OFFICE 2007 RIBBON ACTIVE PROJECT

Final Report

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

Concordia University

Fall 2013

Contents

Project Description

The Microsoft Office 2007 Ribbon Bar is an open source control which allows for .NET developers to add an intuitive Office look and feel to their C# or Visual Basic applications. The control has been designed to be easy to use not only for developers who will be implementing the ribbon for their applications, but also for the end users who will be using the application, by providing a familiar experience, mimicking Word, PowerPoint or Excel's ribbon as closely as possible. [1]

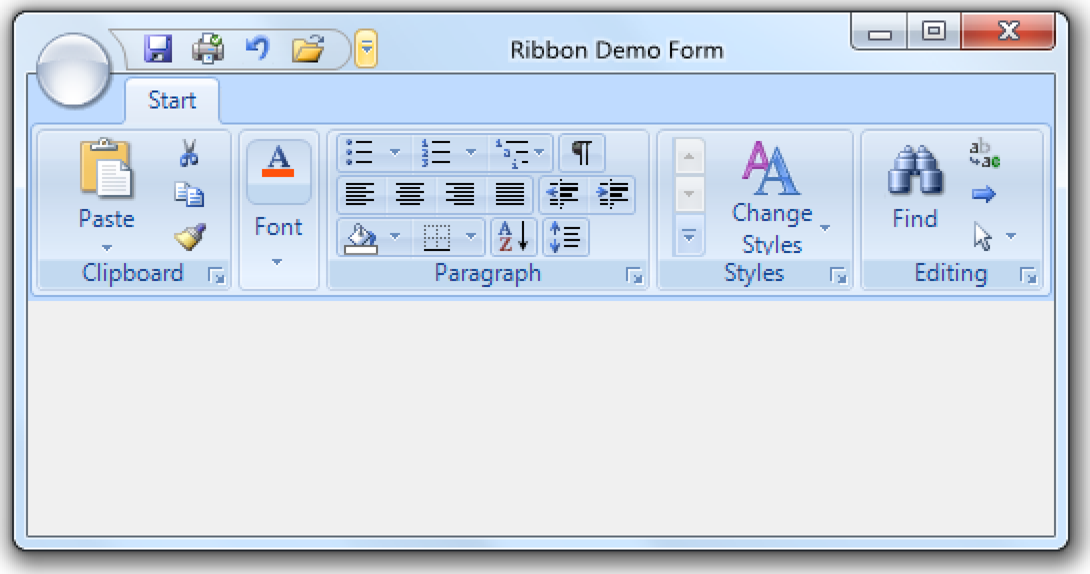


Figure 1: Ribbon Demo Form [1]

Given that this project is only a component for other projects, the domain is a bit more difficult to define. Based on how the project is structured and the name of the classes, the Ribbon, as well as its internal elements (buttons, item groups, tabs, etc.) all form part of the domain.

The project's last stable release was on May 10th 2013, with the last edit having been committed on September 12th 2013. It is currently maintained by 23 people, whose tasks have been outlined in Table 1 below.

Project Members

Coordinators

c0wboy	Jan 12, 2013
Inoodat	Aug 16, 2011
kcarbis	Jul 9, 2011
koglth	Aug 4, 2011

Editors

strims	Feb 26, 2013
yuvrajSengar	Jan 12, 2013

Developers

abhinav2600	Mar 21, 2013
adriancs	Apr 13, 2012
Aidin_Rain	Aug 24, 2012
Ch0pin	Aug 20, 2013
dryasser	Aug 16, 2012
fizmhd	Apr 13, 2012
gregoireboillet	Sep 12, 2013
ju52	Aug 20, 2013
kchenaoui	Aug 18, 2011
menendezpoo	Jul 11, 2011
milesstones	Aug 9, 2011
Ohio232Stang	Feb 19, 2013
RHarden	Apr 2, 2013
sdvedmund	Aug 16, 2012
SebastianDotNet	Dec 1, 2011
toAtWork	Jan 25, 2013
wlatuch	Jul 22, 2013

Table 1: List of members on the Office 2007 Ribbon Project

Project Size and Scope

The static analysis tool Understand from Scientific Toolworks allowed us to obtain software metrics that support the refactoring of this project. The different metrics obtained are project metrics, program unit complexity, class metrics, and class object-oriented metrics.

This project has 20693 lines of codes and 99 classes. Dividing the effort for 5 members yields about 4000 lines and 20 classes each, which is a reasonable quantity. Next looking at the cyclomatic complexity for each methods, we detect a high risk for numerous methods. More specifically, there are 18 methods with complexity M between 11 and 20, 5 methods with M between 21 and 30, and 2 methods with M = 136. Additionally, there is one method whose complexity number is 31 and has 2102 different execution paths.

For class specific metrics, we take a look at each classes' SLOC (source lines of codes). There are 6 classes whose SLOC lie between 1000 and 1999 lines. At the extreme, there is the Ribbon class with 2259 SLOC and the RibbonProfessionalRenderer class with 4054 SLOC. This Ribbon class also suffers from the highest Coupling Between Objects metric with a measure of 57 and the highest Weighted Method Count of 55. Finally, there are 15 classes with a Lack of Cohesion Methods percentage of over 90%, which demonstrates a moderate risk for maintenance.

As shown by these metrics, we are able to identify a small number of classes and methods with very high risk. Since these high risks present themselves within less than 10 classes, it is feasible to refactor them within the duration of this term project. If time permits, then we will address the classes of lower risk.

Group Members

For each group member, describe the experience and skills that he or she will contribute and try and relate this to the chosen project (one paragraph each). You should pick a project that is written in a language that you know and that uses an API you are familiar with.

Shahrad Rezaei

Shahrad Rezaei has decent experience working with Java and C#, mostly writing mobile applications for Android and Windows Phone, as well as the school assignments which are mostly done in Java. He also enjoys setting up servers in my spare time. Since he already has some basic knowledge of C#, he feels that he could contribute well to this project, as well as learn how to maintain his code in the future.

Jignesh Patel

Jignesh Patel's strongest software development asset is developing in C# and C++. He used this experience during an internship at CAE Inc., where he was developing features for an 3D airport editor. He also has some experience in automated and manual testing, as well as some more development skills in Java. He believes that his greatest contribution to this project will his ability to write efficient code and think of novel ways of improving and contributing to the other team member's ideas.

Felix Soumpholphakdy

Felix Soumpholphakdy enjoys using C# for game programming through frameworks like Microsoft XNA and Unity3D. Additionally, he is a supporter of Extreme Programming's merciless refactoring rule and has read Robert C. Martin's Clean Code and the Gang of Four's Design Patterns book to reinforce his refactoring knowledge. Felix' experience and code intuition will be useful in detecting code smells and using proven solutions to erase coding horrors. He also has the ability to well manage the work between team members if the division of tasks given a project is complex.

Paterson Deshommes

Paterson Deshommes is a third year software engineering student at Concordia University. Before entering university, he completed a technical DEC in computer science and he also acquired working experience through multiple work terms. He mostly has experience in Object-Oriented languages such as Java, C# and C++ but he also possesses knowledge of other type of languages such as C, Prolog and Haskell. He also has experience in web programming, database design and management, embedded programming, software requirement analysis and software architecture design. His knowledge of the C# language and its experience in architecture design will make him an asset for the team.

Chun Kit Liu

Although Java is Chun Kit Liu's most comfortable programming language, he knows the basics of both C# and C++. After developing a game with C# using XNA, he feel very comfortable working with this language.

Since Ribbon is written in C#, his understanding of the code will come much easier. Another project he worked on was for a GUI written in C++ using MFC. He hopes some of the knowledge learnt from that project can be used in the Ribbon project as well.

Personas, Actors, and Stakeholders

The following personas have been identified as stakeholders for our project. Due to the nature of our project, we believe that the developer is a main persona, since, from the point of view of the ribbon, the developer is the one who will be interacting directly with the system, and will be the one receiving the most benefit from the program. This focus is also made evident by the project's development team, who made it as simple as possible to integrate with Visual Studio, with the least amount of code possible.

Main Personas

Developer

Bob is a software development consultant writing custom internal applications for Contoso Inc. His job requires him to write applications for an accounting firm, who are not very tech-savvy, but are familiar with the Office 2007 user interface, specifically the ribbon. As such, he uses the Office 2007 Ribbon Active Project to integrate a ribbon in his user interface, in order to ensure that the employees feel more comfortable and not pressured to learn another new user interface. It is imperative that he develops user-friendly software for Contoso, since his contract will be terminated immediately if he receives complaints from the firm.

Furthermore, since the project is easy to integrate into his .NET applications, spending extra time developing a new user interface would not be required. Therefore, he can spend more time bringing value to the organization.

End User

Jimmy Turner is a young man studying in journalism in the Faculty of Arts and Science at Concordia University. As such, he often has to write reports for his classes. Also, he is very passionate about his field and since he began school, he has been writing articles for "The Link", the independent newspaper of Concordia. Moreover, he is an apprentice novelist and he is writing stories during his free time. He really likes to work with Microsoft Word as he feels that the user interface is really intuitive. However, Jimmy does not have the time to work because of his busy schedule and as such cannot afford buying the Office suite every year. He would not mind using another text editor but, he doesn't like the user interface of the other products out there he would like to get something similar to Microsoft Word at an affordable price or free.

Secondary Actors

Text Editor

The text editor in use by Jimmy the journalism student is a secondary actor by providing services to the system. More specifically, the Ribbon offers control over the text documents, but it is the text editor's responsibility to apply the actions requested by these control elements.

Other Stakeholders

Salesman

Sonny Sagan is a software salesman who specializes in selling enterprise-scale software to blue-chip companies for small software companies. His role consists of presenting a given software product to the buyers of these companies, and negotiating a price for the software or the acquisition of the small company. He is a stakeholder for the Microsoft Office 2007 Ribbon because he is in the process of selling a new and innovative application to Microsoft. He would like to have a familiar interface during his next sales pitch to the Microsoft buyers, so that they could imagine this new software being a part of the Microsoft software suite.

Informal Use Case

Developer

Integrate library into Visual Studio

Actors		Developer
Pre-conditions		The developer has a project already created, and wishes to add a ribbon to his or her UI.
Main Scenario		1. The developer **opens** the project that will **integrate** the ribbon in his or her IDE; 2. With the form **open**, the developer **opens** the Toolbox view, and **selects** "Choose Items…" from the context menu; 3. The user **loads** the dynamic link library **included** with the project, and **selects** "Ribbon" in the list of controls; 4. The developer **drags** the Ribbon item from the Toolbox view onto the form to **add** it to his project.
Alt. Scen.	Code Error	1. The developer **encounters** an error at some point in the process; 2. Making sure that the issue is **not caused** by an error in his or her own code, a bug report should be **submitted** to the open source project.

Adding an element to the ribbon

Actors	Developer
Pre-conditions	- Ribbon has been dragged onto the developer's form, and is now part of the UI; - The developer's IDE is open, and he or she is on the Forms view.
Main Scenario	1. The developer wishes to **add** a button or control to the ribbon; 2. The developer **selects** the appropriate link in the Properties view to **add** a tab to the ribbon; 3. With the tab **selected**, the developer can **select** the appropriate link in the Properties view to **add** a panel to the tab; 4. With the panel **selected**, the developer can **select** the appropriate link in the Properties view to **add** a button, a combo box, a color picker and other elements to **add** to the panel;

Alt. Scen.	Code Error	1. The developer **encounters** an error at some point in the process; 2. Making sure that the issue is **not caused** by an error in his or her own code, a bug report should be **submitted** to the open source project.

Removing an element

Actors		Developer
Pre-conditions		- Ribbon has been dragged onto the developer's form, and is now part of the UI; - The developer's IDE is open, and he or she is on the Forms view. - The ribbon contains elements which need to be removed
Main Scenario		1. The developer **selects** the element which he or she wants to **remove** from the ribbon; 2. The developer **presses** the Delete key.
Alternative Scenarios	Code Error	1. The developer **encounters** an error at some point in the process; 2. Making sure that the issue is **not caused** by an error in his or her own code, a bug report should be **submitted** to the open source project.
	Wrong Element Removed	1. The developer accidentally **removes** the wrong element; 2. He or she can follow the steps in the "Adding an Element" use case to restore the element.

Giving a behavior to an element

Actors		Developer
Prerequisites		- Ribbon has been dragged onto the developer's form, and is now part of the UI; - The developer's IDE is open, and he or she is on the Forms view; - The ribbon contains elements which need to react to events.
Main Scenario		1. The developer **selects** the element for which he or she wishes to **add behavior** to; 2. The developer **changes** the Properties view to **display** event handlers; 3. The developer **double-clicks** the appropriate event he or she wants the element to **respond to**; 4. The IDE will **switch** to the code view, where the developer can **enter** the code necessary to **trigger** the required event.
Alt. Scen.	Code Error	1. The developer **encounters** an error at some point in the process; 2. Making sure that the issue is **not caused** by an error in his or her own code, a bug report should be **submitted** to the open source project.

Make a new color theme

Actors		Developer
Prerequisites		- The developer has obtained the demo application from the Office 2007 Ribbon Active Project's repository; - The developer has loaded the demo application into his or her IDE;
Main Scenario		1. The developer **launches** the demo application, and is **presented** with a UI where he or she can **select** a demo; 2. The developer **selects** the "Theme Builder Form" button in order to **launch** the Theme

		Builder demo; 3. In the Theme Builder demo, the developer can **select** colors for every ribbon element and **preview** them in the demo's user interface; 4. Once the color theme is to the developer's **liking**, he or she can **export** the theme as an XML or INI file, to be **used** in the application.
Alternative Scenarios	Theme does not change	1. The application's colors do not **change** after the developer **applies** a new theme; 2. The developer should **ensure** that the application **refreshes** by **adding** the appropriate code to the application.
	Code Error	1. The developer **encounters** an error at some point in the process; 2. The developer ensures that the new theme class inherits from the color rendering class; 3. Making sure that the issue is **not caused** by an error in his or her own code, a bug report should be **submitted** to the open source project.

Bug and Issue reporting

Actors		Developer
Prerequisites		- The developer has encountered an issue with the ribbon; - The developer has taken proper steps to ensure that the issue is not caused by his or her own code.
Main Scenario		1. The developer **encounters** a code-related issue **linked** to the ribbon while **working** on his or her project; 2. The developer **navigates** to the discussion page of the Office 2007 Ribbon Active Project; 3. The developer **creates** a new discussion, **detailing** how the defect was **produced**, what error messages were **reported** by the application, as well as any other information **deemed** useful by the developer towards **solving** his or her issue; 4. The developer **submits** the code which **generated** the defect.
Alt. Scen.	Missing Info	1. The developer **enters** a bug report on the open-source project site; 2. The developer **forgets** tracking information, such as the error message produced, the faulty code producing the error, etc. 3. The developer should modify his or her report, and include the missing information.

Journalism Student

Formatting text with a picture in the middle

Actors		Journalism Student
Main Scenario		1. User **selects** the image 2. User **clicks** on the format submenu 3. User **selects** the "wrap image" option from the Ribbon 4. User **chooses** the option which puts the picture in the middle of the text.
Alt. Scenario	Picture too large	1. The picture is too large for the text to wrap around it properly; 2. The user can shrinks the size of the picture to make it fit; 3. The text now properly wraps around the picture.

Adding a colored footnote

Actors	Journalism Student
Main Scenario	1. <u>User</u> places cursor next to the text that requires a footnote 2. <u>User</u> **selects** the <u>tab</u> containing the footnote button 3. <u>User</u> **clicks** on the footnote <u>button</u> 4. <u>User</u> **selects** the footnote 5. <u>User</u> **clicks** on the <u>color chooser</u> 6. <u>System</u> **displays** a <u>window</u> with color choices 7. <u>User</u> **selects** his desired color

Formatting margins

Actors	Journalism Student
Main Scenario	1. <u>User</u> **selects** the <u>tab</u> containing formatting <u>elements</u> 2. <u>User</u> **clicks** on the margins <u>ComboBox</u> 3. <u>System</u> **displays** a <u>window</u> with margin choices 4. <u>User</u> **clicks** on the margin of interest 5. <u>User</u> **selects** the tab containing the view <u>elements</u> 6. <u>User</u> **clicks** on the ruler <u>CheckBox</u> 7. <u>Text editor</u> displays a ruler for margins

UML Diagram

The UML diagram depicted in Figure 1 has been created based on the important concepts highlighted in the various use cases identified in the previous section.

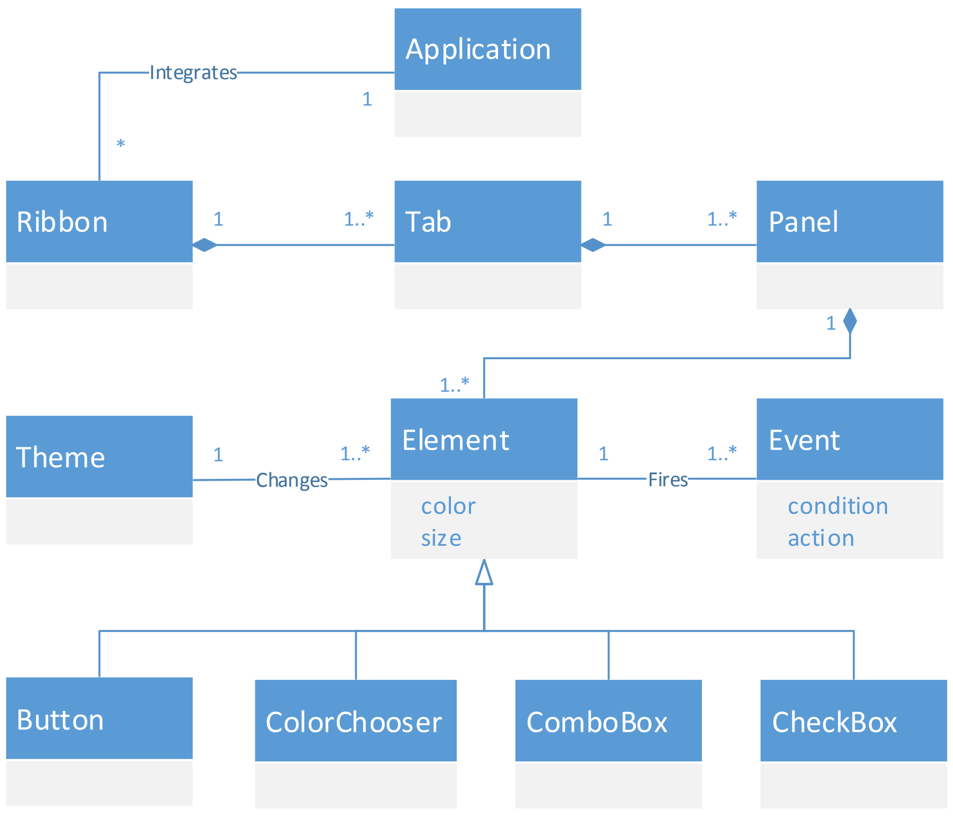


Figure 2: UML Domain Model based on identified use cases

There exists a one-to-many relationship between Application and Ribbon respectively as more than one type of Ribbon may exist in an Application.

For a Ribbon to exist at least one Tab object should be present. Otherwise the Ribbon would not have any functionality. Similarly, each Tab may exist only if one or more Panel object is present. Furthermore, each Panel contains the Element objects which are responsible for presenting End-Users with an access point to various tools. As with the Tab object[s], Panel object[s] may only exist with a minimum of one Element present.

Each Element object fires Event[s]. The latter responsibility represents a one-to-many relationship suggesting that a single User click to a Button or ComboBox may trigger multiple signals to the system. The mentioned Button and ComboBox objects are both children of Element. As such, overall Element attribute changes such as color and size will shape a Theme giving it a many-to-one relationship between Element and Theme respectively.

Finally, Theme is responsible mostly for shaping Element objects. This may include changing the color and size of various types of Button or ComboBox objects. As such, one Theme configuration is responsible for the changes in the cosmetics of Element[s] giving it a one-to-many relationship between Theme and Element respectively.

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.

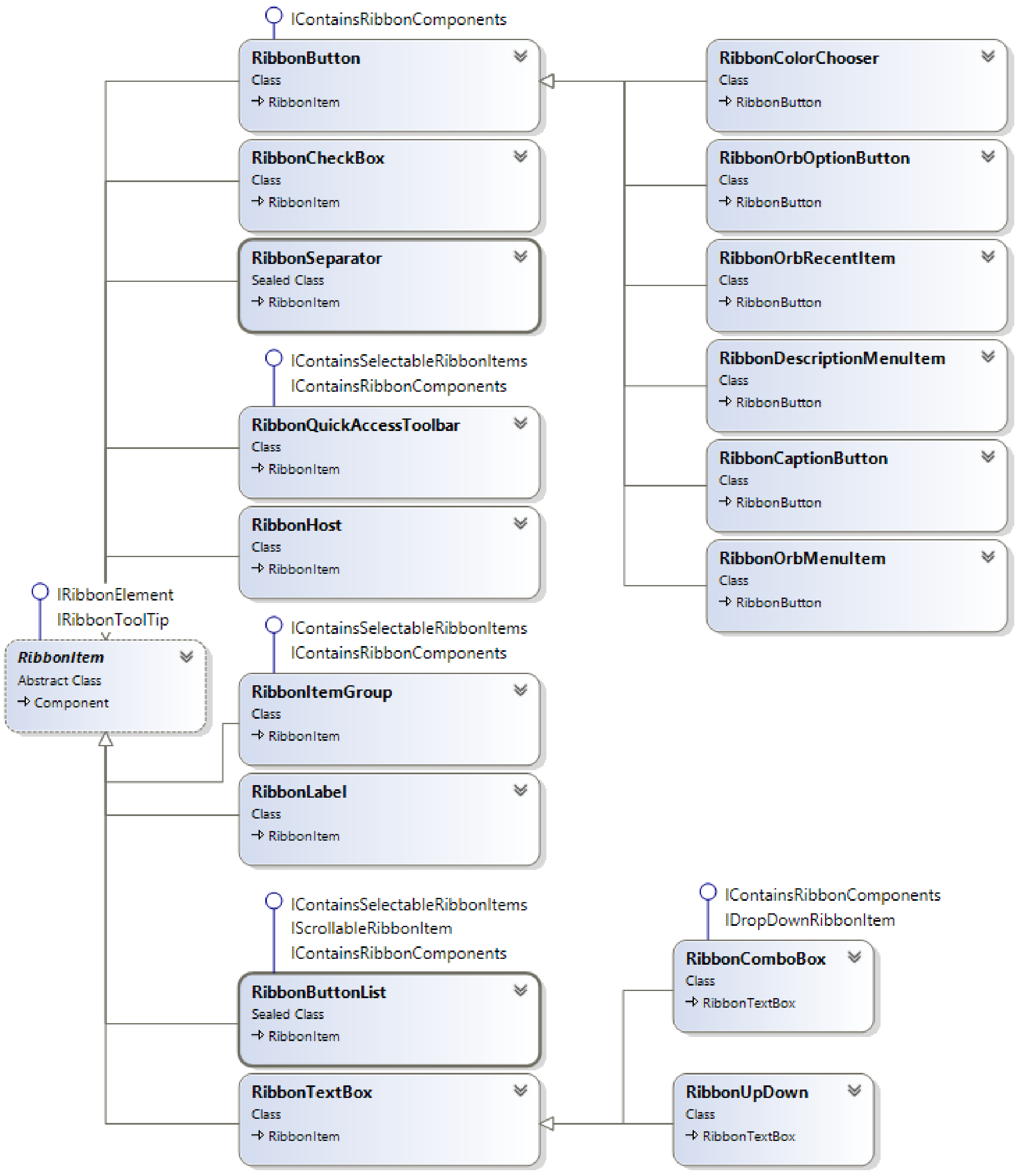


Figure 3: Actual class diagram for the RibbonItem class.

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 4 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.

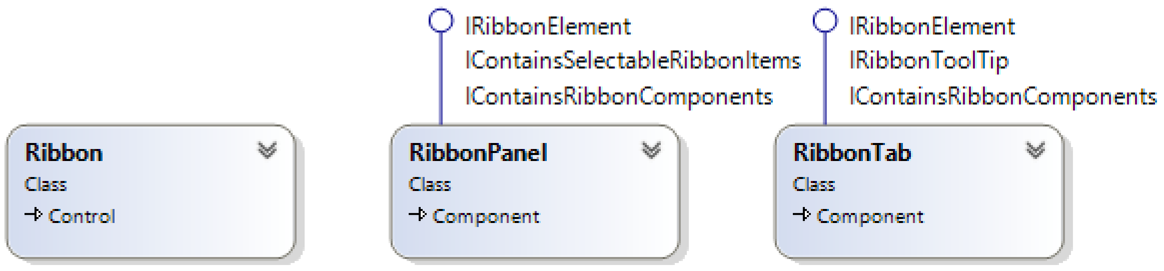


Figure 4: Actual class diagram for the Ribbon, RibbonPanel, and RibbonTab classes.

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 is 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.

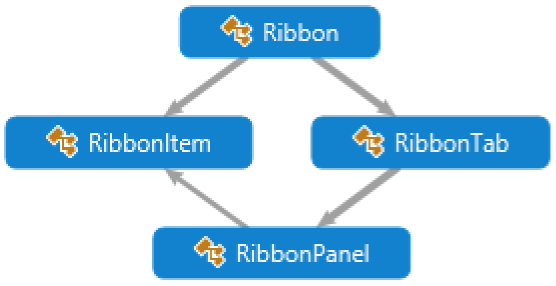


Figure 5: Code map for the ribbon UI elements

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 Figure 6.

We also notice the presence of multiple classes inheriting from C♯'s `EventArgs` class, as shown in Figure 6. These classes are mostly used for redrawing the user interface when a change occurs.

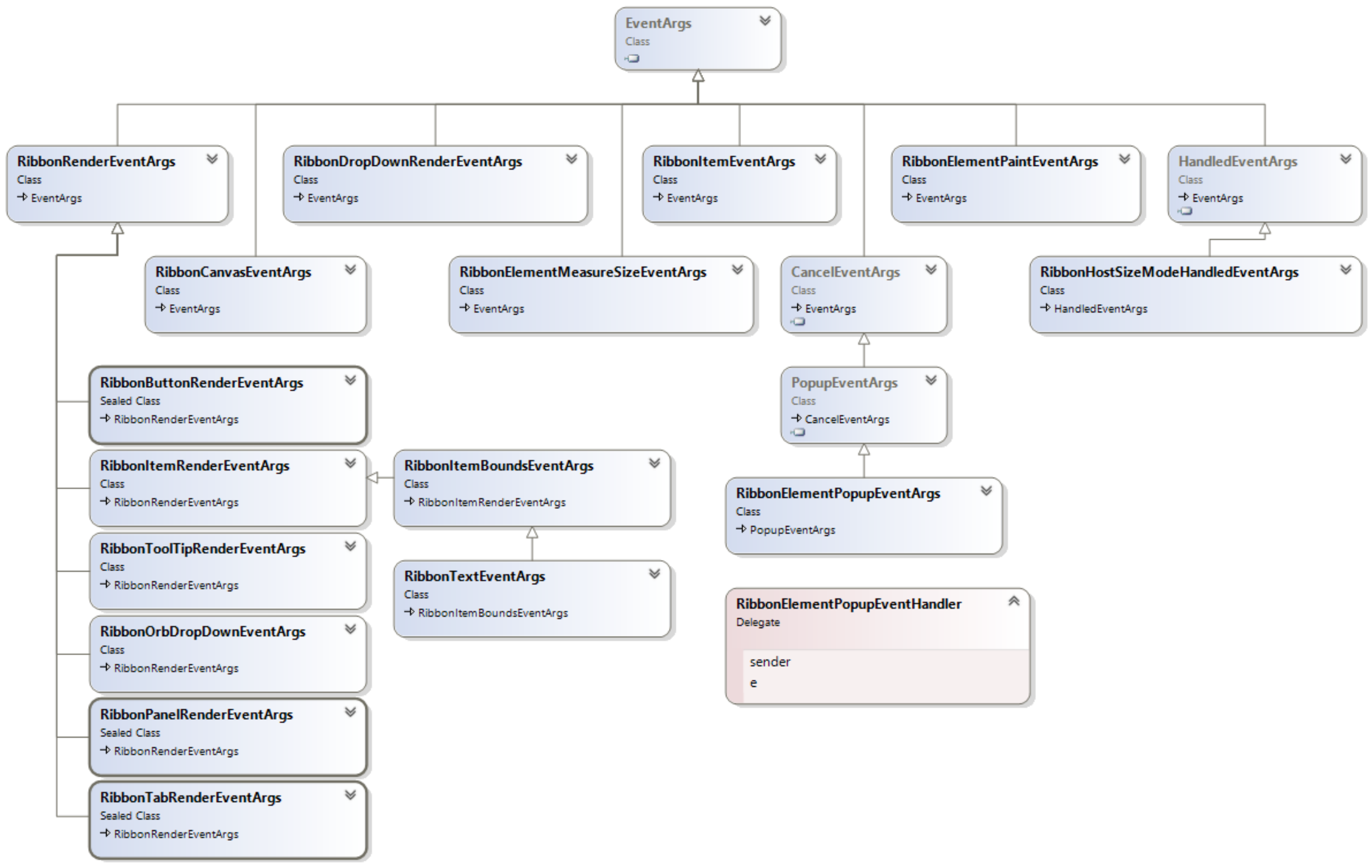
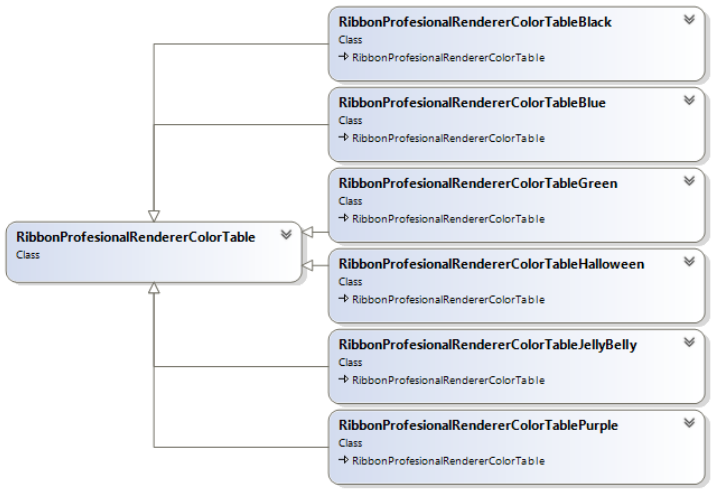


Figure 6: Classes inheriting from EventArgs

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.


RibbonProfesionalRendererColorTableBlack
Class
RibbonProfesionalRendererColorTable
RibbonProfesionalRendererColorTableBlue
Class
RibbonProfesionalRendererColorTable
RibbonProfesionalRendererColorTableGreen
Class
RibbonProfesionalRendererColorTable
RibbonProfesionalRendererColorTable
Class
RibbonProfesionalRendererColorTableHalloween
Class
RibbonProfesionalRendererColorTable
RibbonProfesionalRendererColorTableJellyBelly
Class
RibbonProfesionalRendererColorTable
RibbonProfesionalRendererColorTablePurple
Class
RibbonProfesionalRendererColorTable

```
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.SetUpTheme
Colors(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 7.

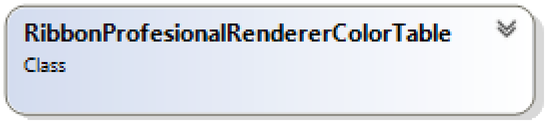


Figure 7: Final hiearchy for the color theme

Mouse-click procedure

Problem identified: In the class `GlobalHook`, we have a method called `MouseProc` which processes mouse procedures, like 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

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:

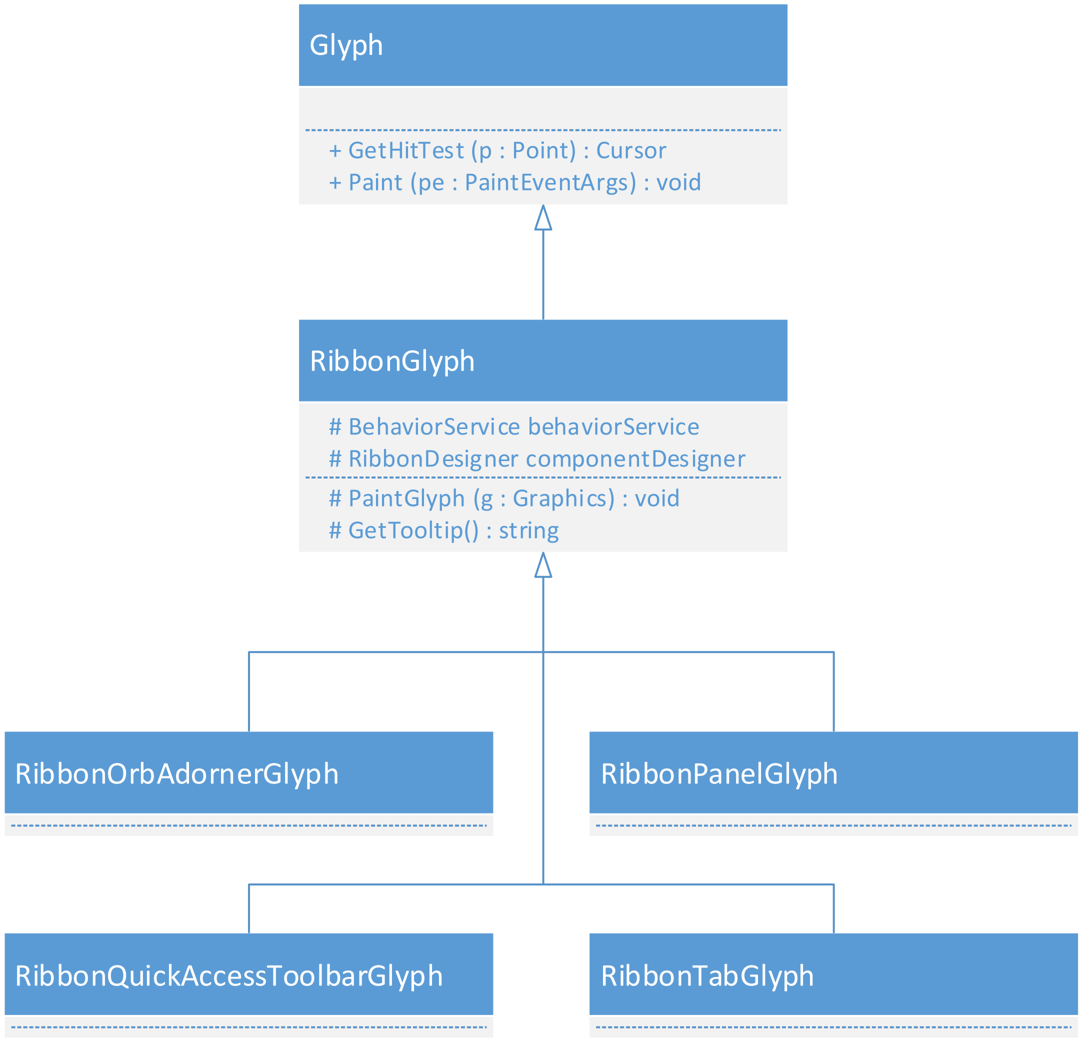
1. Use "Extract Superclass" to have a new parent named for these 4 classes, named `RibbonGlyph` which will be in the middle of the hierarchy, below `Glyph`.
2. Use "Pull Up Field" to move duplicated fields from these 4 classes up to the parent.
3. Use "Pull Up Constructor Body" to create a new constructor in `RibbonGlyph` that will initialize these fields.
4. Use "Pull Up Method" to move the method `GetHitTest` to the parent class, because this method is identical across these 4 classes
5. 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.

6. 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


Glyph
+ GetHitTest (p : Point) : Cursor
+ Paint (pe : PaintEventArgs) : void
RibbonGlyph
# BehaviorService behaviorService
# RibbonDesigner componentDesigner
# PaintGlyph (g : Graphics) : void
# GetTooltip() : string
RibbonOrbAdornerGlyph
RibbonPanelGlyph
RibbonQuickAccessToolbarGlyph
RibbonTabGlyph

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;
       }
    }
}
```

Design Patterns

Observer Pattern – Paterson Deshommes

Reference: http://sourcemaking.com/design_patterns/observer

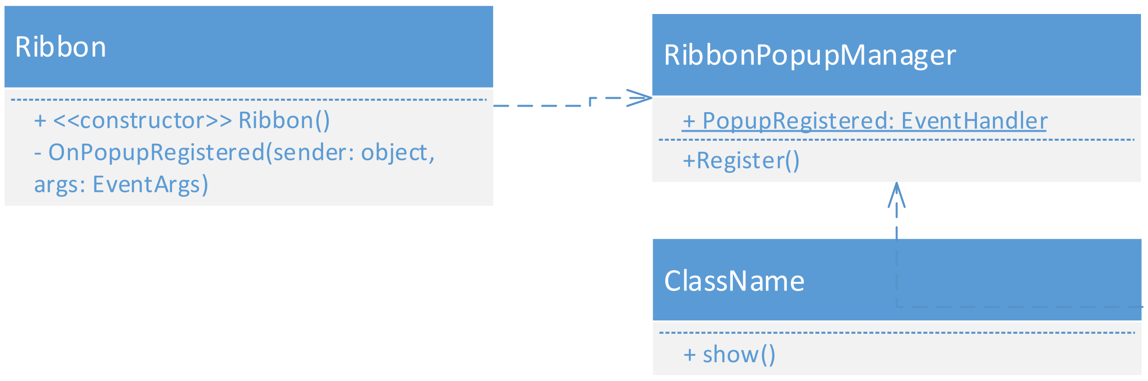
Rationale

Our system has two classes, Ribbon and RibbonPopupManager, which are in a subscriber-publisher relationship. The *Ribbon* class is responsible for creating and providing a ribbon toolbar whereas the RibbonPopupManager class is responsible managing opened popups in the system. The observer pattern is applied to notify the *Ribbon* class when a popup is created in the system so that it can creates the appropriate handlers to receive a system message sent via the mouse or the keyboard.

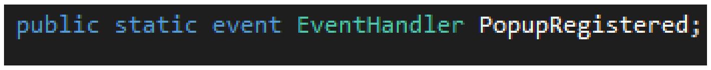
Implementation

The RibbonPopupManager class defines a public, static event of type EventHandler named PopupRegistered. EventHandler is a delegate, which is a type that encapsulates a method like a function pointer in C++. A delegate accepts only methods which have the same signature as the delegate. Multiple methods can be added and removed from a single delegate. An event encapsulates a delegate and provides a public interface for the clients to add and remove handlers while keeping the delegate private to its source class. When the Ribbon class is instantiated, it adds to `PopupRegistered` the method `OnPopupRegistered`. The `RibbonPopupManager` class defines a method called Register which fires the `PopupRegistered` event. The class `RibbonPopup` defines a method named show, which calls the Register method in `RibbonPopupManager`. Then, all the methods of the subscribing classes are executed.

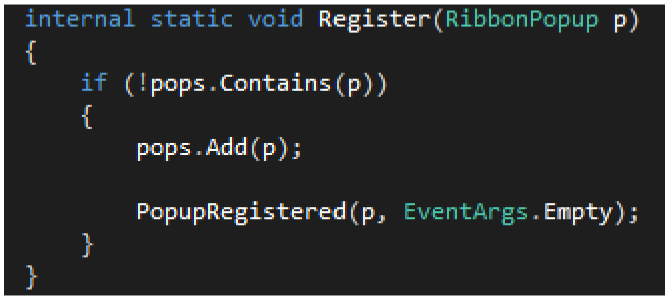
Class diagram


Ribbon
+ <<constructor>> Ribbon()
- OnPopupRegistered(sender: object, args: EventArgs)
RibbonPopupManager
+ PopupRegistered: EventHandler
+Register()
ClassName
+ show()

Observer pattern code in the RibbonPopupManager class

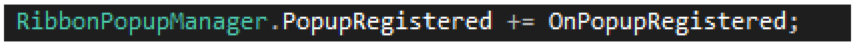


```
public static event EventHandler PopupRegistered;
```

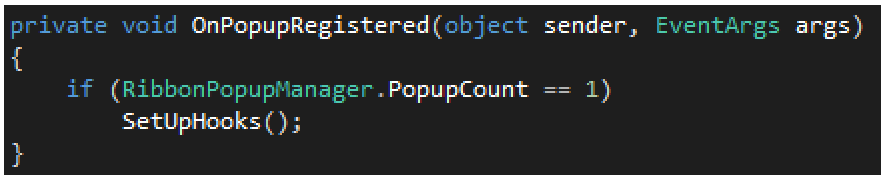


```
internal static void Register(RibbonPopup p)
{
    if (!pops.Contains(p))
    {
        pops.Add(p);

        PopupRegistered(p, EventArgs.Empty);
    }
}
```

Observer pattern code in the Ribbon class

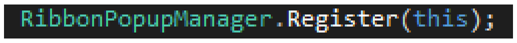


```
RibbonPopupManager.PopupRegistered += OnPopupRegistered;
```



```
private void OnPopupRegistered(object sender, EventArgs args)
{
    if (RibbonPopupManager.PopupCount == 1)
        SetUpHooks();
}
```

Observer pattern code in the RibbonPopup class, in the show method:



```
RibbonPopupManager.Register(this);
```

Composite Pattern – Shahrad Rezaei

We can see an attempt at the composite pattern for drawing UI elements. The Composite pattern, as described in the "Gang of Four" book *Design Patterns*, is used to "compose objects into tree structures to represent part-whole hierarchies". A composite involves a container for other elements, and allows operations to be done recursively across all elements, regardless of its type. The structure of the Composite is outlined in Figure 8

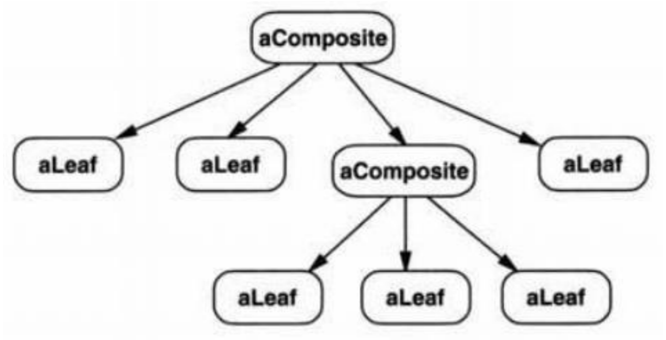


Figure 8: Composite structure

The Office 2007 Ribbon team have created an `IRibbonElement` interface as a shared interface for all drawable elements in the application, as well as several containers. As explained earlier, the Ribbon contains RibbonTabs, which in turn contains RibbonPanels, which contain RibbonItem primitives. Furthermore, each container contains an object inheriting from the .NET List class, which represents a list of the containing objects. This is different from the Composite pattern as described in the *Design Patterns* book, which states that elements should be undiscernible from each other. However, the IRibbonElement interface is not used as much as it should have been, and conditionals are still extensively used throughout the application. The involved classes are shown in Figure 9.

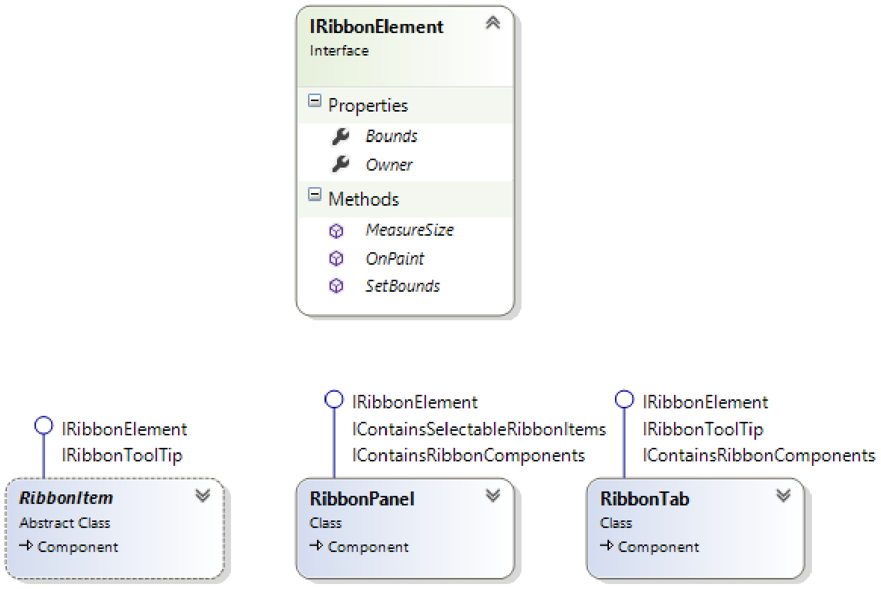


Figure 9: Class diagram for the IRibbonElement interface

If we look more closely at the code, we can see how the team has implemented the pattern. The RibbonTab contains a reference to a RibbonPanelCollection item, which contains a list of all the panels contained within it.

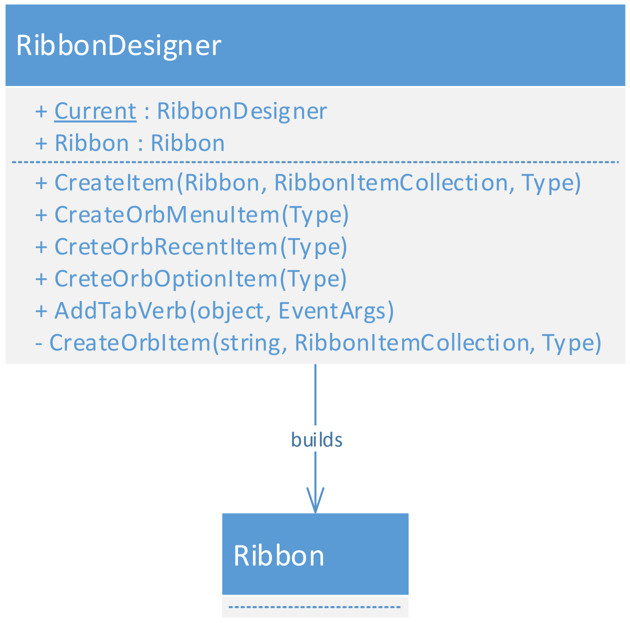
```
public sealed class RibbonPanelCollection
    : List<RibbonPanel>
{
    private RibbonTab _ownerTab;
    public RibbonPanelCollection(RibbonTab ownerTab)
    {
        if (ownerTab == null) throw new ArgumentNullException("ownerTab");

        _ownerTab = ownerTab;
    }

                                                    …
    public new void Add(RibbonPanel item)
    {
        item.SetOwner(Owner);
        item.SetOwnerTab(OwnerTab);
        base.Add(item);
    }

    public new void AddRange(System.Collections.Generic.IEnumerable<System.Windows.Forms.RibbonPanel>
items)
    {
        foreach (RibbonPanel p in items)
        {
            p.SetOwner(Owner);
            p.SetOwnerTab(OwnerTab);
        }

        base.AddRange(items);
    }

    public new void Insert(int index, System.Windows.Forms.RibbonPanel item)
    {
        item.SetOwner(Owner);
        item.SetOwnerTab(OwnerTab);
        base.Insert(index, item);
    }

    internal void SetOwner(Ribbon owner)
    {
        foreach (RibbonPanel panel in this)
        {
            panel.SetOwner(owner);
        }
    }

    internal void SetOwnerTab(RibbonTab ownerTab)
    {
        _ownerTab = ownerTab;

        foreach (RibbonPanel panel in this)
        {
            panel.SetOwnerTab(OwnerTab);
        }
    }
}
```

Builder/Factory Pattern – Felix Soumpholphakdy

The RibbonDesigner class and a few other classes in the designer module have an architecture that resembles both the builder and the factory design pattern. These two design patterns belong to the creational patterns category. Creational design patterns provide design patterns to solve problems related to the instantiation and initialization of objects. The builder design pattern is useful for situations where objects from the same family are created with parts that can vary. More specifically, the builder is responsible for creating only one object type, but its representation and internal parts can differ. On the other hand, the factory design pattern takes care of creating different object types, but still from the same family. Hence, it usually takes only one method call to create an object from a factory, but it usually takes many method calls to build a complete object from a builder.

The RibbonDesigner class does not follow standard implementations of the builder pattern or the factory pattern, but it borrows principles from both of them. First, it has a public Ribbon attribute which represents the product in construction to be assigned by the client. Then, it contains "create" and "add" methods that builds different parts of the Ribbon related to the Orb GUI element. This design of an object responsible for holding a product and creating its composite parts is inspired from the builder pattern. Note that the RibbonDesigner does not fully follow the builder design pattern, because the creation of each part does not vary in implementation, but differ only in their use and order. The similarity with the factory pattern comes from the global access point by the Current public attribute that refers to the most recently instantiated RibbonDesigner and the previously mentioned use of non-polymorphic creation methods.

The use of these principles from creational design patterns like the builder pattern and the factory pattern is required for this system, because it lowers coupling between the Ribbon product object from the client objects and it increases cohesion between objects involved in the building process.


RibbonDesigner
+ Current : RibbonDesigner
+ Ribbon : Ribbon
+ CreateItem(Ribbon, RibbonItemCollection, Type)
+ CreateOrbMenuItem(Type)
+ CreteOrbRecentItem(Type)
+ CreteOrbOptionItem(Type)
+ AddTabVerb(object, EventArgs)
- CreateOrbItem(string, RibbonItemCollection, Type)
builds
Ribbon

```
public class RibbonDesigner : ControlDesigner
{
    public static RibbonDesigner Current;

    public RibbonDesigner()
    {
         Current = this;
    }

    public Ribbon Ribbon
    {
         get { return Control as Ribbon; }
    }

    public virtual void CreateItem(Ribbon ribbon, RibbonItemCollection collection, Type t)
    {
         (…)
    }

    public void CreteOrbMenuItem(Type t)
    {
         CreateOrbItem("MenuItems", Ribbon.OrbDropDown.MenuItems, t);
    }

    public void CreteOrbRecentItem(Type t)
    {
         CreateOrbItem("RecentItems", Ribbon.OrbDropDown.RecentItems, t);
    }

    public void CreteOrbOptionItem(Type t)
    {
         CreateOrbItem("OptionItems", Ribbon.OrbDropDown.OptionItems, t);
    }

    public void AddTabVerb(object sender, EventArgs e)
    {
         (…)
    }

    private void CreateOrbItem(string collectionName, RibbonItemCollection collection, Type t)
    {
         if (Ribbon == null) return;
         (…)
    }

    (…)
}

public class Ribbon : Control
{
     (…)
}
```

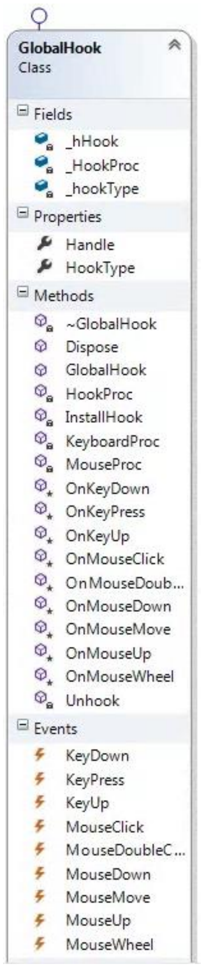
Adaptor/Façade Pattern – Chun Kit Liu

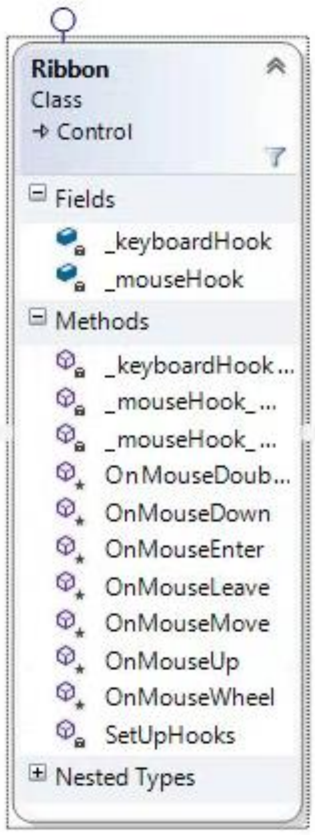
The GlobalHook class is a class that handles mouse and keyboard hooks. Instances of GlobalHook is called by Ribbon class where it is created and assigned to _mouseHook or _keyboardHook private field variables within Ribbon class. It is noted that the GlobalHook class is an actual class and not an interface and does not follow a specific design pattern fully. However, its functionality is very similar to the façade and adaptor patterns.

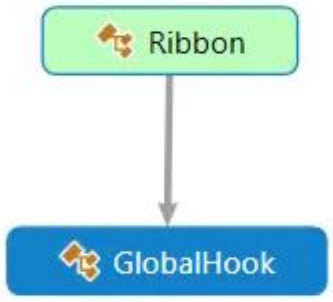
As a class, GlobalHook contains various methods that handle specific events or functions. This includes methods such as OnMouseClick and OnKeyDown. Not only are these methods easy to identify and find, they also do not require varying method calls to achieve certain goals. The idea of these functions being put into one class and having this single class hold these core functions shows a resemblance to how a software library can behave. Of course, as a class, the implementations are not masked. Therefore, it is not a true façade, but undoubtedly, the essence is there. If an interface were to be created which GlobalHook can implement, then the interface will truly become a façade.

In addition to the façade pattern, GlobalHook also borrows some principles from the adaptor design pattern. Once again, it is noted that GlobalHook in itself is an actual class. However, methods such as HookProc, KeyboardProc, MouseProc, InstallHook, and Unhook handle work with WinApi library. In a way, the implementation of calling to WinApi is masked behind these methods much like how adaptors mask the implementation of various tools. If one were to create an adaptor interface to handle WinApi calls, this not only fully implements the adaptor pattern, but would also open up flexibility on using other libraries in the future instead of WinApi.

As such, GlobalHook, though it does not implement fully and adaptor or a façade pattern, it does borrow concepts from both these design patterns. One can say that, with the creation of an adaptor interface and an interface for façade, the end result can become a façade that makes use of adaptors in its implementation.







<u>Façade behaviour can be shown here as in a method:</u>

```
public class GlobalHook : IDisposable
{
    public event MouseEventHandler MouseDoubleClick;
    protected virtual void OnMouseDoubleClick(MouseEventArgs e)
    {
         if (MouseDoubleClick != null)
         {
             MouseDoubleClick(this, e);
         }
    }
}
```

<u>Adaptor behaviour can be shown here as in a method:</u>

```
private int HookProc(int code, IntPtr wParam, IntPtr lParam)
{
     if (code < 0)
```

```
    {
        return WinApi.CallNextHookEx(Handle, code, wParam, lParam);
    }
    else
    {
        switch (HookType)
        {
            case HookTypes.Mouse:
                return MouseProc(code, wParam, lParam);
            case HookTypes.Keyboard:
                return KeyboardProc(code, wParam, lParam);
            default
                throw new Exception("HookType not supported");
        }
    }
}
```

Singleton Pattern – Jignesh Patel

This Singleton pattern was not implemented correctly, however, it can be argued that the developer was trying to implement it.

The declaration is a static instance of the *RibbonDesigner* class. The instance is accessed several times in the software. The instance is created and initialized once in the constructor, and then destroyed by the destructor. This means that during the lifetime of one software run, only the single instance is used. This is consistent with the idea using a global variable, which is what a Singleton tries to accomplish (except that it is an instance of a class, not an attribute).

Instead of using a public accessor method (called *getRibbonDesignerInstance*()) and a *RibbonDesigner* instance that's private, as specified by the specifications, the developer used an instance that is public and static, thus leaving it directly accessible by other classes. The developer also has other code inside of the method. It would have been more cohesive and less coupled to have a class named *RibbonDesignerSingleton* (or something similar), with only the creation logic and the accessor.

The diagrams illustrate the ideal way to have implemented the *RibbonDesigner* instance as a proper Singleton instance. In the Sequence Diagram, the Client can be one of the 10+ classes such as *Ribbon*, *RibbonDropDown* or *RibbonOrbDropDow, etc.* The actual declaration of the static object is shown as well, although it is incorrectly implemented.

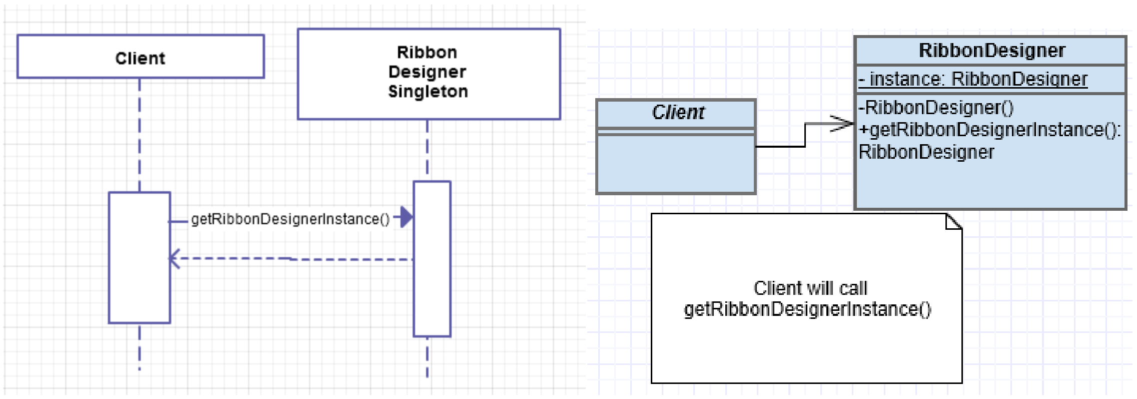

Client
Ribbon Designer Singleton
getRibbonDesignerInstance()
Client
RibbonDesigner
- instance: RibbonDesigner
-RibbonDesigner()
+getRibbonDesignerInstance(): RibbonDesigner
Client will call getRibbonDesignerInstance()

Figure 10 - Sequence Diagram and Class Diagram for Singleton

```
public static RibbonDesigner Current;
#endregion

public RibbonDesigner()
{
   Current = this;
}

~RibbonDesigner()
{
   if (Current == this)
   {
      Current = null;
   }
```

Figure 11 - Declaration of non-ideal Singleton

Refactoring

Patchset 1/5: Create new class RibbonGlyph

The first patch involves creating `RibbonGlyph` class which is an abstract class that inherits from Glyph class. In addition to this change, the classes `RibbonOrbAdornerGlyph`, `RibbonPanelGlyph`, `RibbonQuickAccessToolbarGlyph`, and `RibbonTabGlyph` which inherited from Glyph class now inherits from `RibbonGlyph` class. In order to reduce duplicate code, this new abstract class will hold all appropriate code that its children have in common.

Patchset 2/5: Pull up field behaviorService to RibbonGlyph

Patchset 2 involves removing the instance variable _behaviorService of type BehaviorService from each of `RibbonGlyph's` child classes (RibbonOrbAdornerGlyph, RibbonPanelGlyph, RibbonQuickAccessToolbarGlyph, and RibbonTabGlyph) and putting it into RibbonGlyph itself. The first improvement of this is definitely reducing duplicate code. However, this in turn makes reading each class much simpler as viewers can immediately see the common instance variable from each child class inside RibbonGlyph.

Patchset 3/5: Pull up constructor body to RibbonGlyph

On the third patchset, instead of pulling up a field, part of the constructor is in the child classes RibbonOrbAdornerGlyph, RibbonPanelGlyph, RibbonQuickAccessToolbarGlyph, and RibbonTabGlyph are pulled up into RibbonGlyph abstract class. The important part is only part of the constructor is pulled up to the parent class as each child class contains a slightly different constructor. More specifically, only the setting of _behaviorService instance variable of type BehaviorService is common within the constructor of each child class.

Patchset 4/5: Pull up method GetHitTest to RibbonGlyph

On the fourth patchset, similar to pulling up a field and/or constructor, a common method is pulled up from the child classes RibbonOrbAdornerGlyph, RibbonPanelGlyph, RibbonQuickAccessToolbarGlyph, and RibbonTabGlyph into RibonGlyph abstract class. Method GetHitTest(System.Drawing.Point p) is present in each child class and therefore can be all deleted after moving it into RibbonGlyph class.

Though the four patchsets involve minor changes to the code, as an application's scale and complexity increases, it is important to make code both simple to read and to manage. In the long run, these refactors will definitely show their purposes during both future development and debugging.

Patchset 5/5: Extract method from AddPanel to create setPanelPosition

On the fifth patchset, the conditional logic required to set a panel's position in the AddPanel method is copied into a new method called setPanelPosition. The conditional logic in the AddPanel method is deleted, and replaced with a call to the setPanelPosition method.