Hands-On Lab

Lab Manual for C#

WPF Features Preview – Datagrid, Ribbon, and VSM

Please do not remove this manual from the lab

Author: **Mark Smith**

Publish on Scribd 31/11/2009 by **Ngoc Manh Nguyen**

Contents

Lab 1: What’s coming in WPF

The next version of Windows Presentation Foundation will offer a whole new set of controls for you to utilize in building a state of the art desktop application. In this lab we will examine some of the new features coming in WPF including the DataGrid, Ribbon and Visual State Manager.

You will see these new controls in action by modifying an existing application which uses traditional controls such as the ListView, Menu and ToolBar and modify it to utilize the new features in the WPF preview.

Lab Objective

Estimated time to complete this lab: **1.25 hours (Exercise 1 – 45m, Exercise 2 – 30m)**

The objective of this lab is to take an existing WPF application which displays a checkbook register and convert it to use the new controls available in the WPF Preview.

In this lab, you will have a chance to complete the three following exercises:

Convert a data-bound ListView with custom styles into a data-bound DataGrid utilizing the same styles

Change a menu/toolbar combination into the new Ribbon.

Utilize the new VisualStateManager to provide some animation effects.

Note: code snapshots representing the start-point for each task in the lab are available beneath the Exercise1 and Exercise2 folder. These may be useful if you want to begin the lab somewhere in the middle, or if your project gets into an unrecoverable state.

This lab requires the following components which have all been installed on these machines:

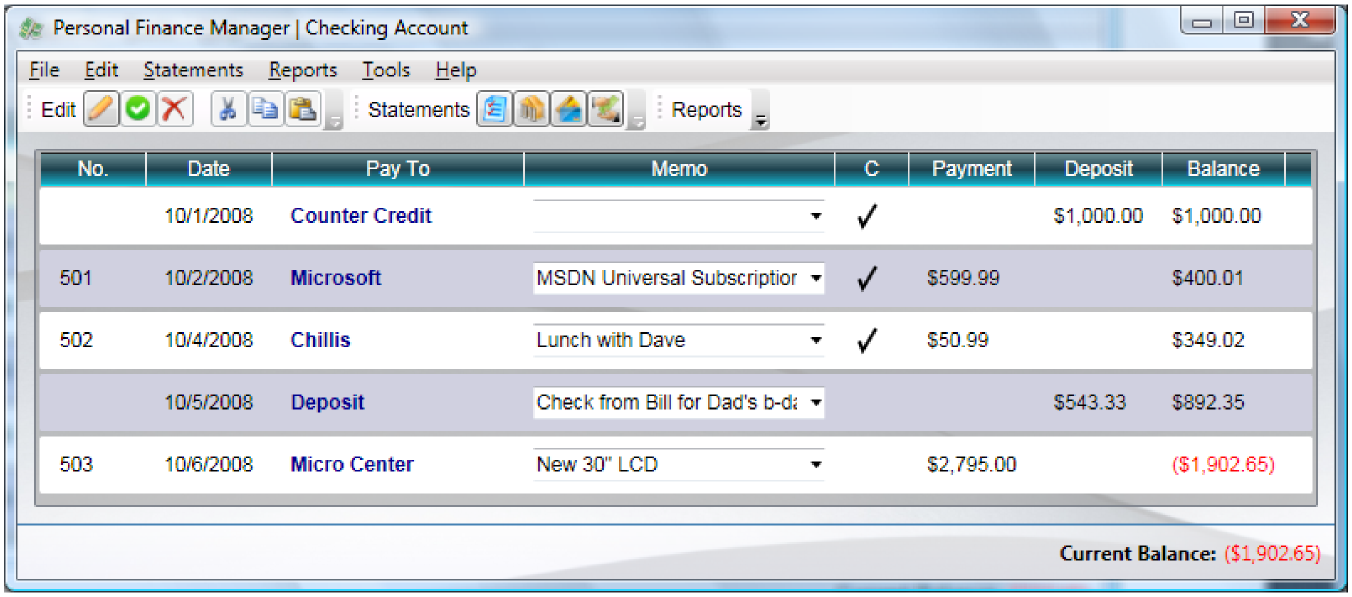
- Microsoft .Net Framework 3.5 Service Pack 1
- Visual Studio 2008 Express (C#) or Visual Studio 2008 Professional Edition
- WPF Futures (included with lab)

Exercise 1 – Using the new WPF DataGrid

In this exercise you will take an existing checkbook application which uses a ListView and modify it to utilize a DataGrid. To accomplish this, the exercise will lead you through several steps:

- Add the DataGrid assembly (WPFToolkit)
- Replace the ListView + GridView with a DataGrid
- Modify the styles and templates to get a similar look and feel.

You will start with a fairly simple WPF application which data binds some checkbook data:



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscriptior	✓	$599.99		$400.01
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$349.02
	10/5/2008	Deposit	Check from Bill for Dad's b-d:			$543.33	$892.35
503	10/6/2008	Micro Center	New 30" LCD		$2,795.00		($1,902.65)

Current Balance: ($1,902.65)

Figure 1: Exercise 1 starter project

This exercise is broken into three tasks – each one has a starter project if you decide you'd rather not run through all the steps you can instead start at the beginning of the next task by opening the appropriate solution file.

Task 1 – Examine the existing application

1. Open source\exercise1\start\CheckbookManager.sln in Visual Studio 2008 with SP1. This is the existing application you will start with.
2. Go ahead and compile the application.
3. Now, run the application to see how it works – you can click on a cell to change the information there. Notice how the balance is adjusted accordingly as you change information.
   a. The menu and toolbar are present but do not currently do anything to keep the sample simplistic.
4. Stop the application and examine the source code. The primary file is **MainWindow.xaml** – this is where the **ListView** is defined, along with the visual styles that affect the colors and columns.
5. Notice that the column definitions required templates – this is because the default **ListView** columns are read-only. We replace them with **TextBox** entries to give us the pseudo-appearance of a data grid.

6. You can examine the data structures that make up this sample if you like – they are in the **Data** directory. It is not necessary to completely understand all of the classes defined there, you will utilize them but not change their functionality.

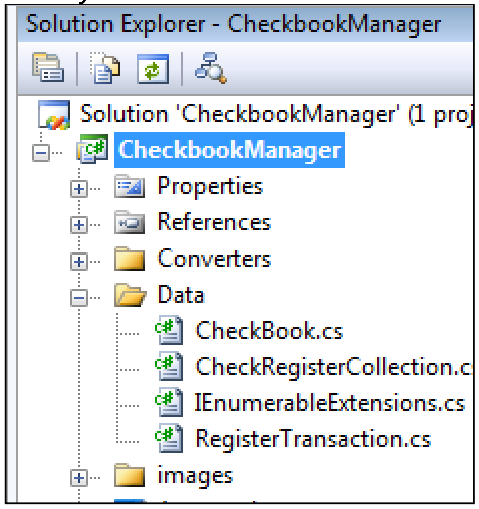


Figure 2: Data classes

a. The **Checkbook.cs** file is the static class that exposes the model to the application.
b. The **CheckRegisterCollection.cs** is an ObservableCollection that holds the transactions
c. The **RegisterTransaction.cs** represents a single entry in the checkbook.

7. Once you are comfortable with the code structure, move onto task 2.

Task 2 – Using the DataGrid

In this task, we will replace the ListView in the application with the new WPF DataGrid.

The first step in using the **DataGrid** is to include the appropriate assembly reference. The DataGrid is contained within the **WPFToolkit** assembly – this is a standalone assembly that implements the grid and a handful of other controls and features new to WPF. There are two ways to get access to this assembly, depending on how you have it installed. Both are covered here, and you can choose either way to proceed with this lab.

1. Make sure the application has been built – this is necessary before you pull up the designer view so the converters are all available.
2. The first mechanism is the manual method – this is used if you have the standalone assembly and did not install the WPF Toolkit package. If you would prefer to use the designer completely, then skip to step 10.
    a. Right click on the **References** folder in the solution and select "Add Reference"

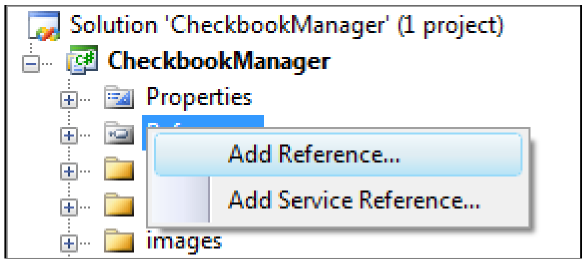
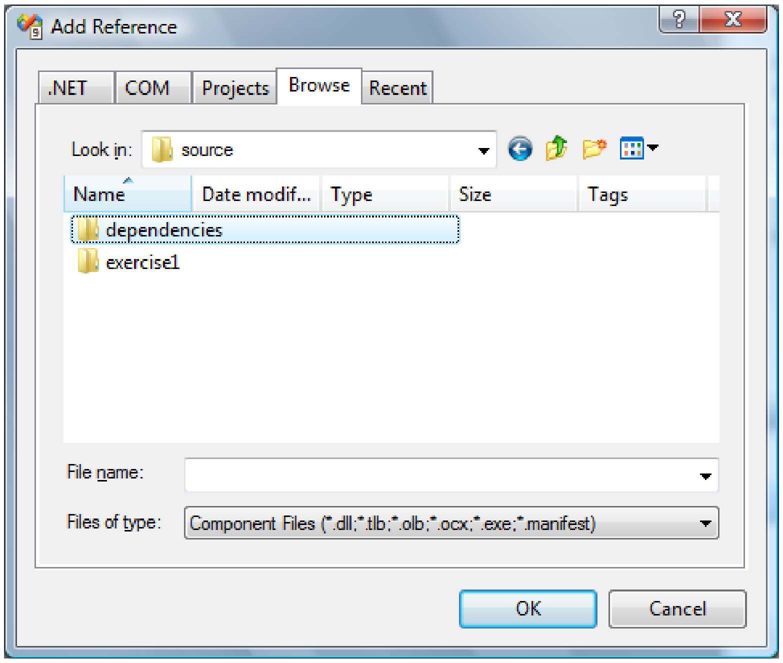


Figure 3: Add Reference

b. Use the Browse Tab and back up to the dependencies folder in the source directory of the lab.



Look in: source

dependencies
exercise1

File name:

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Figure 4: Add Reference - Browse

c. Select the WPFToolkit.dll assembly from that directory and click OK. (Note the data and version number may differ from the picture).

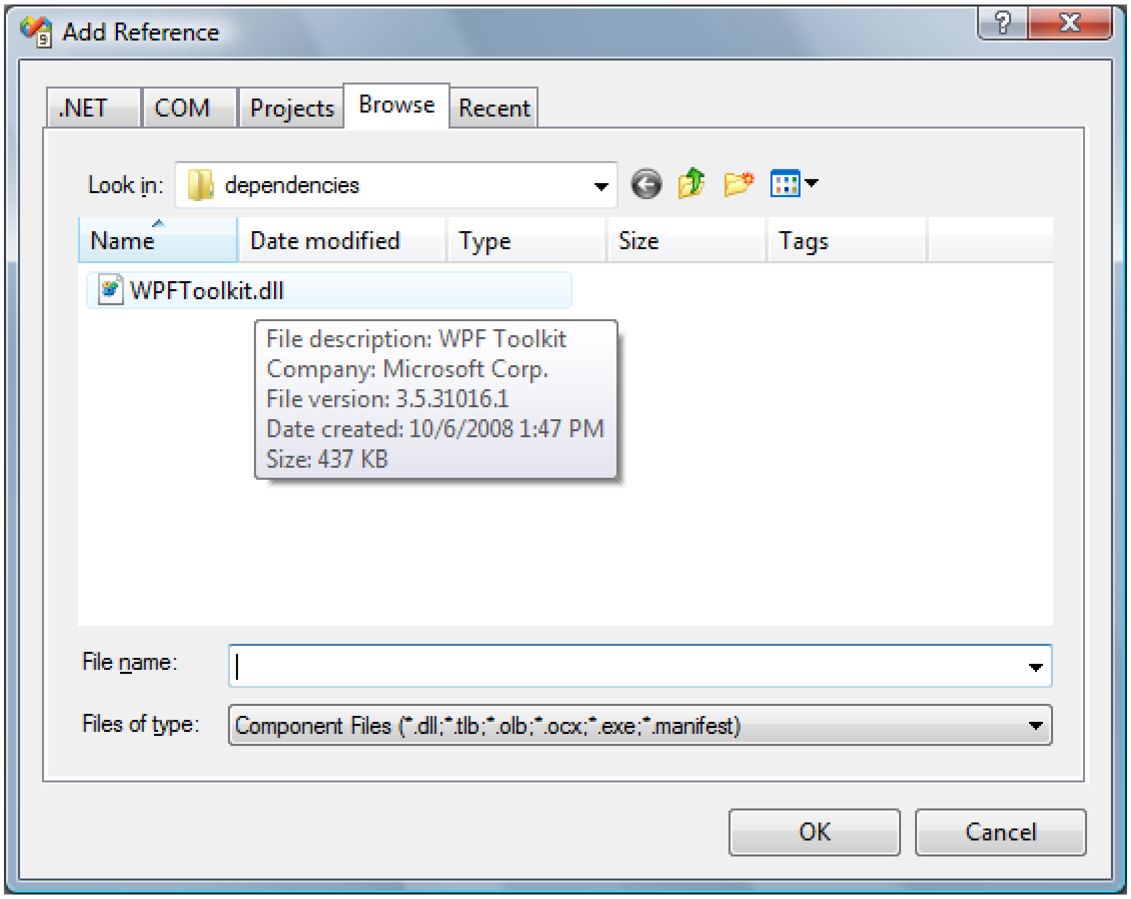


Figure 5: Select WPFToolkit

3. Next, open MainWindow.xaml and add a namespace declaration to the top of the file so we can get to the classes contained within the WPF toolkit.
   a. The namespace is "Microsoft.Windows.Controls".
   b. Map it to some prefix – the lab will use "**dg**".
   c. When you are finished, the line should look like:

```
<Window x:Class="CheckbookManager.MainWindow" ...
    xmlns:dg="clr-namespace:Microsoft.Windows.Controls;assembly=WPFToolkit"
     ...
    Title="Personal Finance Manager | Checking Account" Icon="images/Coins.png">
```

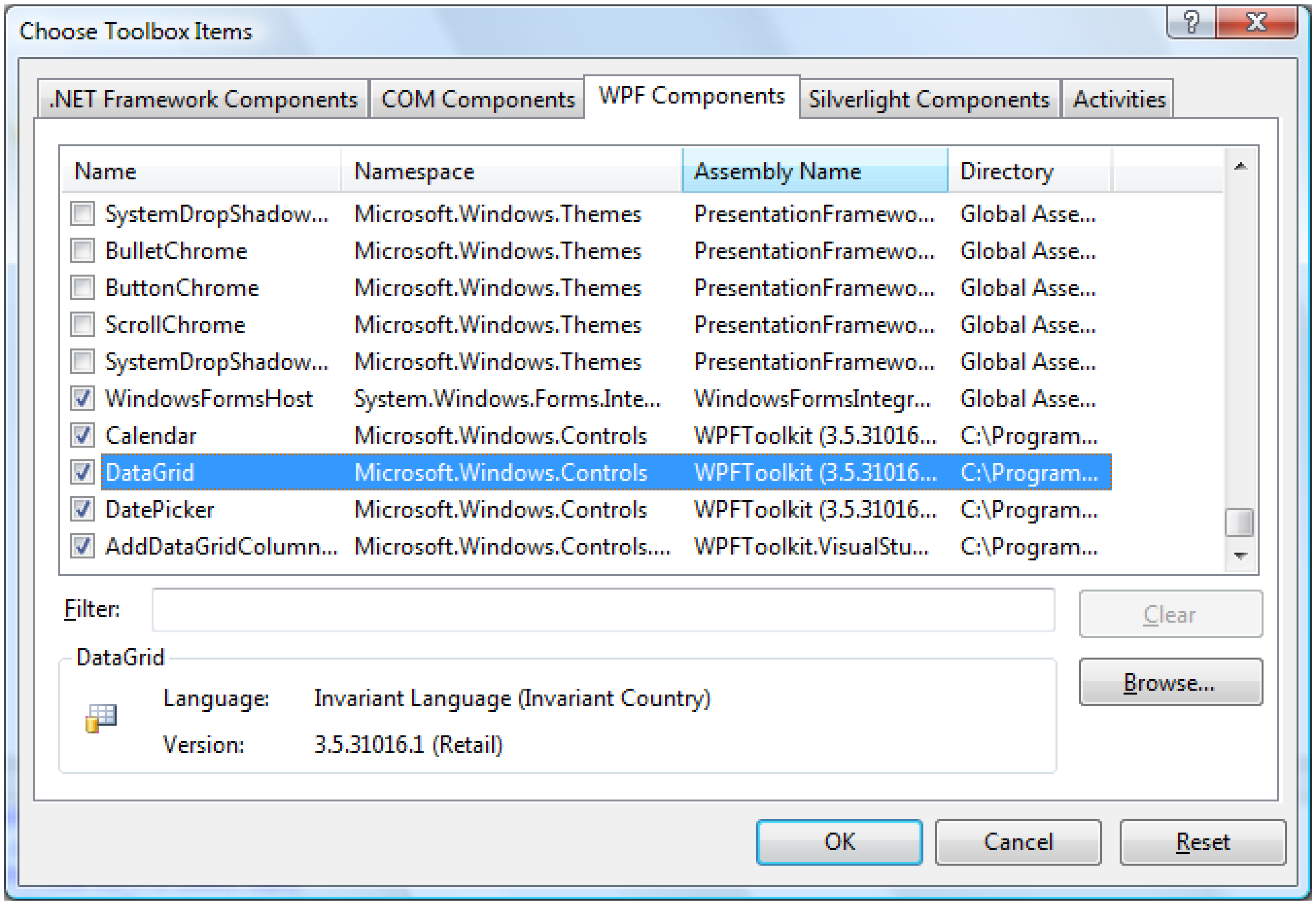
Note: you can alternatively map the namespace "http://schemas.microsoft.com/wpf/2008/toolkit" to the "**dg**" prefix and use that – it registers the appropriate namespaces through an attribute in the WPF toolkit assembly. This is what the designer will use if you drag a DataGrid onto the design surface (starting at step 10 below). Task 2 will direct you to change this later if necessary.

4. Scroll down and locate the ListView in the XAML. This is the control we are going to replace.
5. Change the tag from **ListView** to "**dg:DataGrid**" – this is the new **DataGrid** control.
   a. Note that Intellisense will show a lot of errors at this point as many properties do not map over exactly – we are going to fix that.
6. Remove the ItemContainerStyle property – we'll put it back later, but delete it for now.

7. Delete the TextBox style resource which is in the ListView – we will not need that here since we will be using the new data column types.
8. Replace the ListView prefix the **ContextMenu** property with "**dg:DataGrid**".
9. Comment out the View section – we want the column definitions for later but they are not properly setup now.
10. Your XAML should look like:

```
<!-- DataGrid fills remainder of space -->
<dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
             Background="#80909090" AlternationCount="2">
   <dg:DataGrid.ContextMenu >
      <ContextMenu >
         <MenuItem Header="Copy Selected Transactions"
                   Command="{x:Static ApplicationCommands.Copy}" />
      </ContextMenu>
   </dg:DataGrid.ContextMenu>
   <!-- <ListView.View> ...
        </ListView.View> -->
</dg:DataGrid>
```

11. The second way to add the data grid is through the toolbox. This assumes you have run the WPF Toolkit installer (it should have been run for you on the lab machines). If this is the case, right click on the toolbox in Visual Studio and select "Choose Items". This will display the "Choose Toolbox Items" dialog.
    a. Click on the "WPF Components tab"
    b. Sort the items by the "Assembly Name" column and scroll down to find "WPF Toolkit".
    c. Check the "DataGrid" control and close the dialog.


Choose Toolbox Items

	Name	Namespace	Assembly Name	Directory
☐	SystemDropShadow...	Microsoft.Windows.Themes	PresentationFramewo...	Global Asse...
☐	BulletChrome	Microsoft.Windows.Themes	PresentationFramewo...	Global Asse...
☐	ButtonChrome	Microsoft.Windows.Themes	PresentationFramewo...	Global Asse...
☐	ScrollChrome	Microsoft.Windows.Themes	PresentationFramewo...	Global Asse...
☐	SystemDropShadow...	Microsoft.Windows.Themes	PresentationFramewo...	Global Asse...
✓	WindowsFormsHost	System.Windows.Forms.Inte...	WindowsFormsIntegr...	Global Asse...
✓	Calendar	Microsoft.Windows.Controls	WPFToolkit (3.5.31016...	C:\Program...
✓	DataGrid	Microsoft.Windows.Controls	WPFToolkit (3.5.31016...	C:\Program...
✓	DatePicker	Microsoft.Windows.Controls	WPFToolkit (3.5.31016...	C:\Program...
✓	AddDataGridColumn...	Microsoft.Windows.Controls....	WPFToolkit.VisualStu...	C:\Program...

Filter:

DataGrid

Language: Invariant Language (Invariant Country)

Version: 3.5.31016.1 (Retail)

Figure 6: Add Toolbox Items

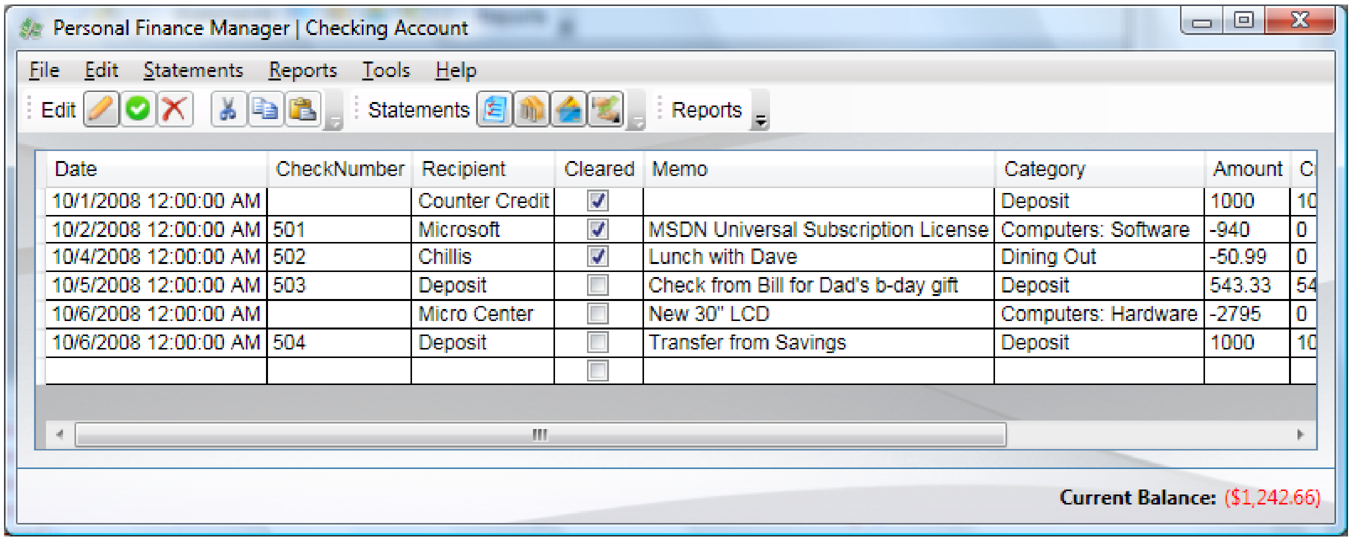
Note: there are other new controls – notably a **Calendar** and **DatePicker** included in the WPF toolkit as well.

12. Now you should have the **DataGrid** control in the toolbox and can drag it onto the design surface directly.

13. Next copy over the **ListView** properties:
    a. Rename the DataGrid to “dg”.
    b. Set the **Background** to “#80909090”
    c. Set the **AlternationCount** to “2”
    d. Set the **Margin** to “2”
    e. Set the **ItemsSource** to “{Binding}” to use the data context as the default binding source.
    f. Add a ContextMenu to the DataGrid – it should have a single **MenuItem** in it which executes the **ApplicationCommands.Copy** command. You can copy it right off the ListView.
    g. Remove the ListView control.
    h. Your XAML should look like this when you are finished (note that the namespace prefix might be different as it is generated by the designer):

```
<!-- DataGrid fills remainder of space -->
<dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
             Background="#80909090" AlternationCount="2">
   <dg:DataGrid.ContextMenu >
      <ContextMenu >
         <MenuItem Header="Copy Selected Transactions"
                   Command="{x:Static ApplicationCommands.Copy}" />
      </ContextMenu>
   </dg:DataGrid.ContextMenu>
</dg:DataGrid>
```

14. Run the application. Notice how the DataGrid automatically populates the columns! This is a feature of the grid – it examines the bound data and generates columns based on the type.



Date	CheckNumber	Recipient	Cleared	Memo	Category	Amount	C
10/1/2008 12:00:00 AM		Counter Credit	✓		Deposit	1000	10
10/2/2008 12:00:00 AM	501	Microsoft	✓	MSDN Universal Subscription License	Computers: Software	-940	0
10/4/2008 12:00:00 AM	502	Chillis	✓	Lunch with Dave	Dining Out	-50.99	0
10/5/2008 12:00:00 AM	503	Deposit	☐	Check from Bill for Dad's b-day gift	Deposit	543.33	54
10/6/2008 12:00:00 AM		Micro Center	☐	New 30" LCD	Computers: Hardware	-2795	0
10/6/2008 12:00:00 AM	504	Deposit	☐	Transfer from Savings	Deposit	1000	10
			☐				

Current Balance: ($1,242.66)

Figure 7: Auto generated columns

15. Click on a cell – notice how it turns into a **TextBox** automatically when it receives focus. This is a styling effect and we’ll see how to change it in a moment. Notice as well that the “Cleared” property which is a Boolean is automatically turned into a **CheckBox**!

16. Change the contents of a cell but press ESC – notice how it changes back automatically.

This is a actually a feature of WPF 3.5 SP1 – the DataGrid examines the underlying objects and if they implement **IEditableObject**, it will call **BeginEdit**, **CommitEdit** and **CancelEdit** on the object automatically. The RegisterTransaction.cs class contains support for this – it copies the original state during the BeginEdit, and either restores it (when canceled) or throws it away when committed. You can use this same technique, or an alternative when making your objects editable through the DataGrid.

17. Close the application and return to the source code.

If you need more control over the column definitions, you can define them directly yourself. Notice that the columns are placed in property-definition order; something we don't want in this case since it changes the display. It turns out that defining columns is easy – the DataGrid has 5 built-in column types which you add to the **Columns** collection to define a column:

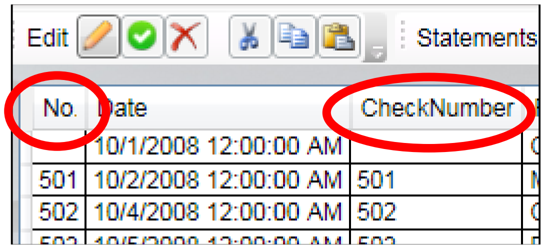
a. **`DataGridCheckBoxColumn`** – displays a **CheckBox** tied to a Boolean
b. **`DataGridComboBoxColumn`** – displays a **ComboBox** of items
c. **`DataGridHyperlinkColumn`** – displays a **Hyperlink** tied to a Uri
d. **`DataGridTextColumn`** – displays a **TextBlock**/**TextBox** tied to a single data type
e. **`DataGridTemplateColumn`** – uses a **DataTemplate** to display contents

The column type defines the type of control which is added to that column, as well as the header, width and other properties. Let's go through the **ListView** columns and convert them to the appropriate column styles starting with the Check Number column.

18. Add a **`<dg:DataGrid.Columns>`** section into the grid.
19. Add a **`<dg:DataGridTextColumn>`** element into the columns collection.
    a. Set the **Header** property to be the text to display in the header ("No.")
    b. Set the **Width** property to be the width you desire. It can be a number (fixed-width), "*" for stretch, "SizeToCells" to size to the largest cell in the column, or "SizeToHeader" to size to the header. In this case, set it to "SizeToCells".
    c. Finally, we can set a Binding expression to indicate where this column gets the data. We assign the binding expression to the **Binding** property. We want to use the same binding expressions assigned to the original ListView columns – for the check number, it's "{Binding CheckNumber}".
    d. When you are finished, it should look like:

```
<dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
            Background="#80909090" AlternationCount="2">

      <dg:DataGrid.Columns>
          <dg:DataGridTextColumn Header="No." Width="SizeToCells"
                                 Binding="{Binding CheckNumber}" />
      </dg:DataGrid.Columns>
```

20. Run the application again – notice how the check number column is now there, but the grid is ***still*** auto-adding the columns.



No.	Date	CheckNumber
	10/1/2008 12:00:00 AM	
501	10/2/2008 12:00:00 AM	501
502	10/4/2008 12:00:00 AM	502

Figure 8: AutoGenerated columns at work

21. We can hook into the column generation process through the **DataGrid.AutoGeneratingColumn** event. This is raised for each column and allows us to examine the column, potentially alter the assumptions made about the column or even cancel the column generation.
22. In this case, however, we want to take control of all the columns – you can tell the grid to stop auto-generating columns by setting the **AutoGenerateColumns** property to "False".

```
        <dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
                  AutoGenerateColumns="False"
                  Background="#80909090" AlternationCount="2">
```

23. Now run the application – notice you only get a single column. Let's add the remainder of the column definitions.
24. Use the bindings from the ListView to create the Date and PayTo columns.
    a. Set the **MinWidth** of the PayTo column to "200" to ensure we always have enough space.

```
            <dg:DataGrid.Columns>
                <dg:DataGridTextColumn Header="No." Width="SizeToCells"
                                       Binding="{Binding CheckNumber}" />
                <dg:DataGridTextColumn Header="Date"
                                       Binding="{Binding Date, StringFormat=d}" />
                <dg:DataGridTextColumn Header="Pay To" MinWidth="200"
                                       Binding="{Binding Recipient}" />
```

So far we've only encountered text box items. The next column – Memo was setup to use a ComboBox in the ListView:

```
        <GridViewColumn Header="Memo" Width="185">
           <GridViewColumn.CellTemplate>
              <DataTemplate>
                 <ComboBox Width="180" IsEditable="True" HorizontalAlignment="Center"
        ItemsSource="{Binding Source={x:Static Data:CheckBook.Descriptions}, Mode=OneWay}"
                                   Text="{Binding Memo}" />
              </DataTemplate>
           </GridViewColumn.CellTemplate>
        </GridViewColumn>
```

We could define a DataTemplate – just like the above GridView did, but in this case it's just as easy to use the DataGridComboBoxColumn here. We can simply move the property values over:

25. Add a **`<dg:DataGridComboBoxColumn>`** to the collection.

a. Set the **Header** to “Memo”
b. Set the **Width** to “*” to indicate that it will consume any remaining space (just like a Grid)
c. Set the **ItemsSource** to the same binding as the ComboBox had in the GridView.
d. Set the **TextBinding** property to the binding associated with ComboBox.Text.

```
    <dg:DataGridComboBoxColumn Header="Memo" Width="*"
        ItemsSource="{Binding Source={x:Static Data:CheckBook.Descriptions}, Mode=OneWay}"
        TextBinding="{Binding Memo}">
    </dg:DataGridComboBoxColumn>
```

The last thing we want to do is set the **IsEditable** property onto the ComboBox to allow users to free-form entry text. This property is not surfaced by the **DataGridComboBoxColumn** however (unlike the ItemsSource and Text). However we can actually impact the generated visual tree through two styles properties exposed on all DataGridColumns: **ElementStyle** and **EditingElementStyle**.

ElementStyle is used as the primary style when the cell does not have focus – for the DataGridTextColumn, this places a TextBlock into the cell. **EditingElementStyle** is swapped in through a trigger when the cell receives input focus. This is how that TextBlock is turned into a TextBox when you click on the cell, or tab into it.

We can use these two properties to change the default properties of the generated visual tree – in this case we want to set the **EditingElementStyle** to a style that sets the **IsEditable** property to “True”:

26. Add an **`<DataGridComboBoxColumn.EditingElementStyle>`** tag and inside that, create a new Style that targets the ComboBox and sets the IsEditable property to true.

```
    <dg:DataGridComboBoxColumn Header="Memo" Width="*"
         ItemsSource="{Binding Source={x:Static Data:CheckBook.Descriptions}, Mode=OneWay}"
         TextBinding="{Binding Memo}">
       <dg:DataGridComboBoxColumn.EditingElementStyle>
          <Style TargetType="ComboBox">
             <Setter Property="IsEditable" Value="True" />
          </Style>
       </dg:DataGridComboBoxColumn.EditingElementStyle>
    </dg:DataGridComboBoxColumn>
```

27. The next column is the “Cleared” column. This is a Boolean and is, by default, mapped to a CheckBox. We want to map it to a **DataGridCheckBoxColumn** but with a different rendering appearance. Looking at the original GridView definition:

```
      <GridViewColumn Header="C">
         <GridViewColumn.CellTemplate>
            <DataTemplate>
               <CheckBox Margin="3"
                         IsChecked="{Binding Cleared}"
                         Style="{StaticResource NoBorderCheckBoxStyle}" />
            </DataTemplate>
         </GridViewColumn.CellTemplate>
    </GridViewColumn>
```

You can see that it is simply a CheckBox with a style applied to it to change the control template. Just like in step 21, we can utilize the **ElementStyle** and **EditingElementStyle** of the column to apply the same style.

28. Add a new **`<dg:DataGridCheckBoxColumn>`** using the binding **{Binding Cleared}** and applying the style to both the ElementStyle and EditingElementStyle (we want it applied in both cases – when the control has focus and when it does not).
    a. Set the Binding property
    b. Set the Header property to “C”
    c. Set the Width to “SizeToHeader” so that it sizes to the header size vs. the CheckBox size.

```
    <dg:DataGridCheckBoxColumn Header="C" Width="SizeToHeader"
         Binding="{Binding Cleared}"
         ElementStyle="{DynamicResource NoBorderCheckBoxStyle}"
         EditingElementStyle="{DynamicResource NoBorderCheckBoxStyle}" />
```

29. Add the Payment and Deposit columns – they should use the **`<dg:DataGridTextColumn>`**
    a. Use the bindings from the original ListView columns (including the converter).
    b. Set the **Width** to “SizeToCells”

```
  <dg:DataGridTextColumn Width="SizeToCells" Header="Payment"
        Binding="{Binding Amount, Converter={StaticResource amountConverter},
                  ConverterParameter=0, StringFormat=C}" />
  <dg:DataGridTextColumn Width="SizeToCells" Header="Deposit"
        Binding="{Binding Amount, Converter={StaticResource amountConverter},
                  ConverterParameter=1, StringFormat=C}" />
```

Finally, let’s add the last column – the balance.

In this case we want a read-only text column – we could achieve that through the **IsReadOnly** property on the DataGridColumn, but in this case we also want the color of the text to change based on the value. This is not achievable with a trigger since it’s not a specific value, here we need to do some data binding to the Foreground property and that’s going to require that we take over the visual tree for the column.

To do that, we can use the **DataGridTemplateColumn** which allows us to supply a DataTemplate to be used for each cell. We assign the template to the CellTemplate property and it turns out that we already have the template – the GridView defined it for us:

```
     <DataTemplate>
         <TextBlock Text="{Binding TotalBalance, StringFormat=C}"
     Foreground="{Binding TotalBalance,Converter={StaticResource BalanceDisplayConverter}}"
                    />
     </DataTemplate>
```

30. Add the **`<dg:DataGridTemplateColumn>`** into the column definitions
    a. Set the Width to “SizeToCells”
    b. Set the Header to “Balance”

c. Add a “ClipboardContentBinding” property and set it to the same binding used in the TextBox. This is what the column will use if the contents are copied to the clipboard. Without this tag, this column would be ignored.

d. Finally, add a DataGridTemplateColumn.CellTemplate property and assign it to the DataTemplate. Your completed column should look like:

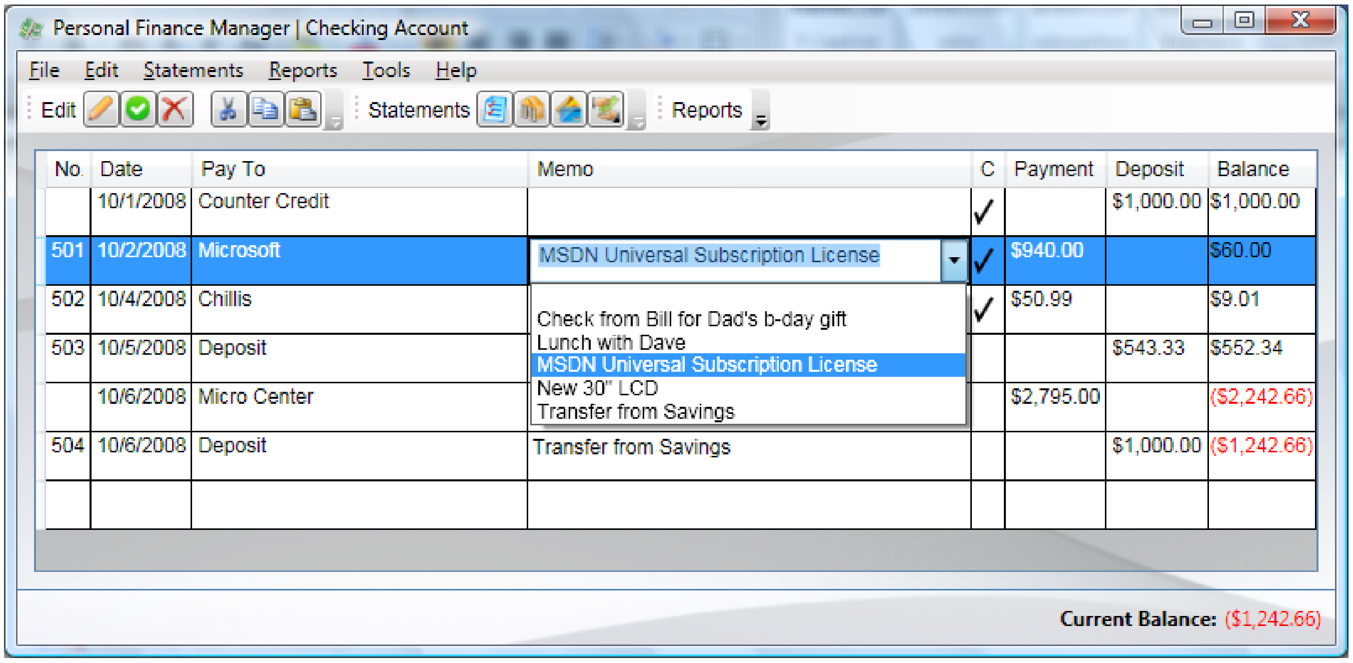
```
  <dg:DataGridTemplateColumn Width="SizeToCells" Header="Balance"
                             ClipboardContentBinding="{Binding TotalBalance}">
     <dg:DataGridTemplateColumn.CellTemplate>
         <DataTemplate>
             <TextBlock Foreground="{Binding TotalBalance,Converter={StaticResource
BalanceDisplayConverter}}" Text="{Binding TotalBalance, StringFormat=C}" />
         </DataTemplate>
     </dg:DataGridTemplateColumn.CellTemplate>
  </dg:DataGridTemplateColumn>
```

To verify your work, the completed column definitions should look like:

```
<dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
                 AutoGenerateColumns="False"
                 Background="#80909090" AlternationCount="2">
         ...
   <dg:DataGrid.Columns>
      <dg:DataGridTextColumn Header="No." Width="SizeToCells"
                             Binding="{Binding CheckNumber}" />
      <dg:DataGridTextColumn Header="Date" Binding="{Binding Date, StringFormat=d}" />
      <dg:DataGridTextColumn Header="Pay To" MinWidth="200"
                             Binding="{Binding Recipient}" />

      <dg:DataGridComboBoxColumn Header="Memo" Width="*"
                ItemsSource="{Binding Source={x:Static Data:CheckBook.Descriptions},
                               Mode=OneWay}"
                TextBinding="{Binding Memo}">
         <dg:DataGridComboBoxColumn.EditingElementStyle>
            <Style TargetType="ComboBox">
               <Setter Property="IsEditable" Value="True" />
            </Style>
         </dg:DataGridComboBoxColumn.EditingElementStyle>
      </dg:DataGridComboBoxColumn>

      <dg:DataGridCheckBoxColumn Header="C" Width="SizeToHeader"
           Binding="{Binding Cleared}"
           ElementStyle="{DynamicResource NoBorderCheckBoxStyle}"
           EditingElementStyle="{DynamicResource NoBorderCheckBoxStyle}" />

      <dg:DataGridTextColumn Width="SizeToCells" Header="Payment"
           Binding="{Binding Amount, Converter={StaticResource amountConverter},
                     ConverterParameter=0, StringFormat=C}" />
      <dg:DataGridTextColumn Width="SizeToCells" Header="Deposit"
          Binding="{Binding Amount, Converter={StaticResource amountConverter},
                    ConverterParameter=1, StringFormat=C}" />

      <dg:DataGridTemplateColumn Width="SizeToCells" Header="Balance"
```

```
                      ClipboardContentBinding="{Binding TotalBalance}">
         <dg:DataGridTemplateColumn.CellTemplate>
            <DataTemplate>
               <TextBlock Foreground="{Binding TotalBalance,
                       Converter={StaticResource BalanceDisplayConverter}}"
                         Text="{Binding TotalBalance, StringFormat=C}" />
            </DataTemplate>
         </dg:DataGridTemplateColumn.CellTemplate>
      </dg:DataGridTemplateColumn>
   </dg:DataGrid.Columns>
</dg:DataGrid>
```

31. Run the application. It should look much closer to the original – now with a drop down box, styled checkbox and colored balances.



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00
502	10/4/2008	Chillis		✓	$50.99		$9.01
503	10/5/2008	Deposit				$543.33	$552.34
	10/6/2008	Micro Center			$2,795.00		($2,242.66)
504	10/6/2008	Deposit	Transfer from Savings			$1,000.00	($1,242.66)

Check from Bill for Dad's b-day gift
Lunch with Dave
MSDN Universal Subscription License
New 30" LCD
Transfer from Savings

Current Balance: ($1,242.66)

Figure 9: Defining Columns manually

32. One thing that is missing is the "PayTo" column was blue + bold in the previous version. How could you add that support? See if you can make that change.
    a. Hint: there are two ways to do it – we've seen both in the above steps!

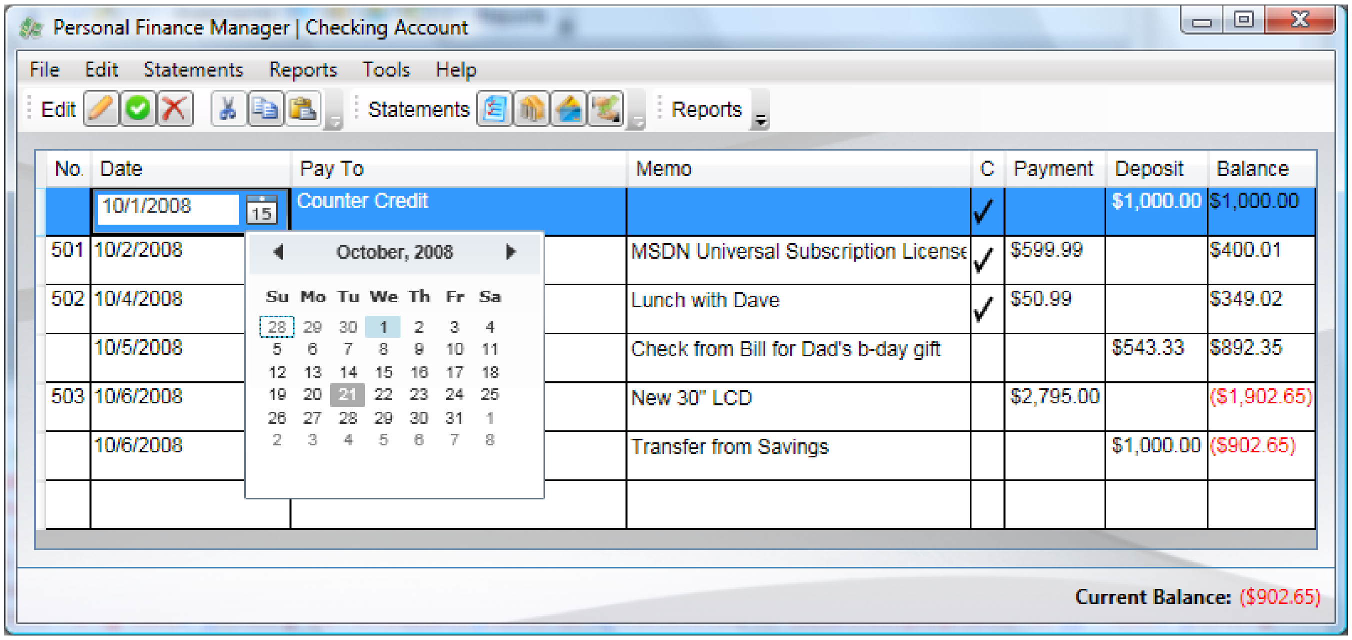
If you are ready to move on, the next task will style the data grid to make it visually appealing; however if you have time the next few steps will utilize the new **Calendar** and **DatePicker** control to show a calendar in the Date field. You can skip this step and move to Task 3 and just examine the starter project if you like.

33. Open the XAML for the **DataGrid** and locate the **DataGridTextColumn** used to map the date. Replace that with a **DataGridTemplateColumn** (just like you did for the balance column above). Use the same header, but set the minimum width to a fixed "100" pixel width (just specify "100" as the **MinWidth** property).

34. Inside the template column, we want to specify both the editing template and the normal (non-editing) template.
35. For the **CellEditingTemplate**:
    a. Create a new DataTemplate and add a **DatePicker** from the WPF Toolkit to it. It is in the same namespace as the DataGrid.
    b. Bind the **SelectedDate** property to the check's Date property.
    c. Set the **SelectedDateFormat** property to "Short".
36. For the **CellTemplate** property use a simple TextBlock bound to the check's Date property. Set the binding **StringFormat** to "d" for DateTime format.
37. The XAML should look something like:

```
<!--<dg:DataGridTextColumn Header="Date" Binding="{Binding Date, StringFormat=d}" />-->
<dg:DataGridTemplateColumn Header="Date" MinWidth="100">
   <dg:DataGridTemplateColumn.CellEditingTemplate>
      <DataTemplate>
         <dg:DatePicker SelectedDate="{Binding Date}" SelectedDateFormat="Short" />
      </DataTemplate>
   </dg:DataGridTemplateColumn.CellEditingTemplate>
   <dg:DataGridTemplateColumn.CellTemplate>
      <DataTemplate>
         <TextBlock Text="{Binding Date, StringFormat=d}" />
      </DataTemplate>
   </dg:DataGridTemplateColumn.CellTemplate>
</dg:DataGridTemplateColumn>
```

38. Run the application – it will now use a **DatePicker** with calendar drop-down when the cell is in edit mode. Otherwise it will use a TextBlock.


Personal Finance Manager | Checking Account
File Edit Statements Reports Tools Help
Edit
Statements
Reports
October, 2008
Su Mo Tu We Th Fr Sa

No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008		MSDN Universal Subscription License	✓	$599.99		$400.01
502	10/4/2008		Lunch with Dave	✓	$50.99		$349.02
	10/5/2008		Check from Bill for Dad's b-day gift			$543.33	$892.35
503	10/6/2008		New 30" LCD		$2,795.00		($1,902.65)
	10/6/2008		Transfer from Savings			$1,000.00	($902.65)

Current Balance: ($902.65)

Figure 10: Using the DatePicker control

Task 3 – Styling the DataGrid

Recall the original look of the application – it had styled headers and better colors. Our new grid is more functional, but is pretty plain and dry for our commercial application. This task will apply various styles to the DataGrid to make it look more colorful and visually appealing.

Like any other WPF based control, the DataGrid is completely styleable. You can apply styles to almost any portion of the UI and if necessary, you can even replace the control templates with your own for a radically different look and feel.

1. Open source\exercise1\task3\CheckbookManager.sln in Visual Studio 2008 with SP1. This is the starting point. Note that if you completed Task 1, you can continue from that point as well.
2. Compile the application to ensure all the value converters are available.
3. Open MainWindow.xaml and scroll to the root **DockPanel** resources. There you should see some styles that were left over from the original GridView implementation. Specifically, we've got a couple of brushes and a **GridViewColumnHeader** and **ListViewItem** style. Our goal is to apply these styles to the DataGrid.

The DataGrid has several Style properties to adjust its look:

a) **CellStyle** – used to style each individual cell (**DataGridCell**)
b) **RowStyle** – used to style each row (**DataGridRow**)
c) **ColumnHeaderStyle** – used to style the header bar (**DataGridColumnHeader**)

In addition, it has a bunch of exposed properties that change how the user can interact with the grid:

a) **SelectionMode** – to switch between single and extended selection
b) **SelectionUnit** – to change what can be selected (FullRow vs. Cell vs. CellOrRowHeader)
c) **GridLinesVisibility** – to change what lines are drawn around cells
d) **VerticalGridLinesBrush** – to adjust the brush colors for vertical lines
e) **HorizontalGridLinesBrush** – to adjust the brush colors for horizontal lines

First, let's adjust the TargetType on the existing styles – we can use all of these properties directly since almost all of them are coming from the common base class ItemsControl (ListView and DataGrid both extend it).

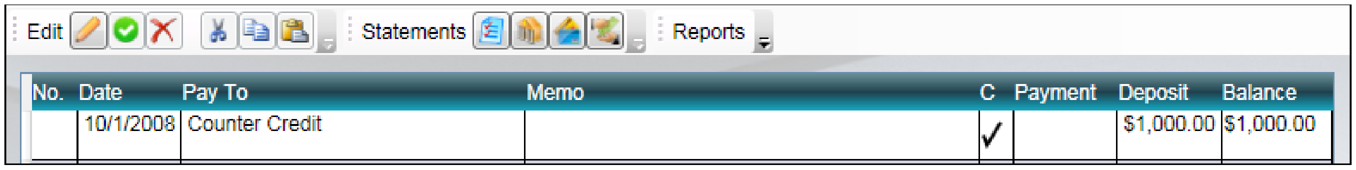
4. Locate the dgHeaderStyle style. Change the TargetType from **GridViewColumnHeader** to **dg:DataGridColumnHeader.**
5. Locate the dgRowStyle style and change the TargetType from **ListViewItem** to **dg:DataGridRow**.
6. You should now have:

```
    <Style x:Key="dgHeaderStyle" TargetType="dg:DataGridColumnHeader">
       <Setter Property="Background" Value="{StaticResource dgHeaderBrush}" />
       <Setter Property="Foreground" Value="White" />
       <Setter Property="BorderBrush" Value="{StaticResource dgHeaderBorderBrush}" />
    </Style>
    <Style x:Key="dgRowStyle" TargetType="dg:DataGridRow">
       <Setter Property="SnapsToDevicePixels" Value="True" />
       <Setter Property="Background" Value="White" />
       <Style.Triggers>
          <Trigger Property="ItemsControl.AlternationIndex" Value="1">
             <Setter Property="Background" Value="#FFD0D0E0" />
          </Trigger>
          <Trigger Property="IsSelected" Value="True">
             <Setter Property="Background" Value="LightGoldenrodYellow" />
          </Trigger>
       </Style.Triggers>
    </Style>
```

7. Scroll down to the DataGrid definition and apply these two styles to the ColumnHeaderStyle and RowStyle properties:

```
         <dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
                   AutoGenerateColumns="False"
                   Background="#80909090" AlternationCount="2"
                   ColumnHeaderStyle="{StaticResource dgHeaderStyle}"
                   RowStyle="{StaticResource dgRowStyle}">
```

8. Run the application and notice the difference that applying these styles has made, notice the header is now a nice shaded gradient and that our alternating colors are applied to the rows.



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00

Figure 11: Applying row and header styles

9. Now, let’s add to that by adding a couple more properties to the grid
   a. Turn on “Extended” selection mode
   b. Set the SelectionUnit to “FullRow”
   c. Set the GridLinesVisibility to “All”
   d. Set the VerticalGridLinesBrush to “DarkGray”.

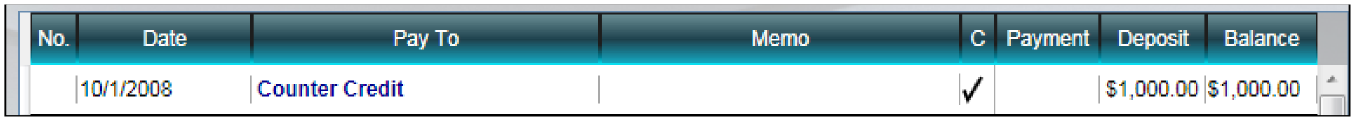
```
         <dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
                   AutoGenerateColumns="False"
                   Background="#80909090" AlternationCount="2"
                   ColumnHeaderStyle="{StaticResource dgHeaderStyle}"
                   RowStyle="{StaticResource dgRowStyle}"
                   SelectionMode="Extended"
                   SelectionUnit="FullRow"
                   GridLinesVisibility="All"
                   VerticalGridLinesBrush="DarkGray">
```

10. Lets also apply some additional settings in the header style, specifically
    a. Set the BorderThickness to “1” so we can see the border
    b. Turn on pixel snapping (SnapsToDevicePixels=”True”)
    c. Align the content horizontally (HorizontalContentAlignment=”Center”)
    d. Apply a MinWidth/MinHeight of “0” and “30”
    e. Set the default Cursor to be the Hand.

```
   <Style x:Key="dgHeaderStyle" TargetType="dg:DataGridColumnHeader">
      <Setter Property="Background" Value="{StaticResource dgHeaderBrush}" />
```

```
    <Setter Property="Foreground" Value="White" />
    <Setter Property="BorderBrush" Value="{StaticResource dgHeaderBorderBrush}" />
    <Setter Property="BorderThickness" Value="1" />
    <Setter Property="SnapsToDevicePixels" Value="True" />
    <Setter Property="HorizontalContentAlignment" Value="Center" />
    <Setter Property="MinWidth" Value="0" />
    <Setter Property="MinHeight" Value="30" />
    <Setter Property="Cursor" Value="Hand" />
</Style>
```

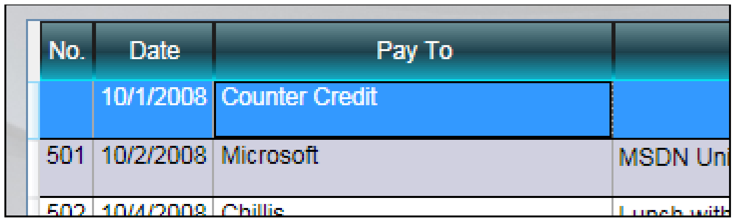
Now the application has a nice glowing border around the header and if you move the cursor onto the header you will see it switch to a handle indicating sorting capabilities.



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00

Figure 12: Final header

11. The other style we can apply is to the cells themselves. It's convenient to apply a global style to cells to do things like center the content, or apply triggers when the cell has focus. Let's try that out, notice that selection isn't very pretty right now and that the data in the columns is aligned to the top:



No.	Date	Pay To	
	10/1/2008	Counter Credit	
501	10/2/2008	Microsoft	MSDN Uni
502	10/4/2008	Chillis	Lunch with

Figure 13: Cell Selection

We can fix that by applying a default cell style.

12. Add a new Style into the DockPanel.Resources – right below the current row Style is fine.
    a. Set the TargetType to be **dg:DataGridCell**.

Note: this assumes you are have assigned the "dg" namespace to "http://schemas.microsoft.com/wpf/2008/toolkit". If instead you are using the assembly name in your XAML namespace definition as shown in Task 1, you will either need to update your namespace definition or define a second prefix for **`System.Windows.Controls.Primitives`** which is where the **`DataGridCell`** type is located.

    b. Since we want to keep the majority of the visual appearance, base this style on the default DataGridCell style by setting the **BasedOn** property to "{StaticResource {x:Type dg:DataGridCell}}".
    c. Set SnapsToDevicePixels to "True"
    d. Set VerticalAlignment to "Center"
    e. Add a Triggers collection – this is where we can change the background/foreground colors when the cell is selected.

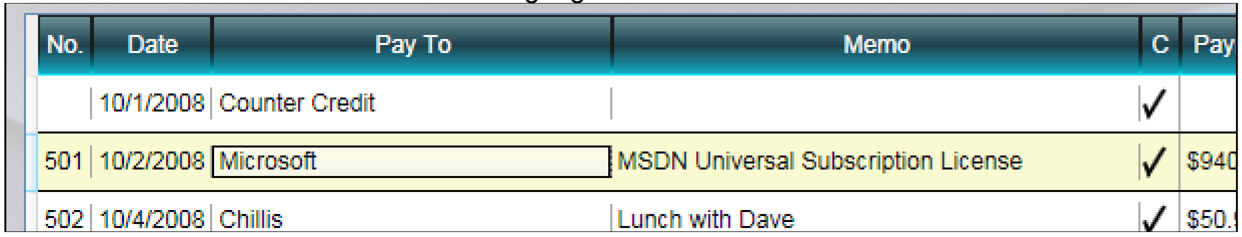
f. In the **`<Style.Triggers>`** section, add a new trigger based on the **IsSelected** property and when it is True:
   i. Set the **Background** to Transparent
   ii. Set the **BorderBrush** to Transparent
   iii. Set the **Foreground** to Black
g. Add a second trigger on the IsKeyboardFocusWithin = “True” property to:
   i. Set the **Background** to the “{StaticResource whiteBackBrush}” resource
   ii. Set the **BorderBrush** to the {DynamicResource {x:Static dg:DataGrid.FocusBorderBrushKey}}” which is the default focus brush
   iii. Set the **Foreground** to Black
h. When you are done, it should look like:

```
    <Style x:Key="dgCellStyle" TargetType="dg:DataGridCell"
           BasedOn="{StaticResource {x:Type dg:DataGridCell}}">
       <Setter Property="SnapsToDevicePixels" Value="True" />
       <Setter Property="VerticalAlignment" Value="Center" />
       <Style.Triggers>
          <Trigger Property="IsSelected" Value="True">
             <Setter Property="Background" Value="Transparent" />
             <Setter Property="BorderBrush" Value="Transparent" />
             <Setter Property="Foreground" Value="Black" />
          </Trigger>
          <Trigger Property="IsKeyboardFocusWithin" Value="True">
             <Setter Property="Background" Value="{StaticResource whiteBackBrush}" />
             <Setter Property="BorderBrush"
                  Value="{DynamicResource {x:Static dg:DataGrid.FocusBorderBrushKey}}" />
             <Setter Property="Foreground" Value="Black" />
          </Trigger>
       </Style.Triggers>
    </Style>
```

13. Apply the resource to the grid’s **CellStyle** property and run the application.

```
        <dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10"
                  AutoGenerateColumns="False"
                  Background="#80909090" AlternationCount="2"
                  ColumnHeaderStyle="{StaticResource dgHeaderStyle}"
                  RowStyle="{StaticResource dgRowStyle}"
                  CellStyle="{StaticResource dgCellStyle}"
                  SelectionMode="Extended"
                  SelectionUnit="FullRow"
                  GridLinesVisibility="All"
                  VerticalGridLinesBrush="DarkGray">
```

14. Click on a cell – notice how it now highlights with a nicer brush color:



No.	Date	Pay To	Memo	C	Pay
	10/1/2008	Counter Credit		✓	
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.

Figure 14: Final cell style

Another thing you can add to the DataGrid which is handy is a **RowDetail** section. This is placed below the row and can be displayed when the row is selected, always or never. You turn it on through the **RowDetailsVisibilityMode** property and then tell the grid how to render the details through a **DataTemplate** assigned to the **RowDetailsTemplate**. The visibility mode can be set to "Visible", "VisibleWhenSelected" or "Collapsed". This can also be adjusted at runtime, or with triggers for a more dynamic setting but for now, let's use it to display the charge category.

15. We are actually missing one piece of information from the RegisterTransaction data class – the Category property. The code behind already supports the property – we just need to add it to the UI and the RowDetails is a great place to do it.
    a. Set the **RowDetailsVisibilityMode** to "VisibleWhenSelected"
    b. Add a new **DataTemplate** and apply it to the **RowDetails** property – just add a TextBlock and TextBox wrapped in a horizontal StackPanel.
    c. Place a Left Margin of 20 pixels on the StackPanel to offset the data:

```
    <dg:DataGrid x:Name="dg" ItemsSource="{Binding}" Margin="10" ...
        VerticalGridLinesBrush="DarkGray"
        RowDetailsVisibilityMode="VisibleWhenSelected">

            <dg:DataGrid.RowDetailsTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal" Margin="20,0,0,0">
                        <TextBlock />
                        <TextBox />
                    </StackPanel>
                </DataTemplate>
            </dg:DataGrid.RowDetailsTemplate>
```

    d. Set the TextBlock's Text property to "Category:" and vertically align it in the center. While you are editing that element, set the font weight to "Bold".

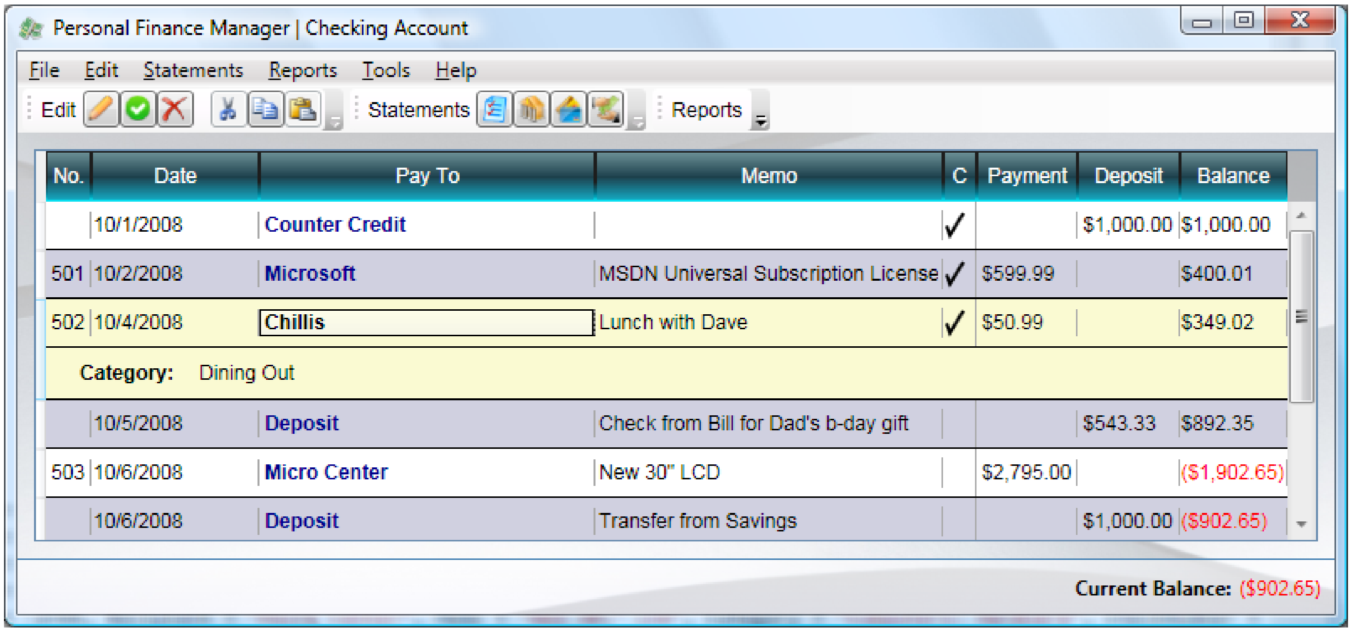
```
    <StackPanel Orientation="Horizontal" Margin="20,0,0,0">
       <TextBlock Text="Category:" VerticalAlignment="Center" FontWeight="Bold" />
       <TextBox />
```

    e. Now turn to the **TextBox** – bind the **Text** property to the Category property and set the Margin to "10,5" and the minimum width of the TextBox to 100.
    f. Finally, let's add a style to the TextBox to make it look more like the controls already present in the data grid. Specifically, we want to remove the border and background color until either the mouse is over the TextBox, or the TextBox has focus. You can add the appropriate Style + Triggers or use the following code:

```
       <dg:DataGrid.RowDetailsTemplate>
          <DataTemplate>
             <StackPanel Orientation="Horizontal" Margin="20,0,0,0">
                <TextBlock Text="Category:" VerticalAlignment="Center" FontWeight="Bold" />
                <TextBox Text="{Binding Category}" Margin="10,5" MinWidth="100">
                   <TextBox.Style>
                      <Style TargetType="TextBox">
                         <Setter Property="BorderBrush" Value="{x:Null}" />
```

```
                       <Setter Property="Background" Value="{x:Null}" />
                       <Style.Triggers>
                          <Trigger Property="IsFocused" Value="True">
                             <Setter Property="BorderBrush"
                                     Value="{x:Static SystemColors.WindowFrameBrush}" />
                             <Setter Property="Background"
                                     Value="{x:Static SystemColors.WindowBrush}" />
                          </Trigger>
                          <Trigger Property="IsMouseOver" Value="True">
                             <Setter Property="BorderBrush"
                                     Value="{x:Static SystemColors.WindowFrameBrush}" />
                             <Setter Property="Background"
                                     Value="{x:Static SystemColors.WindowBrush}" />
                          </Trigger>
                       </Style.Triggers>
                    </Style>
                 </TextBox.Style>
              </TextBox>
           </StackPanel>
        </DataTemplate>
           </dg:DataGrid.RowDetailsTemplate>
```

16. Run the application again and select a row. See how the details are supplied onto the row line dynamically? Note as well how the **TextBox** blends into the background until you hover or click on it. This is the effect of the style we have used to supply different property values through property triggers.


Personal Finance Manager | Checking Account
File Edit Statements Reports Tools Help
Edit
Statements
Reports

No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$599.99		$400.01
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$349.02
Category: Dining Out							
	10/5/2008	Deposit	Check from Bill for Dad's b-day gift			$543.33	$892.35
503	10/6/2008	Micro Center	New 30" LCD		$2,795.00		($1,902.65)
	10/6/2008	Deposit	Transfer from Savings			$1,000.00	($902.65)

Current Balance: ($902.65)

Figure 15: Final data grid app

That's it for the DataGrid sample – next we'll look at the Ribbon control and how we can replace the menu and toolbar with a more intuitive and easier command organizer.

Exercise 2 – Utilizing the Ribbon control

Since the release of Office 2007, people have been fascinated by the Office Ribbon and trying to emulate it in their applications. It provides, what is considered today, the modern look for an application:

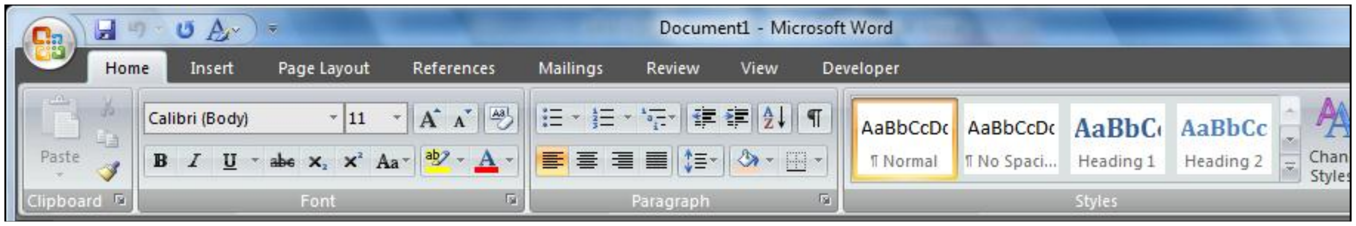


Figure 16: Office Ribbon

Now, with this preview of WPF, you can take advantage of this great UI manager and use the built-in Ribbon control. In this exercise, we will examine the Ribbon and see how to utilize it in our application. It supports all the features you see in the Office variety along with WPF's excellent styling and theme capabilities.

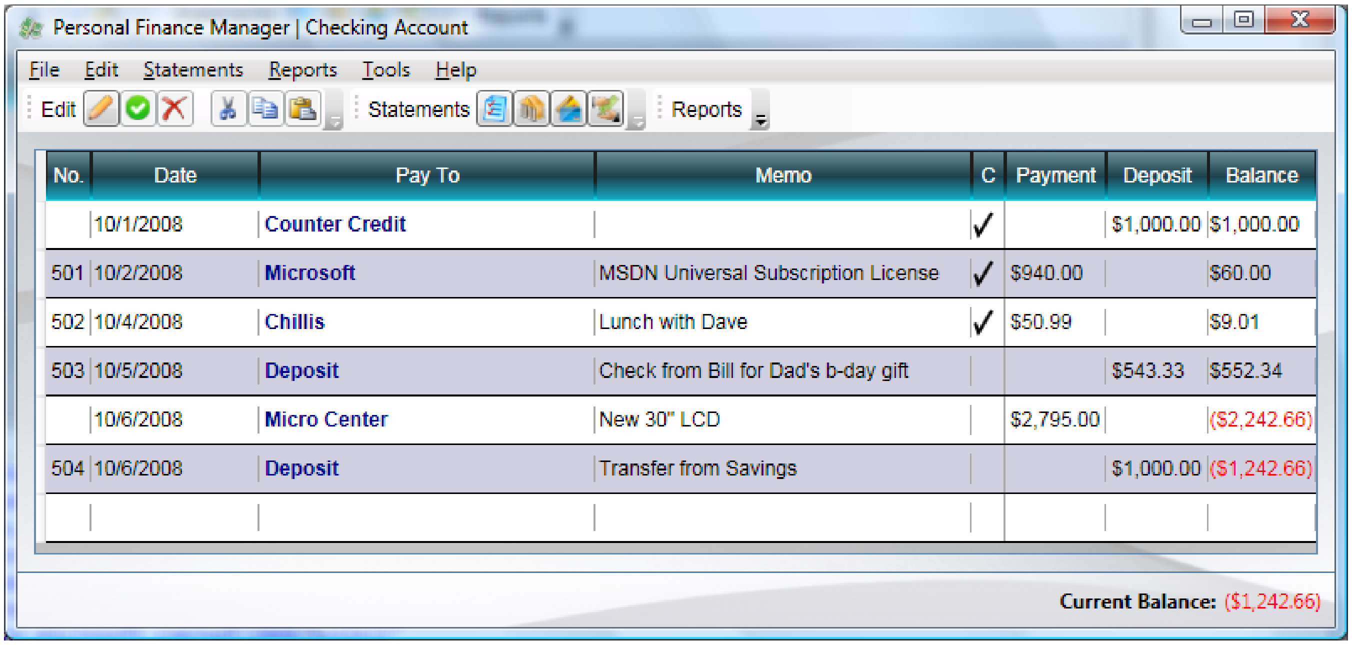
This exercise is intended to demonstrate how to utilize the Ribbon – we won't cover all scenarios here, but at the end you will have used the Ribbon in the most common fashion and seen what it takes to integrate it into an application. Along the way we will:

- Remove an existing menu + toolbar combination
- Add the Ribbon assembly to the application
- Display all the available commands using the Ribbon

We will start with the application developed in Exercise 1, but with some minor additions – so makes sure to use the starter project provided here so you don't have to type as much.

Task 1 – Using the Ribbon control

1. Open source\exercise2\start\CheckbookManager.sln in Visual Studio 2008 with SP1. This is the existing application you will start with.
2. Compile the application.
3. Run the application – it should look like:



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$9.01
503	10/5/2008	Deposit	Check from Bill for Dad's b-day gift			$543.33	$552.34
	10/6/2008	Micro Center	New 30" LCD		$2,795.00		($2,242.66)
504	10/6/2008	Deposit	Transfer from Savings			$1,000.00	($1,242.66)

Current Balance: ($1,242.66)

Figure 17: Starting Ribbon application

4. The first step is to add a reference to the Ribbon assembly. Since the Ribbon is licensed (for free) under the Office Fluent UI License, you must agree to this license before you can use the Ribbon control. To get the Ribbon binary:
   a. Go to http://msdn.microsoft.com/officeui.
   b. Click on "License the Office UI."
   c. Sign in using your Windows Live ID. If you don't have a Windows Live ID already, you can create one for free using the links on that page.
   d. Fill out the Office Fluent UI License form and agree to the license.
   e. On the following page, click on the button which says "WPF Ribbon Control" and download the Ribbon

   Next, you'll need to add the RibbonControlsLibrary.dll to this project:
   a. In the Ribbon download, find the **RibbonControlsLibrary.dll**
   b. Copy and paste the file into the "dependencies" folder of this lab
   c. To add a reference to the Ribbon assembly, follow the steps from Exercise 1-1 but choose the **RibbonControlsLibrary** this time.
5. Next, open MainWindow.xaml and delete the Menu and ToolBarTray tags – that is what we will replace with the Ribbon. Make sure to keep the StatusBar – we want to retain that.
6. You can also delete all the **`<Image>`** tags from the **DockPanel.Resources** – they were used by the menu and toolbar and are not necessary, although we will use the same resources in our Ribbon version.
7. Next, add a namespace declaration to be able to access the Ribbon controls to the top of the file.
   a. Give it a prefix – the lab will use "r".
   b. The CLR namespace is "Microsoft.Windows.Controls.Ribbon" and the assembly is RibbonControlsLibrary.

```
<Window x:Class="CheckbookManager.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:me="clr-namespace:CheckbookManager"
    xmlns:Data="clr-namespace:CheckbookManager.Data"
    xmlns:Converters="clr-namespace:CheckbookManager.Converters"
    xmlns:r="clr-namespace:Microsoft.Windows.Controls.Ribbon;assembly=RibbonControlsLibrary"
    xmlns:dg="clr-namespace:Microsoft.Windows.Controls;assembly=WPFToolkit" ...
```

8. Now, just before the **`<StatusBar>`**, add an **`<r:Ribbon>`** control and dock it to the top of the window. If you have a design view open, then you should see the ribbon appear.

```
    </DockPanel.Resources>

    <r:Ribbon DockPanel.Dock="Top">
    </r:Ribbon>

    <!-- Status bar on bottom: holds balance -->
    <StatusBar DockPanel.Dock="Bottom" Background="{StaticResource whiteBackBrush}"
```

The Ribbon is driven through **RoutedCommands** – just like menus and toolbars (normally). In fact, all the commands we have defined will be exposed through the Ribbon we've created once we tie them together. The Ribbon requires more than just the textual name however – it needs images, tooltip information and descriptions to display the command appropriately. To that end, the Ribbon defines a new subclass of RoutedCommand called **RibbonCommand** and everything you do with the Ribbon will generally involve a RibbonCommand.

Each RibbonCommand has several pieces of information which it adds to the RoutedCommand:

a) **LabelTitle** – used to display a label on the item in the Ribbon
b) **LabelDescription** – used to display descriptive information.
c) **ToolTipTitle** – used to display a title for the tooltip created for the command
d) **ToolTipDescription** – used to display a second line of text on the tooltip
e) **ToolTipImageSource** – used to display an image on the tooltip
f) **SmallImageSource** – the image used when the item is rendered in small size
g) **LargeImageSource** – the image used when the item is rendered in large size
h) **ToolTipFooterTitle** – used to display a footer title for the tooltip created for the command
i) **ToolTipFooterDescription** – used to display a second line of text on the footer tooltip
j) **ToolTipFooterImageSource** – used to display an image on the footer tooltip

The ToolTip properties are used by the Ribbon to automatically create tooltips for controls that are associated to the command. The Small/Large images are used to render the content for the control along with the LabelTitle in some cases.

The first step in using the Ribbon is to define the Application Menu. This is used to present the menu from the application menu which is the most evident portion of the Ribbon we see so far.

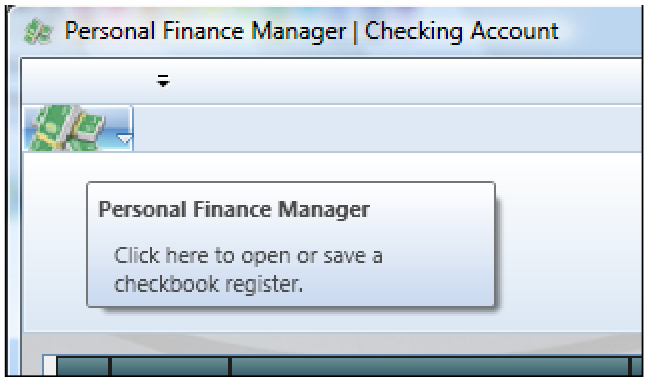
9. Assign the **`<r:Ribbon.ApplicationMenu>`** property to a new **`<r:RibbonApplicationMenu>`** object.
    a. The primary thing we need to assign is a **RibbonCommand** to the **Command** property of the ApplicationMenu – this is where the image and default action will come from.
10. Create a new **RibbonCommand** in XAML and assign it to the **`<r:RibbonApplicationMenu.Command>`** property.
    a. Set the "Executed" handler to "OnCloseApplication" – this is a code behind method that executes "Close()".

b. Set the **LabelTitle** to “Application Button”
c. Set the **LabelDescription** to “Close the Application”.
d. Set the **SmallImageSource** to “images/Coins.png” [NOTE THE CASING!]
e. Set the **LargeImageSource** to “images/Coins.png”
f. Set the **ToolTipTitle** to “Personal Finance Manager”
g. Set the **ToolTipDescription** to some text to display under the title.

11. The XAML should look something like:

```
<r:Ribbon.ApplicationMenu>
   <r:RibbonApplicationMenu>
      <r:RibbonApplicationMenu.Command>
         <r:RibbonCommand
              Executed="OnCloseApplication"
              LabelTitle="Application Button"
              LabelDescription="Close the application."
              SmallImageSource="images/Coins.png"
              LargeImageSource="images/Coins.png"
              ToolTipTitle="Personal Finance Manager"
              ToolTipDescription="Click here to open or save a checkbook register." />
      </r:RibbonApplicationMenu.Command>
   </r:RibbonApplicationMenu>
</r:Ribbon.ApplicationMenu>
```

12. Run the application and hover over the button:


Personal Finance Manager | Checking Account

Personal Finance Manager

Click here to open or save a checkbook register.

Figure 18: Ribbon Pearl tooltip

13. This is the effect of the **RibbonCommand** you created. It supplies the action and visual details to this element in the Ribbon and it’s exactly how everything is setup in the Ribbon.

If you click on the button, you will see that it pulls down an empty menu – this is because we didn’t put any items into it. The application menu expects to have **RibbonApplicationMenuItem** children added to it. Let’s add a simple menu item just as an example:

14. Go back to the code and create a new **<r:RibbonApplicationMenuItem>** inside the Ribbon application menu tag:

```
        ToolTipTitle="Personal Finance Manager"
        ToolTipDescription="Click here to open or save a checkbook register." />
```

```
    </r:RibbonApplicationMenu.Command>

    <r:RibbonApplicationMenuItem>
    </r:RibbonApplicationMenuItem>
</r:RibbonApplicationMenu>
```

15. Assign the RibbonApplicationMenuItem.Command property to a new RibbonCommand – use the same basic definition you assigned to the application menu itself, except change the LabelTitle to "Close":

```
            ToolTipTitle="Personal Finance Manager"
            ToolTipDescription="Click here to open or save a checkbook register." />
        </r:RibbonApplicationMenu.Command>

        <r:RibbonApplicationMenuItem>
            <r:RibbonApplicationMenuItem.Command>
                <r:RibbonCommand
                    LabelTitle="_Close"
                    LabelDescription="Close the Application"
                    Executed="OnCloseApplication" />
                </r:RibbonApplicationMenuItem.Command>
            </r:RibbonApplicationMenuItem>
        </r:RibbonApplicationMenu>
```

16. Run the application again and note that you now have a menu item exposed in the ribbon application menu.

As you can see, **RibbonCommand** is very central to the behavior and appearance of the ribbon control itself. The next step would normally be to convert anything you want to expose through the Ribbon into a **RibbonCommand** – however that would take a significant amount of time so the exercise already has the conversion done for you.

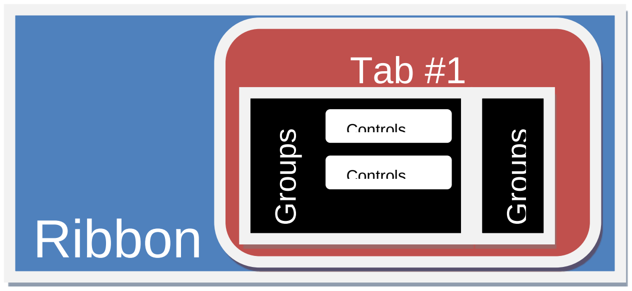
17. Open the **AppCommands.xaml** file – here we've defined every **Command** in terms of a **RibbonCommand** and bound the proper images to each.
    a. Uncomment the **RibbonCommand** definitions – make sure to remove the ending ">" on the prior line which allowed it to compile.
    b. Open the **AppCommands.cs** file – here is where the original commands were defined – we are going to replace these definitions with the new ribbon versions.
    c. Delete all the existing commands and uncomment the Ribbon command section which uses the RibbonCommand objects defined in the resources.
    d. You will need to add a using statement for **`System.Windows`** and **`Microsoft.Windows.Controls.Ribbon`** at the top of the file.

A quick and easy way to uncomment a code block in VS.NET is to highlight the block and press CTRL+K+U or use the Edit | Advanced | Uncomment Selection option from the menubar.

Note: there are a couple of ways to create **RibbonCommands** – we've chosen to use XAML here because most of it is text and this makes it easier to internationalize the content, however you could create them completely in code-behind as well.

18. Compile the code and make sure it runs.

Next, let's begin creating the ribbon. The Ribbon is composed of **Tabs**, **Groups** and **Controls**. Groups and Controls are bound to commands – that's where they get their visual information as well as any Command to raise when they are invoked.



It's helpful to sit down and decide how you want your commands to be organized and what the appropriate representation would be. Typically:

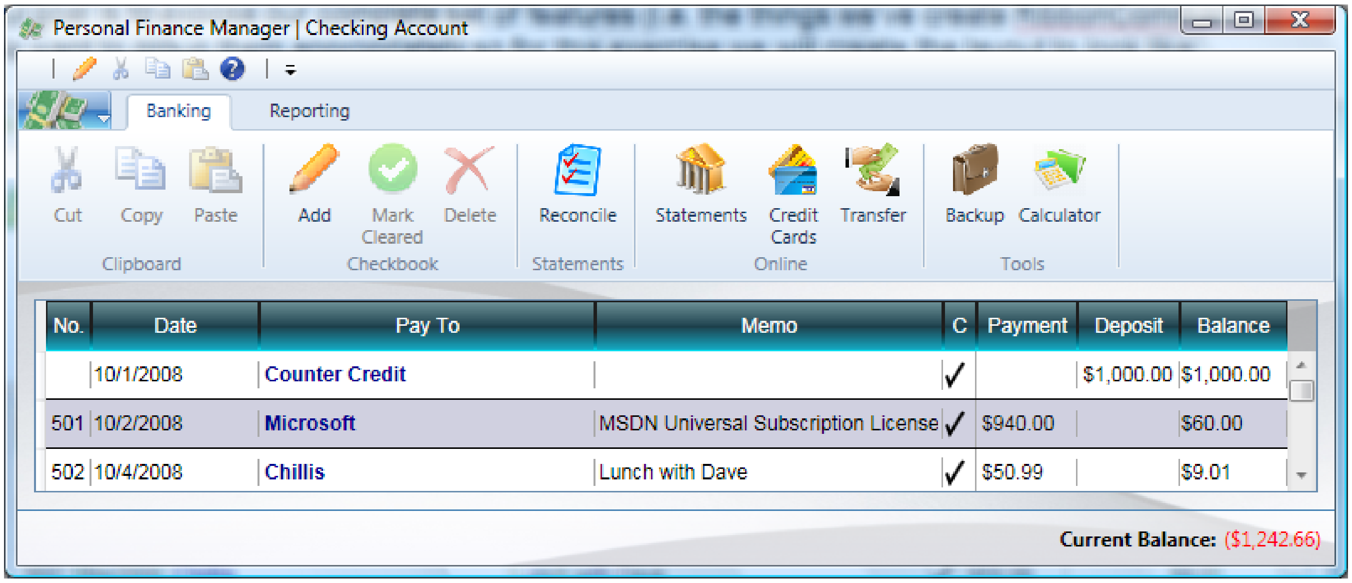
a) **Tabs** are used to group common functionality – things that are used often, or features that naturally work together
b) **Groups** are used within tabs to separate similar, but distinct commands. Office, for example, has a single Tab "Home" but has a group for Clipboard, Font, Paragraph, Styles and Editing.

That brings us to Controls. Controls are the elements the user interacts with. The most common control you will place on the Ribbon will be the button, but the Ribbon has several base controls to choose from for the most common UI scenarios. For ultimate flexibility you can always create your own controls to be hosted in the ribbon. The Ribbon comes with:

- **`RibbonButton`**
- **`RibbonCheckBox`**
- **`RibbonToggleButton`**
- **`RibbonDropDownButton`**
- **`RibbonSplitButton`**
- **`RibbonComboBox`**
- **`RibbonTextBox`**
- **`RibbonLabel`**
- **`RibbonSeparator`**
- **`RibbonToolTip`**

All of these correspond directly to the normal WPF counterpart and in most cases are actually implemented by deriving from the underlying WPF control and adding the implementation for a new interface the Ribbon uses to manage the control called **IRibbonControl**. This is the interface you would use yourself to create new controls.

Our goal is to expose our complete set of features (i.e. the things we've create RibbonCommands for). We want to group them appropriately so for this exercise we will create the layout to look like:



Personal Finance Manager | Checking Account

No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$9.01

Current Balance: ($1,242.66)

Figure 19: Ribbon layout design

Notice we have two Tabs (Banking and Reporting) and five Groups in the Banking tab. Let's see how to implement that.

19. Open MainWindow.xaml and locate your Ribbon control. Create two r:RibbonTab elements and place them into the Ribbon:
    a. Set the **Label** property on each to be the text you want displayed under the tab.

```
        <r:Ribbon DockPanel.Dock="Top">
            <r:RibbonTab Label="Banking">
            </r:RibbonTab>

            <r:RibbonTab Label="Reporting">
            </r:RibbonTab>
        </r:Ribbon>
```

Now, let's define the groupings. Each Group has several things it needs – a Command (where it gets the image when it is sized too small to display the group as well as the Command it will raise for dialogs created from the group, and a set of controls to hold in the group.

20. Create a **`<r:RibbonTab.Group>`** element in the Banking tab.
21. Inside that, create a **`<r:RibbonGroup />`**

```
            <r:RibbonTab Label="Banking">
                <!-- Define the groups in this tab -->
                <r:RibbonTab.Groups>
                    <!-- Clipboard commands -->
                    <r:RibbonGroup>
                    </r:RibbonGroup>
                </r:RibbonTab.Groups>
```

Each group can have a collection of **GroupSizeDefinitions** associated with it that tells the group exactly how to place any children. If it's not defined the Ribbon will use a default layout. Using these definitions, the Ribbon resizes the group based on the available size – this allows the ribbon to resize the group dynamically and the controls will shift around as necessary based on the definition.

If you need even more control, you can create your own custom layout and apply it to the Ribbon so it uses whatever criteria you desire to resize. For now, we will create a simple GroupSizeDefinition to place three controls next to each other using large icons:

22. In the group, define a **`<r:RibbonGroup.GroupSizeDefinitions>`** property
23. Create a **`<r:RibbonGroupSizeDefinitionCollection>`** and assign it to the property
24. Add a **`<r:RibbonGroupSizeDefinition>`** to the collection
25. Add three **`<r:RibbonControlSizeDefinition>`** objects to the group size definition created in the prior step.
    a. Each should set the **ImageSize** to "Large" and **IsLabelVisible** to "True".

```
      <!-- Clipboard commands -->
      <r:RibbonGroup>
         <r:RibbonGroup.GroupSizeDefinitions>
            <r:RibbonGroupSizeDefinitionCollection>
               <r:RibbonGroupSizeDefinition>
                  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
                  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
                  <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
               </r:RibbonGroupSizeDefinition>
            </r:RibbonGroupSizeDefinitionCollection>
         </r:RibbonGroup.GroupSizeDefinitions>
      </r:RibbonGroup>
```

26. To share common ordering and layout, you will commonly place your group size definitions into resources and then just assign the property to a single set of sizes. You can replace the above with:

```
    <r:Ribbon DockPanel.Dock="Top">
       <r:Ribbon.Resources>
          <r:RibbonGroupSizeDefinitionCollection x:Key="RibbonLayout">
             <r:RibbonGroupSizeDefinition>
                <!-- Control sizes: L,L,L -->
                <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
                <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
                <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
             </r:RibbonGroupSizeDefinition>
             <r:RibbonGroupSizeDefinition>
                <!-- Control sizes: L,M,M -->
                <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
                <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
                <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="True"/>
             </r:RibbonGroupSizeDefinition>
             <r:RibbonGroupSizeDefinition>
                <!-- Control sizes: L,S,S -->
                <r:RibbonControlSizeDefinition ImageSize="Large" IsLabelVisible="True"/>
                <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
                <r:RibbonControlSizeDefinition ImageSize="Small" IsLabelVisible="False"/>
```

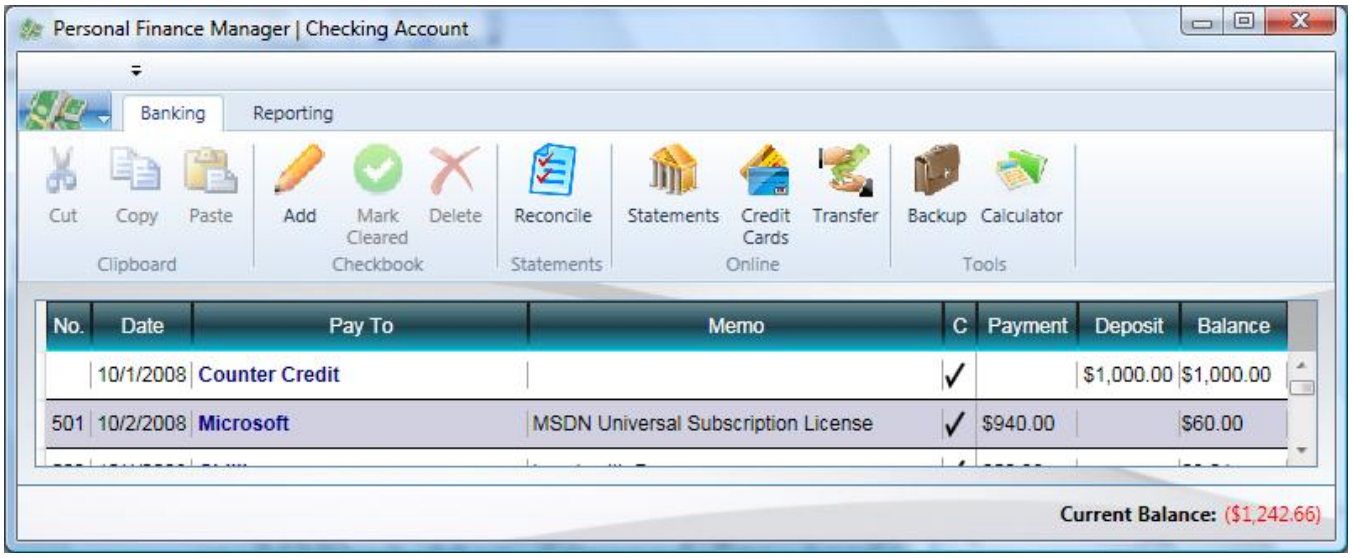
```
            </r:RibbonGroupSizeDefinition>
               <!-- Collapsed -->
            <r:RibbonGroupSizeDefinition IsCollapsed="True" />
         </r:RibbonGroupSizeDefinitionCollection>
      </r:Ribbon.Resources>

      <r:RibbonTab Label="Banking">
         <r:RibbonTab.Groups>
            <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
            </r:RibbonGroup>
         </r:RibbonTab.Groups>
      </r:RibbonTab>
   </r:Ribbon>
```

Here we define several group size definitions – one for each layout we expect. Note that we assume here we will not have more than 3 controls in a group, clearly that would be a design decision and dictated by how you are organizing controls. This definition will now allow us to reuse this single group definition across all our defined groups. Now let's add some controls!

27. We'll use **RibbonButton** controls for the clipboard elements – inside the **`<r:RibbonGroup>`** add three **`<r:RibbonButton>`** elements and tie the Command properties to
    - **a. AppCommands.Cut**
    - **b. AppCommands.Copy**
    - **c. AppCommands.Paste**
28. Finally, add a Command onto the group – just create a RibbonCommand directly and assign it to the **r:RibbonGroup.Command** property.
    - a. Set the **LabelTitle** to "Clipboard" and the **SmallImageSource** to "images/Paste.png". This will be used to render the group as a whole when it is collapsed.
29. Your code should look like:

```
      <!-- Clipboard commands -->
      <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
         <r:RibbonGroup.Command>
            <r:RibbonCommand LabelTitle="Clipboard" SmallImageSource="images/Paste.png" />
         </r:RibbonGroup.Command>
         <r:RibbonButton Command="me:AppCommands.Cut"/>
         <r:RibbonButton Command="me:AppCommands.Copy"/>
         <r:RibbonButton Command="me:AppCommands.Paste"/>
      </r:RibbonGroup>
```

30. Run the application to see the effects of the group – you should now have valid Cut/Copy/Paste elements!
31. Finish this tab by creating the additional groups in the screen shot – bind to the appropriate commands (you can see the list in AppCommands.xaml) and use the same technique you did in the previous steps to define the groups.



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00

Current Balance: ($1,242.66)

Figure 20: Final groups

32. If you need help, the code will look like:

```
<r:RibbonTab Label="Banking">
   <r:RibbonTab.Groups>
      <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <r:RibbonGroup.Command>
         <r:RibbonCommand LabelTitle="Clipboard" SmallImageSource="images/Paste.png" />
      </r:RibbonGroup.Command>
      <r:RibbonButton Command="me:AppCommands.Cut"/>
      <r:RibbonButton Command="me:AppCommands.Copy"/>
      <r:RibbonButton Command="me:AppCommands.Paste"/>
   </r:RibbonGroup>

   <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <r:RibbonGroup.Command>
         <r:RibbonCommand LabelTitle="Checkbook" SmallImageSource="images/AddNew.png" />
      </r:RibbonGroup.Command>
      <r:RibbonButton Command="me:AppCommands.AddNew"/>
      <r:RibbonButton Command="me:AppCommands.Clear" />
      <r:RibbonButton Command="me:AppCommands.Delete"/>
   </r:RibbonGroup>

   <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <r:RibbonGroup.Command>
         <r:RibbonCommand LabelTitle="Statements"
                          SmallImageSource="images/Reconcile.png" />
      </r:RibbonGroup.Command>
      <r:RibbonButton Command="me:AppCommands.Reconcile"/>
   </r:RibbonGroup>

   <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
      <r:RibbonGroup.Command>
         <r:RibbonCommand LabelTitle="Online"
                          SmallImageSource="images/CreditCards.png" />
         </r:RibbonGroup.Command>
```

```
            <r:RibbonButton Command="me:AppCommands.DownloadStatements"/>
            <r:RibbonButton Command="me:AppCommands.DownloadCreditCards"/>
            <r:RibbonButton Command="me:AppCommands.Transfer"/>
         </r:RibbonGroup>

         <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
            <r:RibbonGroup.Command>
               <r:RibbonCommand LabelTitle="Tools" SmallImageSource="images/Backup.png" />
            </r:RibbonGroup.Command>
            <r:RibbonButton Command="me:AppCommands.Backup"/>
            <r:RibbonButton Command="me:AppCommands.Calculator"/>
        </r:RibbonGroup>
     </r:RibbonTab.Groups>
  </r:RibbonTab>
```

33. Now that you have multiple groups, run the application and try resizing it – notice how the group size definition kicks in as the width or height of the application changes:

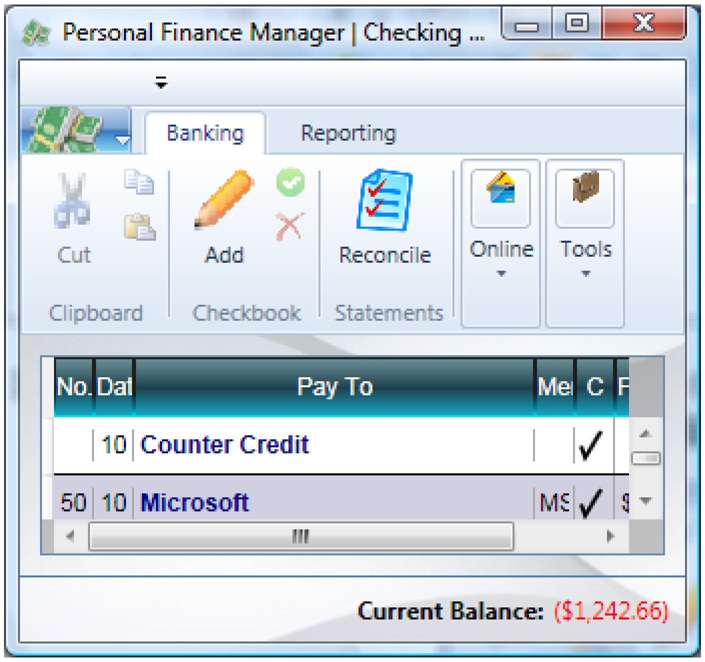


Figure 21: Resized Ribbon

Note: If you don't like the order the Ribbon collapses the groups in, you can adjust it by assigning names to each group and set the **RibbonTab.GroupSizeReductionOrder** to specify which order the groups will change layout templates for as the window is resized.

34. The second tab looks much like the first with one exception – it uses a **RibbonDropDownButton** with a set of **MenuItem** objects to display a list of reports:

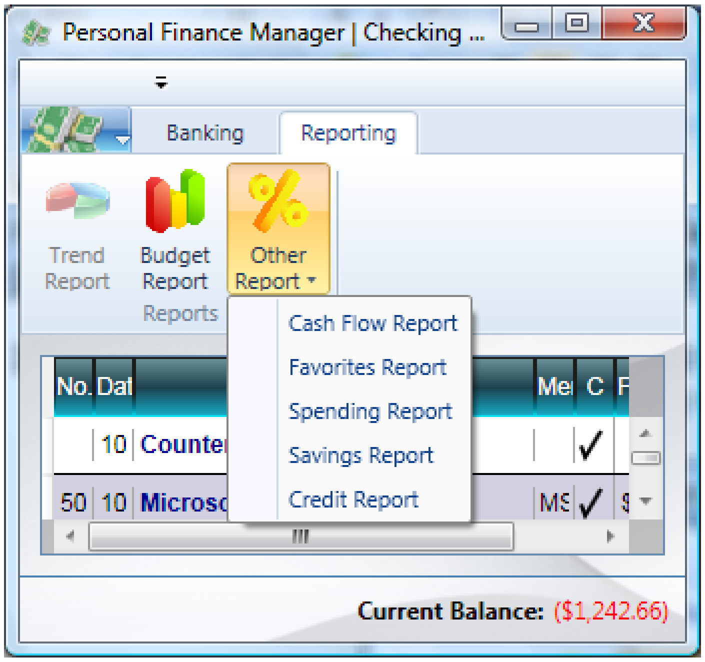


Figure 22: RibbonDropDownButton

35. See if you can add the second tab yourself – if you need help, the code is:

```
    <r:RibbonTab Label="Reporting">
       <r:RibbonTab.Groups>
          <r:RibbonGroup GroupSizeDefinitions="{StaticResource RibbonLayout}">
             <r:RibbonGroup.Command>
                <r:RibbonCommand LabelTitle="Reports"
                        SmallImageSource="images/CashflowReport.png" />
             </r:RibbonGroup.Command>
             <r:RibbonButton Command="me:AppCommands.TrendReport" />
             <r:RibbonButton Command="me:AppCommands.BudgetReport"/>
             <r:RibbonDropDownButton Command="me:AppCommands.OtherReports">
                <MenuItem Header="Cash Flow Report" />
                <MenuItem Header="Favorites Report" />
                <MenuItem Header="Spending Report" />
                <MenuItem Header="Savings Report" />
                <MenuItem Header="Credit Report" />
             </r:RibbonDropDownButton>
          </r:RibbonGroup>
       </r:RibbonTab.Groups>
    </r:RibbonTab>
```

The final thing we want to do in our application is take the most actively used commands in the ribbon and move them to the “Quick Access Toolbar” list which is located next to the Application Menu. This is done by assigning a **QuickAccessToolBar** to the **Ribbon.QuickAccessToolBar** property.

36. Create a **`<r:Ribbon.QuickAccessToolBar>`** tag in the ribbon (just after the resources is a good place).
37. Add a **`<r:RibbonQuickAccessToolBar>`** into the tag
    a. Set the **CanUserCustomize** property to “True”

38. Add your favorite commands using **RibbonButton** objects into the toolbar. The lab will use the commands AddNew, Cut, Copy, Paste and Help but feel free to try any of them.
    a. You can use the **RibbonQuickAccessToolBar.Placement** attached property on each button to decide whether the user can remove the command from the toolbar. The valid settings are "InCustomizeMenu", "InToolBar" and "InCustomizeMenuAndToolBar".
39. The completed code looks something like (your commands can vary):

```
  <!-- Quick access menu -->
  <r:Ribbon.QuickAccessToolBar>
     <r:RibbonQuickAccessToolBar CanUserCustomize="True">
        <r:RibbonButton Command="me:AppCommands.AddNew"
                 r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
        <r:RibbonButton Command="me:AppCommands.Cut"
                 r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
        <r:RibbonButton Command="me:AppCommands.Copy"
                 r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
        <r:RibbonButton Command="me:AppCommands.Paste"
                 r:RibbonQuickAccessToolBar.Placement="InCustomizeMenuAndToolBar" />
        <r:RibbonButton Command="me:AppCommands.Help"
                 r:RibbonQuickAccessToolBar.Placement="InToolBar" />
     </r:RibbonQuickAccessToolBar>
  </r:Ribbon.QuickAccessToolBar>

  <r:RibbonTab Label="Banking">
```

40. Run the application –

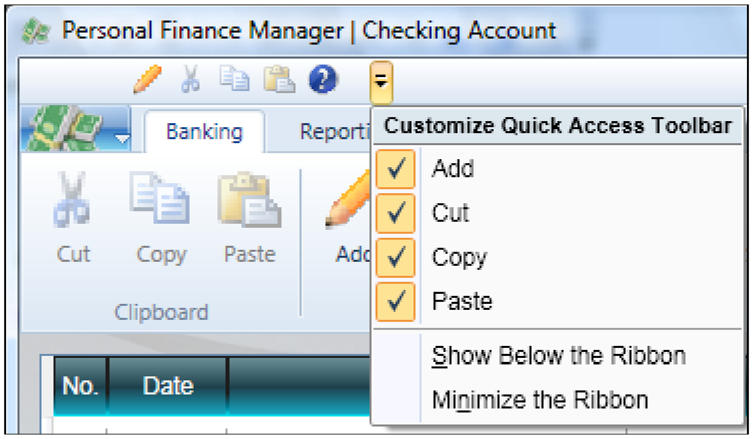


Figure 23: Quick access toolbar

Notice how our popular picks are now always available no matter which tab we are on and the user can click the down-arrow and customize the list (within the limits we provided through the attached properties).

That is the end of this exercise but not the end of the Ribbon control; it is heavily customizable through templates – and has several controls we didn't get the chance to try here. Consider adding some of the other ribbon controls on your own using this as a base, or move onto the next exercise where we will use the **RibbonWindow** control to provide glass effects to our application!

Task 2 – Providing Vista "Glass" effects

In this task, we will look at another common request – how to utilize "glass" effects in our application when running under Windows Vista. The Ribbon control adds this capability through a new Window-derived class called **RibbonWindow**. This new class simply replaces the Window and automatically removes the title bar and makes the application more modern looking.

Note: in order to run this successfully and see the effect, you will need to be running on Windows Vista with the Aero theme enabled.

1. Open the starter application at source\exercise2\task2\CheckbookManager.sln or just continue on from the previous task.
2. Compile the application to ensure all the value converters are available.
3. Open the MainWindow.xaml file and replace the root <Window> tag with <r:RibbonWIndow>. Make sure to also replace the ending tag.
4. Open the MainWindow.xaml.cs file (if you don't see it, click the "+" next to the XAML file to locate it).
    a. Replace the base "Window" class with "RibbonWindow", or just remove it altogether as the partial class generated by the XAML compiler will have the proper base.
5. Run the application – notice how our quick access menu has moved into the title bar – this is what the GlassWindow will do – reduce the space necessary and give you the Office 2007 effect.
6. It looks decent but there are two issues with it.
    a. There is no title on the title bar now – it was removed by the Glass Window
    b. The icon is being displayed under the application menu.

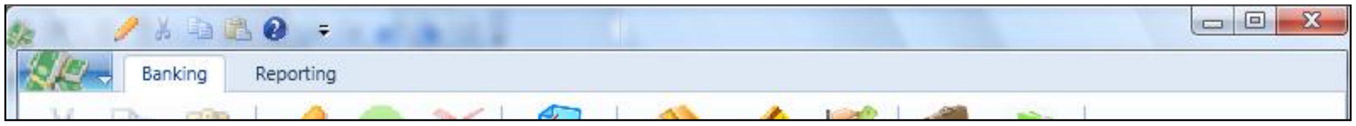


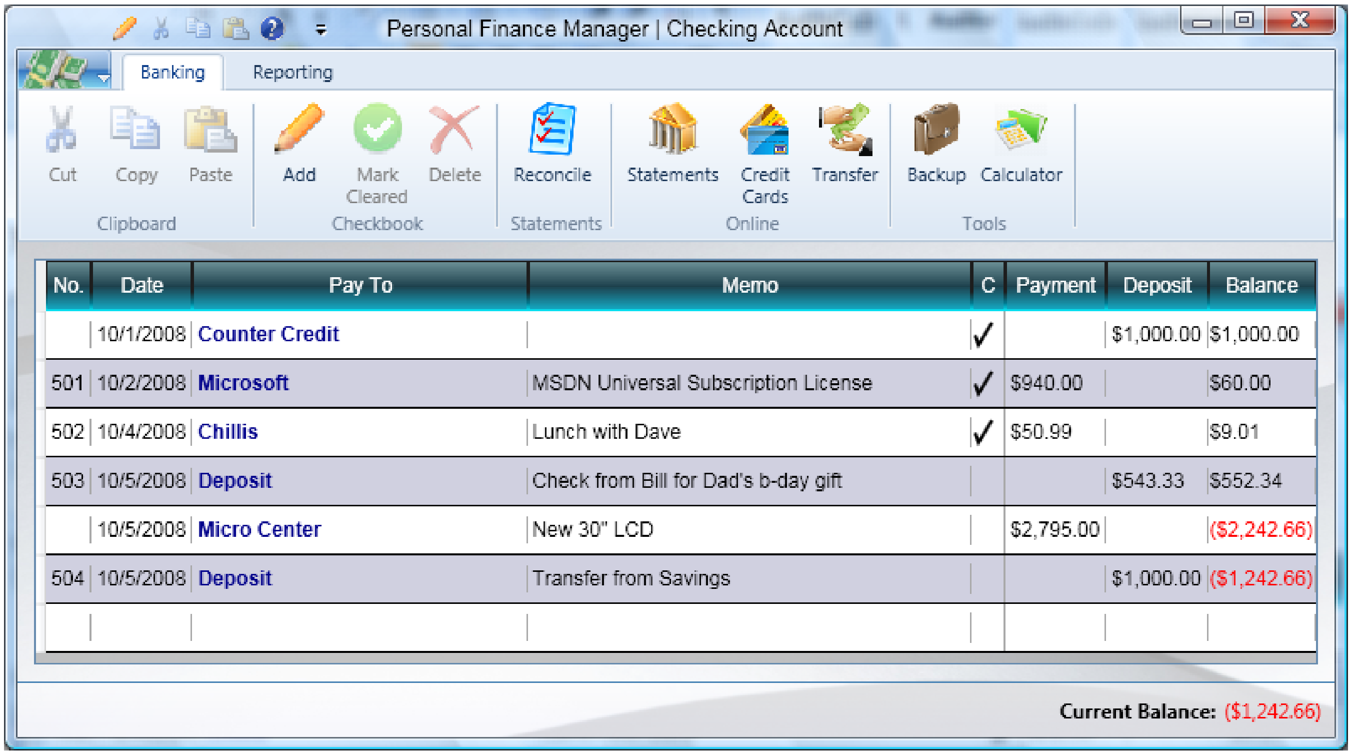
Figure 24: Enabling Glass Step 1

7. We can fix both of these issues in the XAML:
    a. First, locate the **<Ribbon>** tag – we need to set a **Title** property on this element to get our title display with the Ribbon Window. We could move the Window version, or use a little data binding magic to use the existing window title:

```
<r:Ribbon DockPanel.Dock="Top"
          Title="{Binding RelativeSource={RelativeSource FindAncestor,AncestorType={x:Type
  Window}},Path=Title}">
```

This will walk the visual tree and locate the Window ancestor and then bind the ribbon's **Title** property to the **Title** property assigned on the window.

    b. Next, remove the Icon property off the GlassWindow.
8. Run the application – it should now look more pleasing to the eye:



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$9.01
503	10/5/2008	Deposit	Check from Bill for Dad's b-day gift			$543.33	$552.34
	10/5/2008	Micro Center	New 30" LCD		$2,795.00		($2,242.66)
504	10/5/2008	Deposit	Transfer from Savings			$1,000.00	($1,242.66)

Current Balance: ($1,242.66)

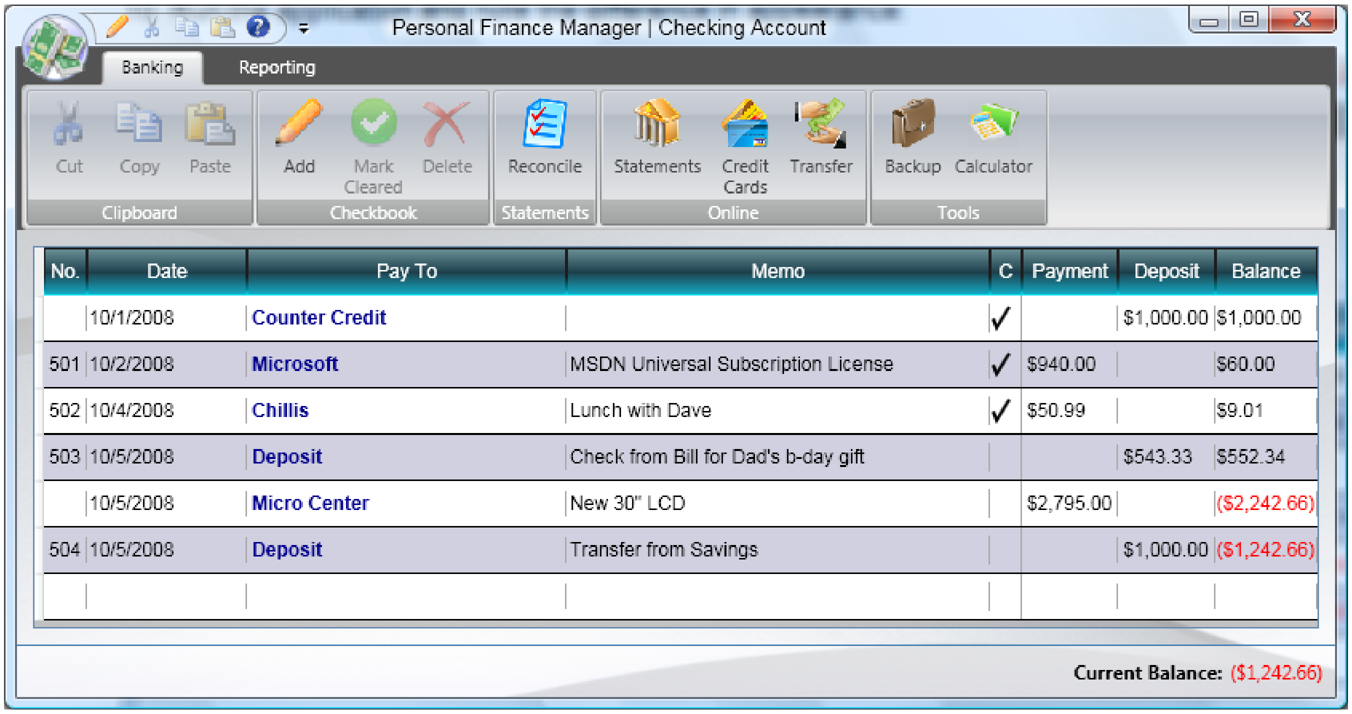
Figure 25: Fixing the glass ribbon

The final thing you might consider is to change the default style of the ribbon control – it, by default, utilizes a Windows 7 theme, but you can change it to look more like Office 2007 by simply merging in different styles into your application or window resources.

9. Open MainWindow.xaml.cs and locate the window constructor.
10. As the first line, add the following code to merge in the Office 2007 "black" theme:

```
        public MainWindow()
        {
            this.Resources.MergedDictionaries
                .Add(Microsoft.Windows.Controls
                      .Ribbon.PopularApplicationSkins.Office2007Black);
```

11. Run the application and note the difference in appearance:



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$9.01
503	10/5/2008	Deposit	Check from Bill for Dad's b-day gift			$543.33	$552.34
	10/5/2008	Micro Center	New 30" LCD		$2,795.00		($2,242.66)
504	10/5/2008	Deposit	Transfer from Savings			$1,000.00	($1,242.66)

Current Balance: ($1,242.66)

Figure 26: Final ribbon "glass" application with Office 2007 theme

Exercise 3 – Using the Visual State Manager

Silverlight 2 is a subset of WPF and one of the features that is not present is Triggers. That makes it difficult to provide visual activity based on property changes without resorting to code behind – and in fact, that's exactly how the original betas of Silverlight solved the problem. Starting in Beta 2 however, Silverlight introduced a new way to deal with visual behavior – the Visual State Manager (VSM).

The VSM provides a way to describe the various states that a particular element might go through and the animations to play for each one, as well as how to transition from one state to another.

For example, the Button has several known states:

a) Pressed
b) Unpressed
c) MouseOver
d) Normal
e) Disabled

Each of these impacts the visual appearance of the button and the VSM can capture that information and automatically activate the appropriate animations in response. You might think that Triggers do the same thing, and you'd be right, however triggers operate at a much lower level – they are based on property changes, not necessarily UI states (although there tends to be a relationship between them). Because the VSM operates at a higher level, it's easier to deal with and specifically easier to have tools and designers generate the appropriate visual behavior for custom controls.

As part of the WPF Futures package, we now have a WPF-based VSM which we can use to drive state changes in our UI. This exercise will introduce the VSM to you using it in two tasks:

1) Providing the visual behavior for a themed button (traditional use)
2) Providing an animation effect for the UI.

We will be using the same financial application, although it has been slightly modified from the previous exercise to keep you from having to type a lot of code for this example.

Task 1 – Using the VSM with custom controls

1. Open source\exercise3\start\CheckbookManager.sln in Visual Studio 2008 with SP1. This is the existing application you will start with.
2. Compile the application.
3. Run the application – it's got a new toggle button "Show Graph" added to the display. This is the element we will be working with in this task.

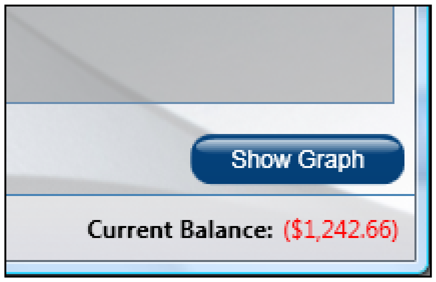


Figure 27: Themed button

4. Move your mouse over the button – notice that nothing happens. You can click it and it works but there is no mouse over effect. This is exactly what the VSM was intended to solve.

Note: normally we would use a tool such as Expression Blend here, but currently it doesn't support the WPF VSM process so we are going to enter the code by hand but recognize that once Blend is updated to support WPF's model you would do the following steps through a UI and not directly.

5. The first step is to locate the style being applied – it's in the ColorsStyles.xaml file and it's called "toggleBlueButtonStyle".
6. In the style, locate the setter for the **Template** property – this is the control template.

To use the Visual State Manager you need to attach a special property to a portion of the Visual Tree that you want to interact with. In the case of control templates, this is commonly the root layout pane (Grid in this case).

We use the attached property **VisualStateManager.VisualStateGroups**. This is populated with a set of Visual State Groups – each defining a set of visual states and the transitions to apply between those states.

In this case we want to apply a visual change when the mouse moves over the element so we will define a new visual state called "MouseOver" and another called "Normal" which will reverse the effect (otherwise the storyboard would keep the mouse over effect on for eternity!)

It turns out that these visual state names are not arbitrary – we need to know the names to drive the VSM in the code behind as you will see in the next task. For controls, such as this button, the VSM defines known states (such as MouseOver and Normal) and as long as we stick with those names for our visual states, they will be applied without us lifting a finger! So, let's get started.

7. Add into the Grid a new tag called **<VisualStateManager.VisualStateGroups>**
8. Inside that tag, add a new **<VisualStateGroup>** with the name "CommonStates".
9. Inside the group, add two **<VisualState>** objects – give the first one the name "MouseOver" and the second one the name "Normal".

```
    <Grid x:Name="rootGrid" RenderTransformOrigin=".5,.5">
                    ...
       <VisualStateManager.VisualStateGroups>
          <VisualStateGroup x:Name="CommonStates">

             <VisualState x:Name="MouseOver">
             </VisualState>

             <VisualState x:Name="Normal">
             </VisualState>

         </VisualStateGroup>
      </VisualStateManager.VisualStateGroups>
        ...
```

Note: you may get a VS.NET Intellisense error related to the VisualStateGroups property – indicating that it doesn't exist. You can ignore this error – the property is there and will work just fine.

Now let's fill in what we want to happen in each of these states. Each VisualState can have a single Storyboard added to it which will be run when that state is entered. We want to change the Stroke of the bottom rectangle to Black and adjust the gradient fill.

If you look at the bottom rectangle (named bottomBorder in the template), the Stroke is set to "{x:Null}". This will be a problem for our storyboard because an animation doesn't replace an object – it only adjusts the value of the existing object. So, to fix this let's add in a Brush we can use to animate.

10. Add a new **SolidColorBrush** into the **Grid.Resources** (you will need to create the resources section in the template). Give it a key of "borderBrush" and set the initial color to Transparent.
11. Apply that brush to the bottomBorder rectangle as its stroke.

```
    <Grid x:Name="rootGrid" RenderTransformOrigin=".5,.5">
       <Grid.Resources>
          <SolidColorBrush x:Key="borderBrush" Color="Transparent" />
       </Grid.Resources>
          ...
       <Rectangle Grid.RowSpan="2" x:Name="bottomBorder" RadiusX="10" RadiusY="10"
                  Stroke="{StaticResource borderBrush}" StrokeThickness="2">
```

12. Now that we have something that is animatable, create a new Storyboard and place it into our **MouseOver VisualState**.
13. Add a **ColorAnimation** to the Storyboard
    a. set the **Storyboard.TargetName** to "bottomBorder"
    b. set the **Storyboard.TargetProperty** to "(Shape.Stroke).(SolidColorBrush.Color)"
    c. set the "**To**" property to "Black"
    d. set the **Duration** to "0:0:0"
14. Add a second **ColorAnimation** to the Storyboard
    a. set the **Storyboard.TargetName** to "bottomBorder"
    b. set the **Storyboard.TargetProperty** to "(Shape.Fill).(GradientStops[0].Color)"
    c. set the "**To**" property to "DarkRed"
    d. set the **Duration** to "0:0:0.25"
15. Now, reverse the effects by adding the same storyboard to the "Normal" visual state except with the proper values and all durations set to "0:0:0" so it applies immediately. The final code should look like:

```
     <VisualState x:Name="MouseOver">
        <Storyboard>
           <ColorAnimation To="DarkRed"
                       Storyboard.TargetName="bottomBorder"
                       Storyboard.TargetProperty="(Shape.Fill).GradientStops[0].Color"
                       Duration="0:0:0.25" />
           <ColorAnimation To="Black"
                       Storyboard.TargetName="bottomBorder"
                       Storyboard.TargetProperty="(Shape.Stroke).(SolidColorBrush.Color)"
                       Duration="0:0:0" />
       </Storyboard>
    </VisualState>
     <VisualState x:Name="Normal">
```

```
        <Storyboard>
           <ColorAnimation To="#FF043F76"
                           Storyboard.TargetName="bottomBorder"
                           Storyboard.TargetProperty="(Shape.Fill).GradientStops[0].Color"
                           Duration="0:0:0" />
           <ColorAnimation To="Transparent"
                           Storyboard.TargetName="bottomBorder"
                           Storyboard.TargetProperty="(Shape.Stroke).(SolidColorBrush.Color)"
                           Duration="0:0:0" />
       </Storyboard>
    </VisualState>
```

16. Run the application and move the mouse over the button. Notice how the effect is being applied automatically to the button – this is because the VSM knows about the MouseOver and Normal state for any **ButtonBase** element and it located the proper **VisualGroup** and ran the animation for us.

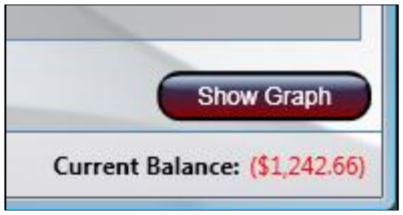


Figure 28: VSM in action

The last step we need to do to round this out is apply visual transitions. VisualTransitions allow you to define how the states interact with each other – for example what to do when going from Normal to Pressed, or Pressed to Disabled. It provides another storyboard which is executed just prior to the VisualState storyboard. We can add transitions easily, although for this example we don't need them:

17. Add a **VisualTransition** to the **VisualStateGroup.VisualTransition** property.
    a. Set the "To" property to "MouseOver" and the GeneratedDuration property to "0:0:0"

```
      <VisualStateGroup x:Name="CommonStates">
         <VisualStateGroup.Transitions>
            <VisualTransition To="MouseOver" GeneratedDuration="0:0:0.0">
               <Storyboard />
            </VisualTransition>
            </VisualStateGroup.Transitions>
         <VisualState x:Name="MouseOver">
```

This one indicates that anytime we are moving to the MouseOver state from some other state (i.e. not when the control is created, but once it is in a known state) then this transition's storyboard would be applied.

In the next task, we'll look at applying the VSM to application-level animations.

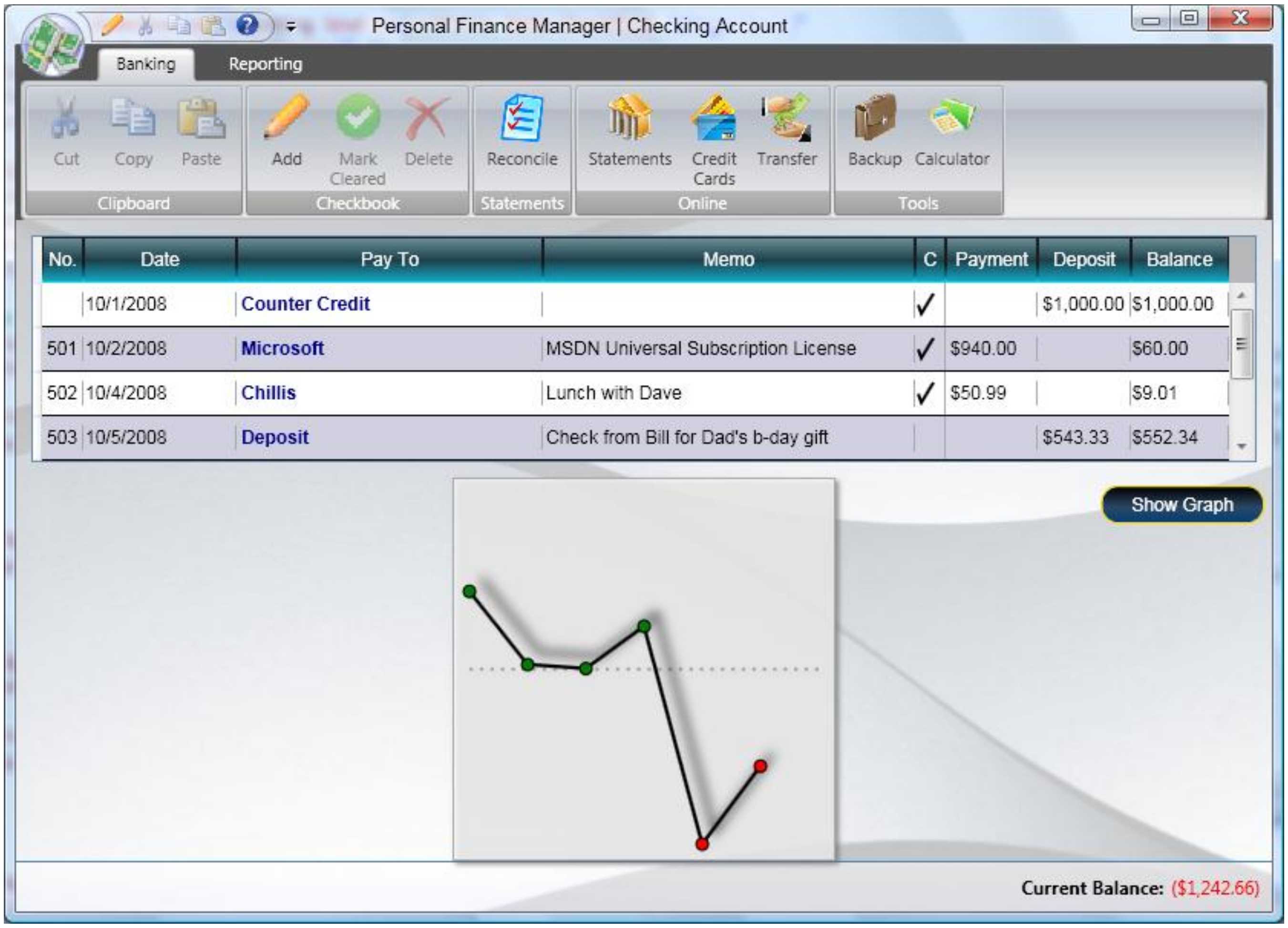
Task 2 – Using the VSM for application-level animations

In this task we are going to add an application animation using the button we created in task 1. The idea is to have a portion of our application slide up when the toggle button is pressed and then hide itself when it is not. The UI for all of this is built already – you are just going to provide the Visual State Manager glue to make it work!

1. Open source\exercise3\task2\CheckbookManager.sln in Visual Studio 2008 with SP1. This is the existing application you will start with. If you performed all the steps in Task 1 then you can simply continue with that project.
2. Compile the application to ensure all the value converters are available.
3. Open MainWindow.xaml – we need to add our new visual element to this screen.
4. Scroll down to the **ToggleButton** (around line 317). Just below that, add the following code to create a custom graphing user control:

```
            <Grid Grid.Row="1">
                <ToggleButton HorizontalAlignment="Right" VerticalAlignment="Top"
                    Click="OnShowGraph"
                    Style="{StaticResource toggleBlueButtonStyle}" Content="Show Graph" />

                <me:AccountBalanceGraph x:Name="graph"
                    Height="250" HorizontalAlignment="Center" />

            </Grid>
```

5. Go ahead and run the application to see what it does.



No.	Date	Pay To	Memo	C	Payment	Deposit	Balance
	10/1/2008	Counter Credit		✓		$1,000.00	$1,000.00
501	10/2/2008	Microsoft	MSDN Universal Subscription License	✓	$940.00		$60.00
502	10/4/2008	Chillis	Lunch with Dave	✓	$50.99		$9.01
503	10/5/2008	Deposit	Check from Bill for Dad's b-day gift			$543.33	$552.34

Current Balance: ($1,242.66)

Figure 29: Graph

The graph is live – if you change data in your checkbook, it adjusts in the graph as well.
The goal is to have the graph show and hide itself based on the state of the Show Graph toggle button. We could easily do this with data binding – but it wouldn't be animated. So, let's use the VSM!

6. Remove the **Height** off the AccountBalanceGraph tag in MainWindow.xaml. That way it will size to the control itself which we will be adjusting
7. Open the AccountBalanceGraph.xaml file – this is where we need to add our visual state information.
8. Add a new **`<VisualStateManager.VisualStateGroups>`** into the root Grid.
9. Create two states – "Active" and "Inactive" in a VisualStateGroup called "CommonStates".
10. In the Active visual state, animate the Border (named "bd") in the Grid –
    a. Change the **Opacity** from "0" to "1" in "0:0:0.25" ms.
    b. Change the **Height** from "0" to "250" in "0:0:0.25" ms.
11. In the Inactive visual state, reverse the animation
    a. Change the **Opacity** to "0" in "0:0:0.1" ms
    b. Change the **Height** to "0" in "0:0:0.1" ms
12. You can add visual transitions if you like but it is unnecessary here.

```
<Grid>
   <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="CommonStates">
```

```
        <VisualStateGroup.Transitions>
           <VisualTransition To="Active" GeneratedDuration="0:0:0.0" />
           <VisualTransition To="Inactive" GeneratedDuration="0:0:0" />
        </VisualStateGroup.Transitions>
        <VisualState x:Name="Active">
           <Storyboard>
              <DoubleAnimation From="0" To="1" Duration="0:0:0.25"
                 Storyboard.TargetProperty="Opacity" Storyboard.TargetName="bd" />
              <DoubleAnimation From="0" To="250" Duration="0:0:0.25"
                 Storyboard.TargetProperty="Height" Storyboard.TargetName="bd" />
            </Storyboard>
         </VisualState>
          <VisualState x:Name="Inactive">
             <Storyboard>
                <DoubleAnimation To="0" Duration="0:0:0.1"
                   Storyboard.TargetProperty="Opacity" Storyboard.TargetName="bd" />
                <DoubleAnimation To="0" Duration="0:0:0.1"
                   Storyboard.TargetProperty="Height" Storyboard.TargetName="bd" />
             </Storyboard>
          </VisualState>
       </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

13. Finally, change the **Border** size ("bd") to "0" initially and the **Opacity** to "0".

```
        <!-- Graph -->
        <Border x:Name="bd" Height="0" Opacity="0" BorderBrush="DarkGray"
                            BorderThickness="1" Background="#C0FFFFFF">
            <Border.Effect>
                <DropShadowEffect  ShadowDepth="3" Color="DarkGray" Opacity="0.85" />
            </Border.Effect>
```

14. Run the application – click the button. What happens?

Nothing. The problem is that now we are creating custom visual states and the VSM knows nothing about them so nothing drives the state machine. In this case, we need to take control and tell the VSM when to go from one state to another – when that button is clicked.

We do this by interacting with the **VisualStateManager** class – it has a static method called "**GoToState**" which we can call at anytime to tell it to transition a control from one state to another. The method looks like this:

```
bool VisualStateManager.GoToState(Control ctrl, string State,
                                  bool useTransitions);
```

The first parameter is the control to change visual states on, the second is the new state and the third tells the VSM whether to use the transitions (if any exist) or to just run the VisualState animations directly.

15. Open the MainWindow.xaml.cs code behind file and locate the **OnShowGraph** method – this is the method wired up to the toggle button.
    a. If the toggle button is checked (IsChecked == true) then call VisualStateManager.GoToState on the graph UserControl to transition to the "Active" state using transitions.
    b. Otherwise, transition to the "Inactive" state.
16. The code should look something like:

```
        private void OnShowGraph(object sender, RoutedEventArgs e)
        {
            ToggleButton tb = (ToggleButton) sender;
            if (tb.IsChecked.Value == true)
                VisualStateManager.GoToState(graph, "Active", true);
            else
                VisualStateManager.GoToState(graph, "Inactive", true);
        }
```

Run the application again and notice the change. Now the animation should play when the button state changes.

Lab Summary

In this lab you performed the following exercises:

- Used the new DataGrid to display rows of editable data.
- Used the new DatePicker to show a drop-down Calendar.
- Used the Ribbon to replace a menu + toolbar and expose the entire command set of the application to the user, allowing for resizing and better organization
- Used the Visual State Manager to apply simple animations to the project.

Hopefully you enjoyed learning about the new features of WPF in this lab. The next step is taking these ideas and applying them to your own projects. Good luck!