the message filter queue. This is done with a call to the `System.Windows.Forms.Application.AddMessageFilter` method.

The class that is used for the key events, in the forms designer, is implemented in `src\SharpDevelop\FormDesigner\FormDesigner\FormKeyHandler.cs`:

```
public class FormKeyHandler : IMessageFilter
{
  const int keyPressedMessage = 0x100; // WM_KEYDOWN

  Hashtable keyTable = new Hashtable();
```

The `keyPressedMessage` is a magic number, which is rooted in the native Windows message system. It just says that the message, which is sent, is of type 'key pressed'.

A hash table is used for providing a fast, clean, and easy way to map a key to a command. The table is initialized in the constructor:

```
  public FormKeyHandler ()
  {
    // normal keys
    keyTable[Keys.Left]  = new CommandWrapper(MenuCommands.KeyMoveLeft);
//code omitted for listing

    // shift modified keys
    keyTable[Keys.Left | Keys.Shift]  =
      new CommandWrapper(MenuCommands.KeySizeWidthDecrease);
//code omitted for listing

    // ctrl modified keys
    keyTable[Keys.Left | Keys.Control]  =
      new CommandWrapper(MenuCommands.KeyNudgeLeft);
//code omitted for listing

    // ctrl + shift modified keys
    keyTable[Keys.Left | Keys.Control | Keys.Shift]  =
      new CommandWrapper(MenuCommands.KeyNudgeWidthDecrease);
//code omitted for listing
  }
```

Now we will look at the filter method, which filters the keyboard events. This method gets all Windows messages but we only care about the keyboard pressed events; therefore, we use the `keyPressedMessage` value that we defined earlier (in this section) to check whether the message refers to a key being pressed.

```
  public bool PreFilterMessage(ref Message m)
  {
    if (m.Msg != keyPressedMessage) {
      return false;
    }

    if (WorkbenchSingleton.Workbench.ActiveWorkbenchWindow == null) {
      return false;
    }

    FormDesignerDisplayBindingBase formDesigner =
      WorkbenchSingleton.Workbench.ActiveWorkbenchWindow.ViewContent as
```

```
      FormDesignerDisplayBindingBase;
    if (formDesigner == null || !formDesigner.IsFormDesignerVisible) {
      return false;
    }

    Keys keyPressed = (Keys)m.WParam.ToInt32() | Control.ModifierKeys;
    CommandWrapper commandWrapper = (CommandWrapper)keyTable[keyPressed];

    if (commandWrapper != null) {
      IMenuCommandService menuCommandService =
        (IMenuCommandService)formDesigner.DesignerHost.
        GetService(typeof(IMenuCommandService));
      ISelectionService   selectionService =
        (ISelectionService)formDesigner.DesignerHost.
        GetService(typeof(ISelectionService));
      ICollection components = selectionService.GetSelectedComponents();
      menuCommandService.GlobalInvoke(commandWrapper.CommandID);

      if (commandWrapper.RestoreSelection) {
        selectionService.SetSelectedComponents(components);
      }
      return true;
    }
    return false;
  }
```

Some commands remove the selection when they are executed when, in fact, they should not, for example, some resizing commands (remember that we haven't implemented these commands, as we can't access their source code). For this reason, the selection from the selection manager is saved before the command is called, and restored after it is executed. This is done for all commands, except the commands that alter the selection (for example, when the Tab key selects the next component).

A `CommandWrapper` class is used instead of putting the `CommandID` object directly into the hash table. The `CommandWrapper` provides a `RestoreSelection` field; if this is set to `false` it won't restore the selection afterwards. The `CommandWrapper` is an internal class.

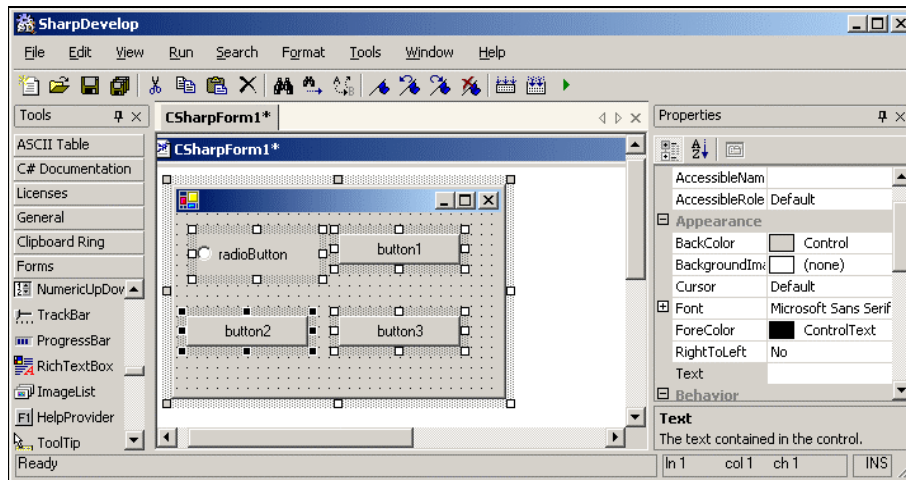
```
  class CommandWrapper
  {
    CommandID commandID;
    bool      restoreSelection;

    public CommandID CommandID {
      get {
        return commandID;
      }
    }

    public bool RestoreSelection {
      get {
        return restoreSelection;
      }
    }
```

```
        public CommandWrapper(CommandID commandID) : this(commandID, true)
        {
        }
        public CommandWrapper(CommandID commandID, bool restoreSelection)
        {
          this.commandID        = commandID;
          this.restoreSelection = restoreSelection;
        }
      }
```

So far we have seen a bit of the Selection service usage; now we will look at it in more detail.

Selection Service

The Selection service holds all selected components. The following screenshot shows what selections can look like:



Note that the button2 component is surrounded by small black squares, whereas for the other selected components they are white. This indicates that button2 is a special component, or the primary selection. The Selection service must have a way to handle the primary selection, as well as the other selected components.

In the Properties window, the common properties are displayed. Common properties are those properties that all selected components share. For example, you can change the font of all the selected components throught the Properties window.

How is this implemented? We can find the code in `src\SharpDevelop\FormDesigner\FormDesigner\Services\SelectionService.cs`:

```
    public class SelectionService : ISelectionService
    {
      IDesignerHost host;
      ArrayList selectedComponents = new ArrayList();
```

The Selection service stores all selected components in an `ArrayList`. The primary selection is the first element in this list:

```
public object PrimarySelection {
  get {
    if (selectedComponents.Count > 0) {
      return selectedComponents[0];
    }
    return null;
  }
}
public int SelectionCount {
  get {
    return selectedComponents.Count;
  }
}

public SelectionService(IDesignerHost host)
{
  Debug.Assert(host != null);
  this.host = host;
  ((IComponentChangeService)host.
    GetService(typeof(IComponentChangeService))).ComponentRemoved +=
      new ComponentEventHandler(ComponentRemovedHandler);
}
```

The next listing shows the `get` methods for the components:

```
public bool GetComponentSelected(object component)
{
  return selectedComponents.Contains(component);
}

public ICollection GetSelectedComponents()
{
  return selectedComponents.ToArray();
}
```

The `set` method is special because it needs to take care of modifier key states. The expected behavior is:

- **Shift pressed** – When more than one component is passed to the method, they should replace the selection (this is done by other designers; therefore, this behavior is standard). When one component is passed, it should be added to the selection as a new primary selection. If the component is already selected it should be removed from the selection.
- **Control pressed** – If one component is passed to the method, the control key behaves like the shift key. Otherwise the components should be added to the current selection and the new root component is one of the added components.
- **No Modifier key** – If a selected component is passed to the method, the selected component should become the new root component. Otherwise, the new component(s) should replace the current selection.

Now, let's look at the method that handles this behavior:

```
    public void SetSelectedComponents(ICollection components,
                                      SelectionTypes selectionType)
    {
      OnSelectionChanging(EventArgs.Empty);
      if (components == null || components.Count == 0) {
        selectedComponents.Clear();
        FireSelectionChange();
        return;
      }

      bool controlPressed = (Control.ModifierKeys & Keys.Control) ==
        Keys.Control;
      bool shiftPressed   = (Control.ModifierKeys & Keys.Shift)   ==
        Keys.Shift;
      switch (selectionType) {
        case SelectionTypes.Replace:
          ReplaceSelection(components);
          break;
        default:
          if (components.Count == 1 && (controlPressed || shiftPressed)) {
            ToggleSelection(components);
          } else if (controlPressed) {
            AddSelection(components);
          } else if (shiftPressed) {
            ReplaceSelection(components);
          } else {
            NormalSelection(components);
          }
          break;
      }
      selectedComponents.TrimToSize();
      FireSelectionChange();
    }

    public void SetSelectedComponents(ICollection components)
    {
      SetSelectedComponents(components, SelectionTypes.MouseDown);
    }

  #region SetSelection helper methods
    void ToggleSelection(ICollection components)
    {
      foreach (object component in components) {
        if (GetComponentSelected(component)) {
          selectedComponents.Remove(component);
        } else {
          selectedComponents.Insert(0, component);
        }
      }
    }
```

```
    void AddSelection(ICollection components)
    {
      foreach (object component in components) {
        if (!GetComponentSelected(component)) {
          selectedComponents.Insert(0, component);
        }
      }
    }

    void ReplaceSelection(ICollection components)
    {
      selectedComponents.Clear();
      AddSelection(components);
    }

    void NormalSelection(ICollection components)
    {
      if (components.Count == 1) {
        // just getting the first == last element in the components
        object componentToAdd = null;
        foreach (object component in components) {
          componentToAdd = component;
        }

        if (GetComponentSelected(componentToAdd)) {
          selectedComponents.Remove(componentToAdd);
          selectedComponents.Insert(0, componentToAdd);
          return;
        }
      }
      ReplaceSelection(components);
    }
  #endregion
```

As always, there might be someone interested in knowing when a selection changes. Therefore, we have some events that inform about selection change:

```
  #region Event methods
    void ComponentRemovedHandler(object sender, ComponentEventArgs e)
    {
      if (selectedComponents.Contains(e.Component)) {
        OnSelectionChanging(EventArgs.Empty);
        selectedComponents.Remove(e.Component);
        if (selectedComponents.Count == 0) {
          selectedComponents.Add(host.RootComponent);
        }
        FireSelectionChange();
      }
    }

    void FireSelectionChange()
    {
      OnSelectionChanged(EventArgs.Empty);
```

```
      ((DesignerEventService)host.RootComponent.Site.
        GetService(typeof(IDesignerEventService))).FileSelectionChanged();
    }

    protected virtual void OnSelectionChanging(EventArgs e)
    {
      if (SelectionChanging != null) {
        SelectionChanging(this, e);
      }
    }

    protected virtual void OnSelectionChanged(EventArgs e)
    {
      if (SelectionChanged != null) {
        SelectionChanged(this, e);
      }
    }
  #endregion
    public event EventHandler SelectionChanging;
    public event EventHandler SelectionChanged;
  }
```

This completes the entire Selection service.

There is one important service left – the `IDesignerSerializationService`. This service is used for serializing components. Without it, the Cut, Copy, and Paste commands won't work correctly. It is discussed in the next chapter as it builds on the XML persistence format, which we will be studying in that chapter.

The services we have looked at so far are the bare minimum requirement, which must be implemented to make the .NET Forms Designer work. Now we will look at some of the services that help us with the Forms Designer.

Additional Important Services

In this section, we will look at some services that are less important but nonetheless worth examing:

- **Designer Option Service**
  This service is used for grid options
- **Dictionary Service**
  This wraps a `HashTable` as a service
- **UI Service**
  This is used to display message boxes and has some UI properties
- **Type Descriptor Filter Service**
  This filters attributes, events, and properties

Designer Option Service

The Designer Option service is able to handle specified designer options. Currently, it holds only three options: `GridSize`, `ShowGrid`, and `SnapToGrid`.

The service is defined under
`src\SharpDevelop\FormDesigner\FormDesigner\Services\DesignerOptionService.cs`:

```
  public class DesignerOptionService : IDesignerOptionService
  {
    public const string GridSize   = "GridSize";
    public const string ShowGrid   = "ShowGrid";
    public const string SnapToGrid = "SnapToGrid";

    const string GridSizeWidth  = "GridSize.Width";
    const string GridSizeHeight = "GridSize.Height";
```

These constants are the option names, which are pre-defined by the framework infrastructure. Note that the last two values are not public, as the service is referenced through the interface and the interface doesn't have these values.

These options are stored into pages. Each page has its own set of options:

```
    public const string FormsDesignerPageName =
      "SharpDevelop Forms Designer\\General";

    Hashtable pageOptionTable  = new Hashtable();

    public DesignerOptionService()
    {
      Hashtable defaultTable = new Hashtable();

      defaultTable[GridSize]   = new Size(8, 8);
      defaultTable[ShowGrid]   = false;
      defaultTable[SnapToGrid] = false;

      pageOptionTable[FormsDesignerPageName] = defaultTable;
    }

    public object GetOptionValue(string pageName, string valueName)
    {
      Hashtable pageTable = (Hashtable)pageOptionTable[pageName];

      if (pageTable == null) {
        return null;
      }

      switch (valueName) {
        case GridSizeWidth:
          return ((Size)pageTable[GridSize]).Width;
        case GridSizeHeight:
          return ((Size)pageTable[GridSize]).Height;
        default:
```

```
          return pageTable[valueName];
        }
      }

      public void SetOptionValue(string pageName, string valueName, object val)
      {
        Hashtable pageTable = (Hashtable)pageOptionTable[pageName];

        if (pageTable == null) {
          pageOptionTable[pageName] = pageTable = new Hashtable();
        }

        switch (valueName) {
          case GridSizeWidth:
            Size size      = ((Size)pageTable[GridSize]);
            size.Width     = (int)val;
            pageTable[GridSize] = size;
            break;
          case GridSizeHeight:
            size           = ((Size)pageTable[GridSize]);
            size.Height    = (int)val;
            pageTable[GridSize] = size;
            break;
          default:
            pageTable[valueName] = val;
            break;
        }
      }
```

One thing that we can learn from this service is how to handle structures. Instead of getting the size, setting the `Width` size value, and writing the size back in the hash table, we could have done:

```
size = ((Size)pageTable[GridSize]);
size.Width = (int)val;
pageTable[GridSize] = size;
```

You may ask, why can't we simply use:

```
((Size)pageTable[GridSize]).Width = (int)val;
```

The reason is simple but can easily be forgotten. `Size` is not a class; it is a structure. This means that when it is unboxed with `(Size)pageTable[GridSize]`, a copy is given back, and if the `Width` property is updated in the copy, the original structure is not affected.

Therefore, we create a copy, alter the property in the copy, and then write the copy back; thereby, overwriting the original structure. Unfortunately, just from viewing the source code we can't tell if we are handling structures or classes. We can only say this if we know about the type that we are using.

Dictionary Service

The Dictionary service just wraps a hash table data structure as a service. It is the smallest (and simplest) of all services. Currently, we aren't sure where it gets used inside the Designer Framework but we have implemented it, as it is defined and easy to implement according to the framework specification.

The service is implemented under `src\SharpDevelop\FormDesigner\FormDesigner\Services\DictionaryService.cs`:

```
public class DictionaryService : IDictionaryService
{
  Hashtable table = new Hashtable();

  public object GetKey(object val)
  {

    foreach (DictionaryEntry entry in table) {
      if (entry.Value == val) {
        return entry.Key;
      }
    }
    return null;
  }

  public object GetValue(object key)
  {
    return key == null ? null : table[key];
  }

  public void SetValue(object key, object val)
  {
    if (key != null) {
      table[key] = val;
    }
} }
```

Now we will move on to the UI service.

UI Service

The UI service has several purposes:

- It displays messages (error messages and standard messages).
- It gives access to the Dialog Owner window. This window is used to set the owner property correctly for dialogs that pop up.
- It defines a dictionary that stores UI styles. Currently two styles are defined: `HighlightColor` and `DialogFont`. The `HighlightColor` is used as background color of the Component tray and the `DialogFont` property specifies the font that is used in the tray for displaying the component name.

The UI Service is implemented under `src\SharpDevelop\FormDesigner\FormDesigner\Services\UIService.cs`:

```
public class UIService : IUIService
{
  IDictionary styles = new Hashtable();

  public IDictionary Styles {
    get {
      return styles;
    }
  }

  public UIService()
  {
    ResourceService resourceService = (ResourceService)
      ServiceManager.Services.GetService(typeof(ResourceService));
    styles["DialogFont"]     = resourceService.LoadFont("Tahoma", 10);
    styles["HighlightColor"] = Color.White;
  }

  public void SetUIDirty()
  {
    // TODO: Fixme!
  }
```

The `SetUIDirty` method works a bit like the `Redraw` methods, inside the SharpDevelop abstract GUI system. The `SetUIDirty` method gets called when a component is selected, and then the implementation may choose to alter the enable state of some menu commands. However, currently, SharpDevelop has no use for this method.

The UI service is able to show a component editor but this functionality is currently not implemented:

```
  public bool CanShowComponentEditor(object component)
  {
    return false;
  }

  public bool ShowComponentEditor(object component, IWin32Window parent)
  {
    throw new System.NotImplementedException
      ("Cannot display component editor for " + component);
  }
```

Now we will look at the method that gets the main window, and the method that shows a dialog:

```
  public IWin32Window GetDialogOwnerWindow()
  {
    return (IWin32Window)WorkbenchSingleton.Workbench;
  }

  public DialogResult ShowDialog(Form form)
```

```
    {
      return form.ShowDialog(GetDialogOwnerWindow());
    }
```

The following methods are used for displaying error messages:

```
    public void ShowError(Exception ex)
    {
      ShowError(ex, null);
    }

    public void ShowError(string message)
    {
      ShowError(null, message);
    }

    public void ShowError(Exception ex, string message)
    {
      string msg = String.Empty;

      if (ex != null) {
        msg = "Exception occurred: " + ex.ToString() + "\n";
      }

      if (message != null) {
        msg += message;
      }
      ResourceService resourceService = (ResourceService)
        ServiceManager.Services.GetService(typeof(ResourceService));

      MessageBox.Show(GetDialogOwnerWindow(), msg,
                      resourceService.GetString("Global.ErrorText"),
                      MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
```

The following three methods show standard messages:

```
    public void ShowMessage(string message)
    {
      ShowMessage(message, "", MessageBoxButtons.OK);
    }

    public void ShowMessage(string message, string caption)
    {
      ShowMessage(message, caption, MessageBoxButtons.OK);
    }

    public DialogResult ShowMessage(string message, string caption,
                                    MessageBoxButtons buttons)
    {
      return MessageBox.Show(GetDialogOwnerWindow(), message, caption,
                             buttons);
    }
```

The ShowToolWindow function is currently not implemented. It gets a Guid and displays the specified Tool window. The Tool window's Guid values are defined in the System.ComponentModel.Design.StandardToolWindows class.

Tool windows are the pads inside SharpDevelop (for details, refer to the *Pads* section in Chapter 6), and this function may be added later during the development process.

However, the designer infrastructure doesn't use this function and the show pad functionality is implemented differently in SharpDevelop. Therefore, it doesn't really make sense to implement all functions in the services when they are not used by the designer infrastructure.

```
    public bool ShowToolWindow(Guid toolWindow)
    {
      return false;
    }
  }
```

Now we will move on to the Type Descriptor Filter service.

Type Descriptor Filter Service

The Type Descriptor Filter service is used for filtering attributes, events, and properties of components. The designer of a component may implement the IDesignerFilter interface, which allows the designer implementation to filter unused or unnecessary properties and events.

The service just passes the request it receives over to the IDesignerFilter object. The service is defined under src\SharpDevelop\FormDesigner\FormDesigner\Services\TypeDescriptorFilterService.cs:

```
  public class TypeDescriptorFilterService : ITypeDescriptorFilterService
  {
```

The GetDesignerFilter method gets the designer of the component using the Site property. It gets the designer host, which returns the designer of a component. The designer is cast to IDesignerFilter and if the designer implements this interface, the object is returned. If it does not implement this interface, null is returned. Therefore, it is up to the designer implementer to make sense of filtering attributes, events, or properties.

This method is used internally:

```
    IDesignerFilter GetDesignerFilter(IComponent component)
    {
      ISite site = component.Site;

      if (site == null) {
        return null;
      }

      IDesignerHost host = site.GetService(typeof(IDesignerHost)) as
        IDesignerHost;
```

```
        if (host == null) {
          return null;
        }

        IDesigner designer = host.GetDesigner(component);

        return designer as IDesignerFilter;
      }
```

Now we will look at the filter methods, which just delegate their calls over to the `IDesignerFilter` object.

We aren't sure this is the right way of implementing it but it seems to make sense, as the designer filter has two methods of filtering attributes, events, and properties. These methods are `PreFilterXYZ` and `PostFilterXYZ` (where `XYZ` stands for attributes, events, or properties). We just call these methods to get the filtering done:

```
      public bool FilterAttributes(IComponent component, IDictionary attributes)
      {
        IDesignerFilter designerFilter = GetDesignerFilter(component);
        if (designerFilter != null) {
          designerFilter.PreFilterAttributes(attributes);
          designerFilter.PostFilterAttributes(attributes);
        }
        return false;
      }

      public bool FilterEvents(IComponent component, IDictionary events)
      {
        IDesignerFilter designerFilter = GetDesignerFilter(component);
        if (designerFilter != null) {
          designerFilter.PreFilterEvents(events);
          designerFilter.PostFilterEvents(events);
        }
        return false;
      }

      public bool FilterProperties(IComponent component, IDictionary properties)
      {
        IDesignerFilter designerFilter = GetDesignerFilter(component);
        if (designerFilter != null) {
          designerFilter.PreFilterProperties(properties);
          designerFilter.PostFilterProperties(properties);
        }
        return false;
      }
    }
```

Now let's look at the unimplemented services.

Unimplemented Services

So far, we have looked at the services that are implemented, and have discussed at least the basic functions that make them work. Some of these services are more important than others, but there are still some services left that have only skeleton implementations, which do nothing.

These services are not required to make a forms designer work but they will be implemented in the future. Many of the services that we examined in this chapter began their life as skeleton implementations that grew useful over time. The skeleton implementations that are currently present are:

- `IExtenderProviderService`
  This service adds extender providers, which extend the properties shown in the property grid of a component.
- `IEventBindingService`
  This service exposes events as property objects, and is used to show the events of a method in a property grid (to make the events 'designable').
- `ITypeResolutionService`
  This service should be used to load types at design time. It provides `GetAssembly/GetType` methods. Currently all types inside the SharpDevelop designer are loaded from the standard assemblies. This service will become important when custom controls are supported.

Summary

In this chapter, we have looked at all of the services required to make the Forms Designer work.

We have examined how to handle the Toolbox and notifying that a component is selected for insertion in the Design area. We now know how Toolbox items are constructed and we have looked at the service that represents the Toolbox.

We have gone through how the context menu is displayed in the Design area and how the standard commands work. We examined how to call a pre-defined command and handle keyboard actions inside the Design area.

We looked at the Selection service, which handles the selected components. We discussed how the modifier keys change the selection behavior and we looked at how the primary selection is implemented.

In the next chapter, we will learn about making components persistent, by using XML. We will look at how code is generated and learn how to implement roundtripping.