OFFICE 2007 RIBBON ACTIVE PROJECT

Student Name	Student ID
Chun Kit Liu	9680381
Felix Soumpholphakdy	5987393
Jignesh Patel	2597284
Paterson Deshommes	6317669
Shahrad Rezaei	6286968

Summary of Project

The Microsoft Office 2007 Ribbon Bar is an open source control that allows .NET developers to add an intuitive Office look and feel to their C♯ or Visual Basic applications. The control has been designed to be user friendly for developers implementing the ribbon into their applications and for end users using the application. The latter is done by providing a familiar experience through mimicking Word, PowerPoint and Excel's ribbon as closely as possible.

Class Diagram of Actual System

The actual system is designed similar to the conceptual design created in Milestone 2, as shown in Figure 1.

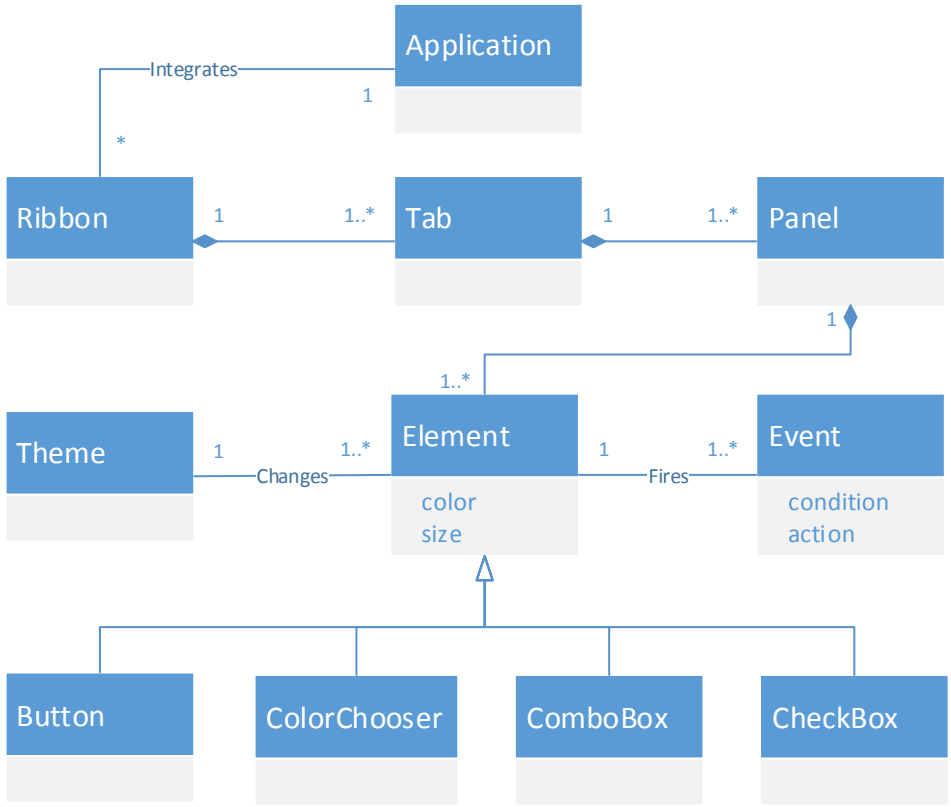


FIGURE 1: MILESTONE 2 CONCEPTUAL CLASS DIAGRAM

In this section, we will walk through the important elements of the application, while comparing it with the conceptual design above. Although this will not be an exhaustive discussion, due to the size of the library, we will cover what we feel are the most important parts of it, while keeping a focus on the items identified in the previous milestone's conceptual design.

Element

The various UI elements represented by the Element class in our class diagram, such as the Button, ComboBox, CheckBox, are represented by the `RibbonItem` abstract class and its children. From the class diagram generated by Visual Studio, we can see that the UI elements are separated by type, and when appropriate, separated into subtypes.

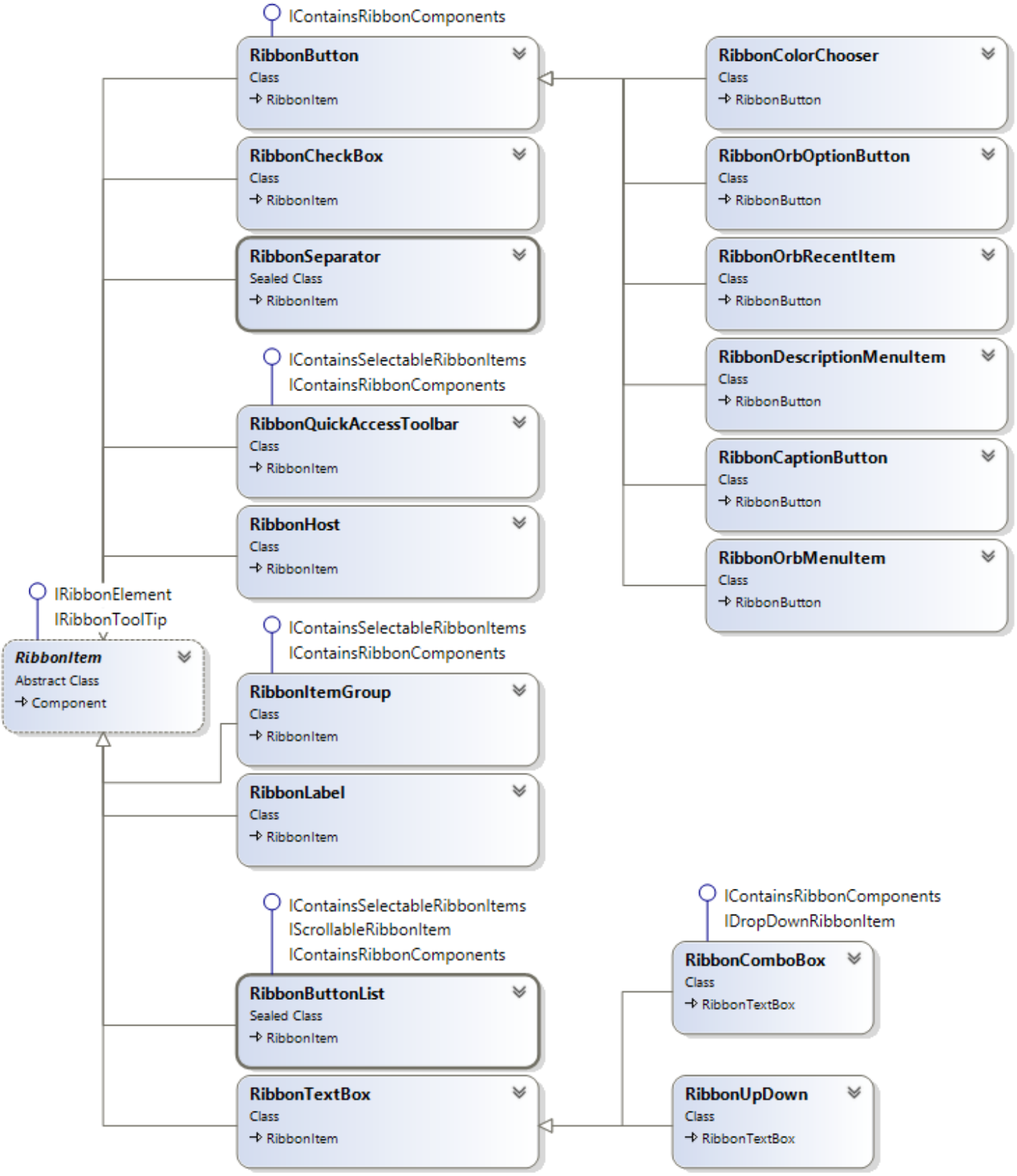


FIGURE 2: ACTUAL CLASS DIAGRAM FOR THE RIBBONITEM CLASS.

All UI elements are rendered by the

Ribbon, Tab, and Panel

The main ribbon components, such as the panels and tabs that compose the ribbon as well as the ribbon itself are represented by the `RibbonPanel`, `RibbonTab`, and `Ribbon` classes respectively. We notice from their class diagrams in Figure 3 that the classes are not associated together via any form of inheritance. However, given that a Panel cannot exist outside a tab, and that a tab cannot exist outside a ribbon, we can safely assume that the composition connections we've applied in our conceptual class diagram still applies.

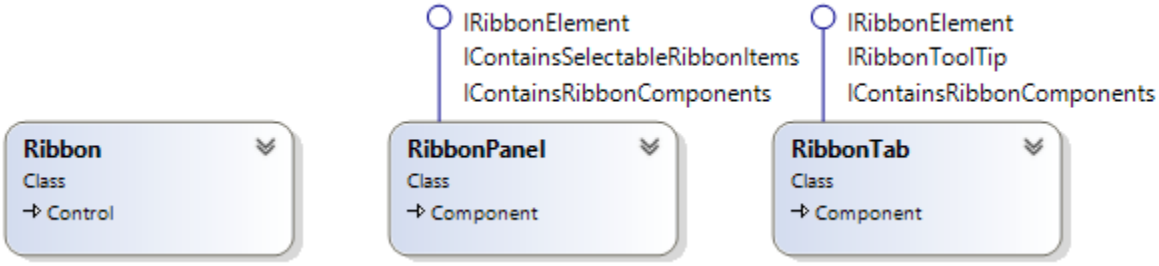


FIGURE 3: ACTUAL CLASS DIAGRAM FOR THE RIBBON, RIBBONPANEL, AND RIBBONTAB CLASSES.

If we view the code map generated by Visual Studio, we can see that they do in fact depend on each other. The arrows in the code map shows calls made from one class to another, where the number of calls are proportional to the thickness of the arrow. We notice that calls are made from the Ribbon class to the `RibbonTab` class, the `RibbonTab` class to the `RibbonPanel` class, and from the `RibbonPanel` class to the `RibbonItem` class. The way each class depends on the other enforces the assumptions made in Milestone 2.

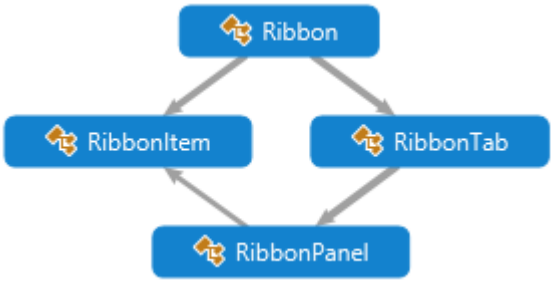


FIGURE 4: CODE MAP FOR THE RIBBON UI ELEMENTS

Event

The Event class in our diagram is implemented in the actual application by using nested classes. Every element which the end user can interact with contains an event handler delegate which is called when an action is taken against it. For example, if we take the `RibbonComboBox` class, we notice through the class diagram that it contains a nested delegate, `RibbonItemEventHandler`, as shown in **Error! Reference source not found.**.

We also notice the presence of multiple classes inheriting from C♯'s `EventArgs` class, as shown in Figure 5. These classes are mostly used for redrawing the user interface when a change occurs.

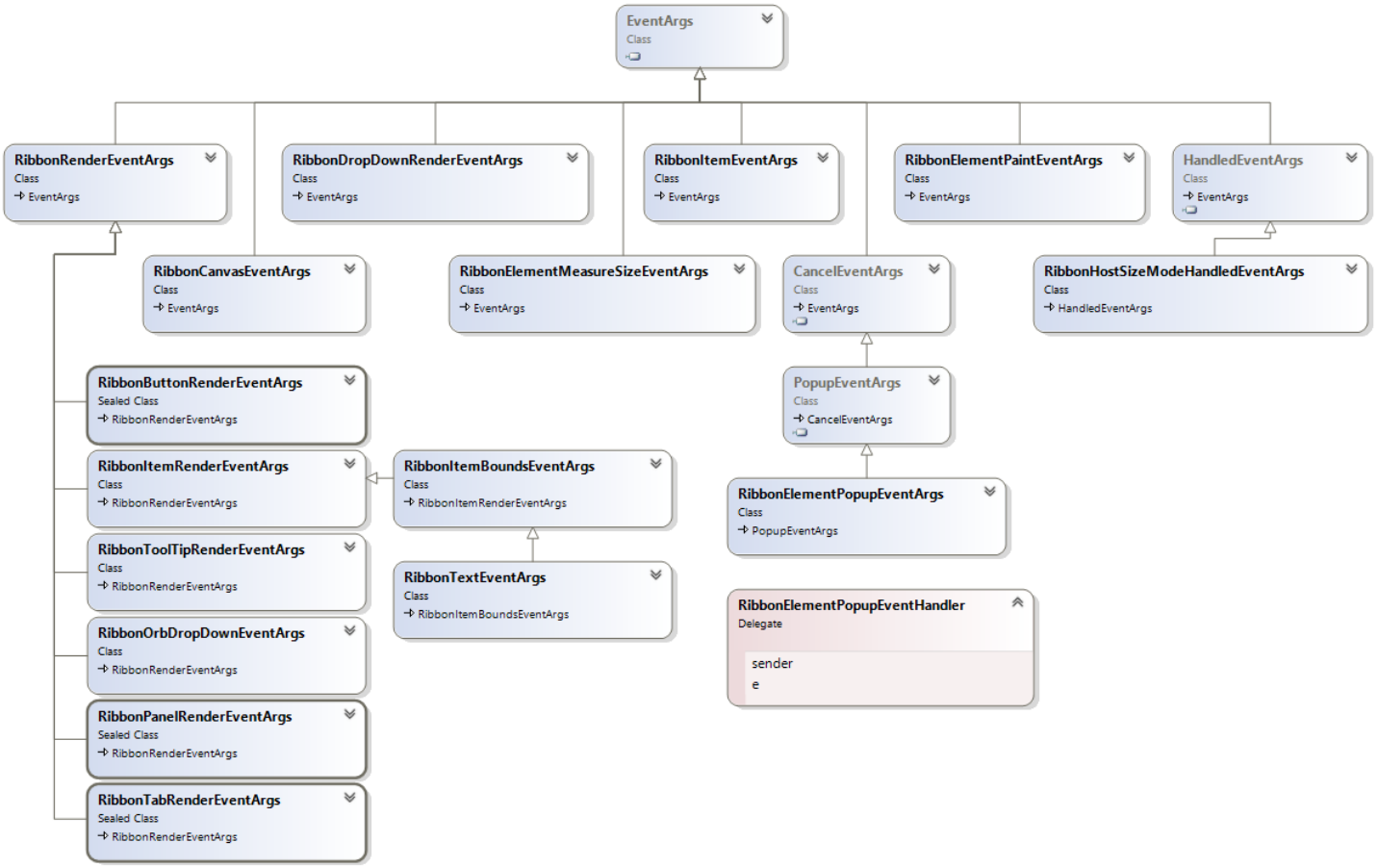
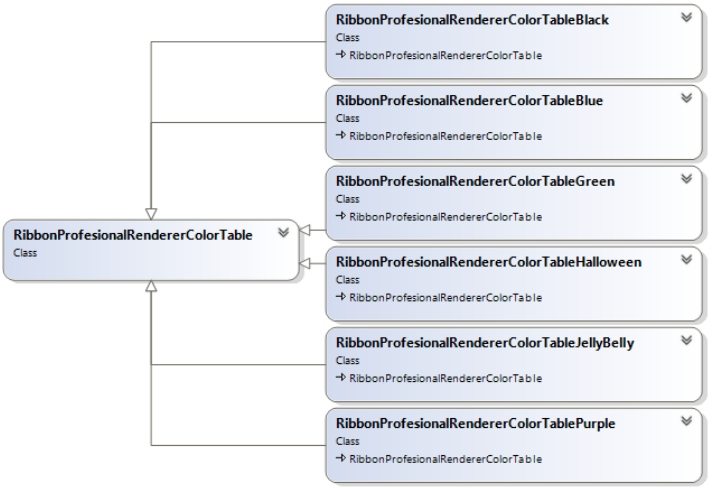


FIGURE 5: CLASSES INHERITING FROM EVENTARGS

Theme

The theme in our design represents the application's ability to change its color theme to match a color palette defined by the developer. In the application, it is defined by the `RibbonProfessionalRendererColorTable` and its subclasses. These classes are called by the Theme class's `ThemeColor` method, which uses a series of if statements in order to determine which class should be used to render the colors.




```
public static RibbonTheme ThemeColor
{
    get { return _Theme; }
    set
    {
        _Theme = value;
        //if (blnSetOnly == false) {MainRibbon.ThemeColor = _Theme;} //08/01/2013 Michael Spradlin - 
Changed to Code Below because if MainRibbon was set to null it was blowing up and you should still be 
able to set the theme even if you are not using the ribbon control.
        if (blnSetOnly == false)// && MainRibbon != null)
        {
            //MainRibbon.ThemeColor = _Theme;
```

```
            if (ThemeColor == RibbonTheme.Blue | ThemeColor == RibbonTheme.Normal)
                ColorTable = new RibbonProfesionalRendererColorTable();
            else if (ThemeColor == RibbonTheme.Black)
                ColorTable = new RibbonProfesionalRendererColorTableBlack();
            else if (ThemeColor == RibbonTheme.Green)
                ColorTable = new RibbonProfesionalRendererColorTableGreen();
            else if (ThemeColor == RibbonTheme.Purple)
                ColorTable = new RibbonProfesionalRendererColorTablePurple();
            else if (ThemeColor == RibbonTheme.JellyBelly)
                ColorTable = new RibbonProfesionalRendererColorTableJellyBelly();
            else if (ThemeColor == RibbonTheme.Halloween)
                ColorTable = new RibbonProfesionalRendererColorTableHalloween();

        }

        //System.Windows.Forms.ToolStripColors.SetUpThemeColors(blnRenderOnly);
    }
}
```

Code Smells and Possible Refactorings (5 marks)

EventArgs Hiearchy

Problem identified

There are four classes forming a hierarchy with each class being the parent of the next. The classes `RibbonRenderEventArgs`, `RibbonItemRenderEventArgs`, `RibbonItemBoundsEventArgs`, and `RibbonTextEventArgs` are ordered from highest to lowest within the hierarchy tree. The relationship between the four classes come from having additional instance variables as the hierarchy descends to the lowest child, namely `RibbonTextEventArgs`, which encompasses the most instance variables of the four. This leads to the next observation showing an increase in the number of constructor parameters moving down the hierarchy. Up to nine parameters can be found in a constructor within `RibbonTextEventArgs`. There exists a total of 4 constructors in `RibbonTextEventArgs` class overloaded with different parameter options that have very similar functions. All functions that create the above mentioned four classes are children of the abstract class `RibbonItem`. Whenever one of the four mentioned classes are created, the `RibbonItem` type class will take the responsibility of filtering out the necessary instance variables before passing the list into the constructor which should be the responsibility of the above mentioned four classes.

Code smell(s): long parameter list, duplicated code.

Solution: We will use Preserve Whole Method to reduce long parameter list and reduce stress on the calling class. Next Extract Method is used to reduce duplicate code specifically in the overloaded constructors within `RibbonTextEventArgs`.

1. In `RibbonItemRenderEventArgs`, `RibbonItemBoundsEventArgs`, `RibbonItemBoundsEventArgs`, and `RibbonTextEventArgs`, change constructors to accept `RibbonItem` and `RibbonElementPaintEventArgs`(an argument used external to `RibbonItem`) and remove any parameters that can be derived from the two new arguments.
2. At each occurrence in which a `RibbonItem` type object calls one of the above four mentioned classes, parameters passed will need to change. Instead of passing its own instance variables, it will pass itself in as a parameter.
3. Within `RibbonTextEventArgs`, create a new methods to delegate parsing tasks that currently the caller method within `RibbonItem` type objects are doing leading to overloaded constructors and duplicate instance variable set statements. The checks will be moved to `RibbonTextEventArgs` resulting in a single constructor along with helper methods for determining which approach to use.

Color Theme Inheritance

Problem Identified: The class `RibbonProfessionalRendererColorTable` is used to apply a theme color to the ribbon. There are in total 6 theme color and this class represent one. For each of the other five theme color, there is a class which inherits from `RibbonProfessionalRendererColorTable`. There are no difference between the super class and the subclasses, except for the color that they used. In this case, there are no valid reasons to have an inheritance tree since the subclasses do not reuse the behaviors of the super class or add new functionalities; they are literally the super class. Also, some methods are duplicated in the super class and the subclasses and one of the subclasses is never invoke in the whole project. Finally, conditional logic is used to instantiate an object of this inheritance tree in the *Theme* class.

Code smells: Lazy class, duplicate code, dead code, similar subclasses, use of conditional logic to determine behavior

Solution: Since the code to change the color is the same across all classes in the inheritance tree, this is how we will refactor this code:

1. Pull out this information out of each class and create properties files for each color. In each file, we will associate all field variables defined in the super class to apply a theme with their appropriate hexadecimal color value.
2. Create a temporary class, `TempRibbonProfessionalRendererColorTable`, and copy the class `RibbonProfessionalRendererColorTable` in it.
3. In `RibbonProfessionalRendererColorTable`, create another constructor taking as a parameter the conditional logic value used in the `Theme` class. Move the conditional logic in the constructor but instead of creating a new object, call a method that will get the properties file, depending on the conditional logic value, and set all the theme field variables by using it. Also, set all theme field variables to their default value.
4. Test all theme color one by one. For each test that succeeds, delete the related class. At the end, we should have only one class and 6 properties files. Delete `TempRibbonProfessionalRendererColorTable`.
5. Replace the conditional logic in the *Theme* class by calling the new constructor of the `RibbonProfessionalRendererColorTable` and by passing the conditional logic value as its parameter.

The final hierarchy will be like the one shown in Figure 6.

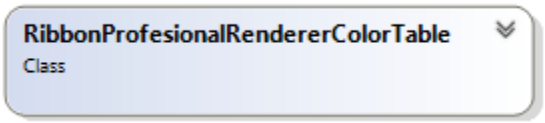


FIGURE 6: FINAL HIEARCHY FOR THE COLOR THEME

Mouse-click procedure

Problem identified: In the class `GlobalHook`, we have a method called `MouseProc` which processes mouse procedures, such as moving the mouse wheel or double clicking. Since there are 14 kinds of mouse gestures that can be processed for a given program and there is a corresponding conditional statement that has 14 different logical cases. This method is too long because of this conditional, and so it must be shortened to preserve its cohesiveness. The condition depends on arguments of the same type, i.e. from the `WinApi` class. Therefore, using the "Replace Conditional with Polymorphism" shouldn't be used to fix this code smell.

Code smell(s): long method

Solution: We will use the Extract Method refactoring method to fix the long method.

1. Create a helper method called `OnMouseAction` in the class `GlobalHook`, which is a private void method that takes the `msg` variable as a parameter

2. Copy the conditional statements (all if-else statements from the `MouseProc` method) into the `OnMouseAction` method.
3. Delete the conditional statements from the `MouseProc` method
4. Add a call to the `OnMouseAction` method from where the deleted conditional statements were called.

Glyph Module

6. **Problem identified**: There is code duplication within the 4 classes related to Glyphs, both within and across classes. More precisely, the classes `RibbonOrbAdornerGlyph`, `RibbonPanelGlyph`, `RibbonQuickAccessToolbarGlyph` and `RibbonTabGlyph` contain code that is duplicated between each other. Additionally, the classes `RibbonQuickAccessToolbarGlyph` and `RibbonTabGlyph` have duplicated code within their individual class. The first problem of code duplication across these 4 children of the `Glyph` class is prioritized for refactoring, while the latter problem poses less risk.

Code smell: Duplicated Code, both across related classes and within each class

Solution: The following refactoring methods are required to fix the code duplication: Extract Superclass, Pull Up Field, Pull Up Constructor Body, Pull Up Method, Form Template Method, Extract Method.

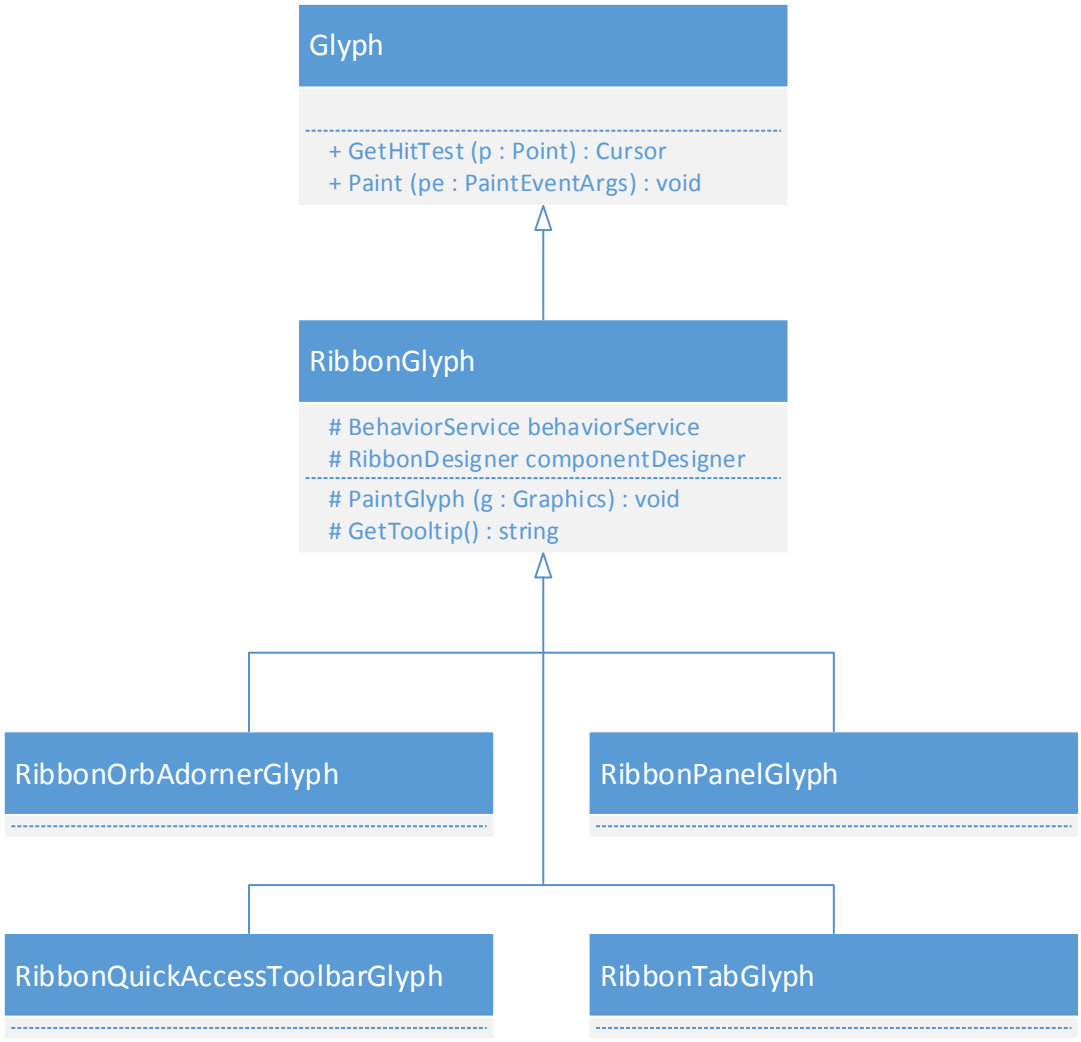
For the classes affected by duplicated code in the hierarchy:

7. Use “Extract Superclass” to have a new parent named for these 4 classes, named `RibbonGlyph` which will be in the middle of the hierarchy, below `Glyph`.
1. Use “Pull Up Field” to move duplicated fields from these 4 classes up to the parent.
8. Use “Pull Up Constructor Body” to create a new constructor in `RibbonGlyph` that will initialize these fields.
9. Use “Pull Up Method” to move the method `GetHitTest`” to the parent class, because this method is identical across these 4 classes
10. Use “Form Template Method” to extract the similarities and create virtual methods to allow variation for the method `Paint`, because it is present in all 4 classes and has similar implementations, but the method differs in a few statements.
11. See the `RibbonGlyph` Diagram for the final hierarchy.

For the classes affected by duplicated code that is not shared with others:

1. Extract into a method the 2 conditional blocks that have all statements in common except one.
2. Add one parameter to this method that will determine which varying statement to use.

RibbonGlyph Diagram



Code Snippet for Glyph Module Refactoring

```
public class RibbonPanelGlyph : Glyph
{
    BehaviorService _behaviorService;
    RibbonTabDesigner _componentDesigner;

    public override Cursor GetHitTest(System.Drawing.Point p)
    {
        if (Bounds.Contains(p))
            return Cursors.Hand;
        return null;
    }

    public override void Paint(PaintEventArgs pe)
    {
        SmoothingMode smbuff = pe.Graphics.SmoothingMode;
        pe.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
        using (GraphicsPath p = RibbonProfessionalRenderer.RoundRectangle(Bounds, 9))
        {
            using (SolidBrush b = new SolidBrush(Color.FromArgb(50, Color.Blue)))
            {
                pe.Graphics.FillPath(b, p);
            }
        }
        StringFormat sf = new StringFormat(); sf.Alignment = StringAlignment.Center; sf.LineAlignment =
StringAlignment.Center;
        pe.Graphics.DrawString("Add Panel", SystemFonts.DefaultFont, Brushes.White, Bounds, sf);
        pe.Graphics.SmoothingMode = smbuff;
    }
}

public class RibbonQuickAccessToolbarGlyph : Glyph
{
    BehaviorService _behaviorService;
    RibbonDesigner _componentDesigner;

    public override Cursor GetHitTest(System.Drawing.Point p)
    {
        if (Bounds.Contains(p))
            return Cursors.Hand;
        return null;
    }

    public override void Paint(PaintEventArgs pe)
    {
       if (_ribbon.CaptionBarVisible && _ribbon.QuickAcessToolbar.Visible)
       {
          SmoothingMode smbuff = pe.Graphics.SmoothingMode;
          pe.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
          using (SolidBrush b = new SolidBrush(Color.FromArgb(50, Color.Blue)))
          {
             pe.Graphics.FillEllipse(b, Bounds);
          }
          StringFormat sf = new StringFormat(); sf.Alignment = StringAlignment.Center; sf.LineAlignment
= StringAlignment.Center;
          pe.Graphics.DrawString("+", SystemFonts.DefaultFont, Brushes.White, Bounds, sf);
          pe.Graphics.SmoothingMode = smbuff;
       }
    }
}
```