Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Your Own MP3 Duration Calculator: Part 2

Rolling out Version 2.0

March 03, 2010
URL:http://www.drdobbs.com/windows/your-own-mp3-duration-calculator-part-2/223101349

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

In Your Own MP3 Duration Calculator: Part 1, I presented a program called "MP3 Duration Calculator" (MDC) that I wrote for my wife Liat who recently completed certification as a spinning instructor. Spinning instructors need to create CDs with music for each class that matches the class content and MDC allows Liat to quickly select different song ands see the total duration of all the songs.

Liat was excited and started browsing her music collection and selecting songs for her first ever spinning class. She was marginally pleased with the interface but complained that the total duration text box is a little obscure. Then something terrible happend! Liat had already selected a bunch of songs and by accident clicked without holding control on another songs. Her entire selection was gone and replaced with the single last song. That was pretty late (about 2:00 a.m.) and her class was scheduled for 8:00 a.m. Liat was upset and re-selected all the songs again. Finally, all the songs were there and it was time to burn the CD. That's when we discovered that we ran out of blank CDs. Alas, Liat brought several commercial CDs to her first spinning class and swapped them manually.

So, it wasn't the best software debut ever. The good thing is that there was a lot of room for improvement. Liat ran into a few other issues. When selecting songs, she sometimes didn't recognize the song's name or just wanted to listen to it to make sure it's appropriate. She had to go to Windows Explorer, find the song, and click on it to being up the Windows Media Player. This process was tedious and broke the flow of selecting songs. After selecting all the songs, she had to manually select them again in Windows Explorer to burn them. Finally, Liat also wanted to print a list of all the songs and their duration. Obviously it was time for Version 2.0.

New Requirements

Before launching into full-blown development, we sat down and compiled the following list of requirements for MDC 2.0:

- Make total duration more prominent
- Check boxes for selection (one song toggle)
- Show # of selected songs
- Play songs from within MDC
- Copy selected songs to a new folder
- Export song list to a CSV file
- Remember last folder

One of the main features of MDC 1.0 was its interactivity. Collecting the track information can take a while for folders with tens or hundreds of songs. MDC was never blocked. You could always do the all the selection operations you wanted on the available subset of songs and MDC displayed the total duration. I wanted to preserve this capability for MDC 2.0 where the user will be able to select and also play songs while more tracks are collected.

MP3 Duration Calculator 2.0

Version 2.0 is usually where you discover if your system is well designed. If your design had too many implicit assumptions about what it should support, then you will have a hard time accommodating the changes and new requirements for version 2.0. In general, there are two types of requirements that are difficult: Changes to existing functionality, and the addition of new functionality. In a well-designed application, adding new functionality should be straightforward and have little impact on the existing code. The complete source code and related files MDC Version 2.0 are available here.

In this case most of the requirements are for new functionality. The changes to existing functionality are making the total duration more prominent and changing the way songs are selected.

Make total duration more prominent

In Version 1.0 the total duration text box was at bottom of the window squished between the select/unselect buttons and the status text box; see Figure 1.

[Click image to view at full size]

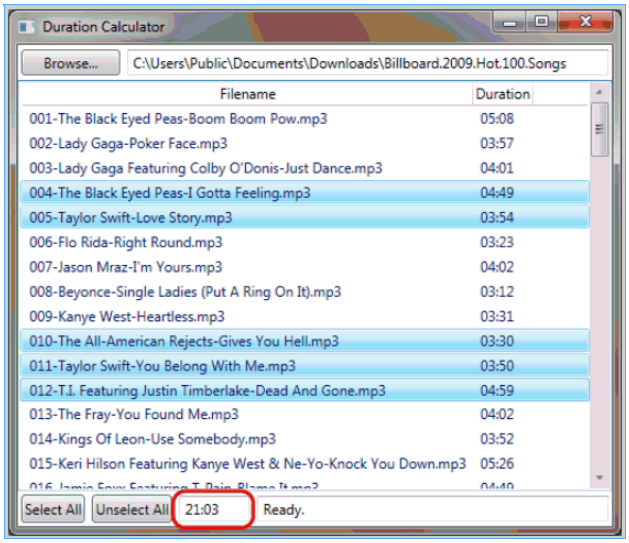


Figure 1

In Version 2.0, it moved to the top-right corner of the window and its text became blue and bold; see Figure 2.

[Click image to view at full size]

Figure 2

WPF was designed to support this kind of changes where you move elements around and modify some properties without any code changes (unless you think of XAML as code, which a perfectly valid point of view). All I had to do modify the XAML a bit. Here is the code for the bottom strip in Version 1.0:

```
<DockPanel
        DockPanel.Dock="Bottom"
        LastChildFill="True"
        Height="30"
        Margin="0" VerticalAlignment="Stretch"
  >
        <Button DockPanel.Dock="Left" Margin="3" Click="SelectAll_Click">
          Select All
        </Button>
        <Button DockPanel.Dock="Left" Margin="3" Click="UnselectAll_Click">
          Unselect All
        </Button>
        <TextBox DockPanel.Dock="Left" Height="22"
          Name="total" Width="60" Margin="3"
          Text="{Binding Path=Self, Converter={StaticResource DurationConverter}}">
        </TextBox>
        <TextBox Height="22" Name="status" MinWidth="100" Margin="3" />
  </DockPanel>
```

In Version 2.0 it is in the top strip. Note that the font size is 14 compared to 11 of all the other controls and the foreground color is Blue. That was trivial to do in the properties pane of Visual Studio.

```
<DockPanel
        DockPanel.Dock="Top"
        LastChildFill="True"
        Height="30"
        Margin="0" VerticalAlignment="Stretch"
      >
        <Button DockPanel.Dock="Left"   Height="22" Name="btnSelectFolder"       Width="84" Margin="3" Click="btnSelectFolder_Click">
          Browse...
        </Button>
        <TextBox DockPanel.Dock="Right" Margin="3" Height="22" Name="total"
```

```
        Width="60" Text="{Binding Path=Self, Converter={StaticResource DurationConverter}}"
        Foreground="Blue" TextDecorations="None" FontWeight="Bold" FontSize="14" Background="White" TextAlignment="Center" VerticalContentAlignment="Center"
      >
      </TextBox>
      <TextBox Height="22" Name="tbTargetFolder" MinWidth="258" Margin="3"
        TextChanged="tbTargetFolder_TextChanged"
      >
      </TextBox>
    </DockPanel>
```

The data binding that populates the **total** field doesn't really care where it is located and how big is the font and continues to work just fine.

Check boxes for selection (one song toggle)

It was straightforward to add the check boxes (see Figure 3).

[Click image to view at full size]

Figure 3

Remember that WPF and XAML are all about composition and nesting. But making the checkbox of each item in the list reflect the selection state is not trivial. This is a classic use case for two-way binding. If the check box is clicked and become checked you want the item to be selected (one way) and if the item is selected through some other mechanism like the user navigating with the arrow keys and hitting the space bar you want the check box to become checked (the other way). After some "binging" I discovered it requires a data template in XAML (of course you can do anything in code too). You define the data template in resources section of the main window and bind the **isChecked** property of the check box to the **isSelected** property of its containing **ListViewItem**. It sounds elegant, but the arcane incantations it requires can't be discovered easily just by reading the documentation. Here is the data template:

```
<DataTemplate x:Key="FirstCell">
      <StackPanel Orientation="Horizontal">
        <CheckBox IsChecked="{Binding Path=IsSelected,
          Mode=TwoWay,
          RelativeSource={RelativeSource FindAncestor,
          AncestorType={x:Type ListViewItem}}}"
        />
      </StackPanel>
        </DataTemplate>
```

Once the data template is defined it is simple to extend the list view to have a check box next to every item. All it takes is adding a new column to **GridView.Columns** (the first column):

```
  <GridView.Columns>
    <GridViewColumn
          CellTemplate="{StaticResource FirstCell}"
          Width="30"
    />
    <GridViewColumn
      Header="Filename"
      DisplayMemberBinding="{Binding Path=Name}"
      Width="Auto"
    />

   <GridViewColumn
     Header="Duration"
     DisplayMemberBinding="{Binding Path=Duration,
       Converter={StaticResource DurationConverter}}"
     Width="Auto"
   />
 </GridView.Columns>
```

That wasn't good enough though. I had my nifty little check boxes that were nicely synchronized with the selected items, but the basic problem remained if you select an item without holding control all your previous selections were still replaced by the new item. The check boxes didn't enforce the behavior I wanted (item select/unselect toggles just the current item). I tried various solutions with ever more desperation but I couldn't override fully the selection behavior of the list view (I tried hooking into the routed event processing and directly manipulating the selected items). Eventually, I noticed that the list view has a little property called **SelectionMode**. The MSDN documentation is a little confused about it and doesn't mention it, but it does mention that it's a property of a list box (a list view is derived from list box and just extends it with a view). Anyway, the **SelectionMode** is set to "extended" by default, which is the bad behavior, but it has also a value of "Multiple", which is exactly the behavior I was after. There is a "Single" value to that allows you to select at most one item at a time. So, I set the **SelectionMode** to "Multiple" and my work was done (including seamless integration with the check boxes).

Here is the XAML if you must:

```
<ListView
       MinHeight="223"
       Name="files"
       MinWidth="355"
       Background="White"
       SelectionChanged="files_SelectionChanged"
```

```
      SelectionMode="Multiple"
PreviewMouseRightButtonDown="files_PreviewMouseRightButtonDown"
     >
      ...
      </ListView>
```

Show the number of selected songs

This was another trivial request, but it required some code changes. The "status" text box display various status messages during the lifetime of the application (I probably should add a status bar for that). In theory, I could have a status object bound to the "status" text box, but that seemed like an overkill just for changing the contents of a text box every now and then (you would still need to update the status object from code). The number of selected songs should be displayed in two separate cases. The first one is when the collection of tracking info is complete. During the collection process the user may select already collected songs and view their total duration, but the status text box displays the progress of the collection process. Once the collection is complete it makes sense to display the current number of selected songs. The second case is of course when the number of selected songs is changed. The code is just two lines so I didn't even bother with a special method to update the status and just used the two-liner in both places. I use **String.Format()** to construct the message. Here is the first usage in **onTrackInfo()** when there are no more tracks:

```
  void onTrackInfo(TrackInfo ti)
 {
   if (ti == null)
   {
     // Update the status line
     var st = String.Format("{0}/{1} files are selected",
                            files.SelectedItems.Count,
                            _fileCount);
     status.Text = st;
      ...
   }
    ...
  }
```

The second usage is in the selectionChanged event handler of the files list view:

```
  void files_SelectionChanged(object sender, SelectionChangedEventArgs e)

 {
    ...
   // Update the status line
   var st = String.Format("{0}/{1} files are selected",
                          files.SelectedItems.Count,
                          _fileCount);
   status.Text = st;
    ...
  }
```

Play songs from within MDC

Okay, here is a feature with some substance. The media element is fully capable of playing tracks and we have one already in our main window. However, it is not a good idea to reuse the same media element used for collecting track info for playing tracks because I want the user to be able to play tracks during the collection process. This is a very important feature because the collection process can long seconds an even minutes if a folder contains hundreds of tracks. Another dedicated media element for playing songs solves the problem. Note that the media element I use for collecting track info doesn't interfere with the player media element because it never tries to play anything. It just opens each file, extracts the duration and closes the file.

I decided to go with a very spartan interface where you start and stop playing a song by right clicking with the mouse of a song. There is no way to pause or skip ahead and no way to see how far you are into the current song.

The act of playing a song doesn't change its selection state. You may play selected or unselected songs, during and after the collection process.

The player media element is an invisible element because it just plays audio in this case so it can be placed anywhere. I placed it next to the original media element in the **Grid** container of the main window:

```
<Grid>
    <MediaElement
        Height="0"
        Width="0"
        Name="mediaElement"
        LoadedBehavior="Manual"
     />
    <MediaElement
        Height="0"
        Width="0"
        Name="player"
        LoadedBehavior="Manual"
     />
     ...
   </Grid>
```

The code to enable play/stop is simple. It is all located in the event handler for the **PreviewMouseRightButtionDown**. WPF uses routed events. Routed events travel through the control hierarchy and any control along the way can handle the event and then it will not be passed further. There are two types of routed events: tunneling events and bubbling events. Tunneling events travel form parent to child and allow the parent to handle the event on behalf of many children or do some global handling and then let the children process them further. Bubbling events start at the child and bubble up to parents. The routed events travel in pairs. First, there is a tunneling event called **PreviewXXX** followed by a bubbling event called **XXX**. If some control handled the tunneling event the bubbling event will NOT be fired. So, the **PreviewMouseRightButtonDown** event is a tunneling event that's fired when the user clicks the right mouse button on the files list view. The main window is the parent of the list view so it gets first shot at handling it. Immediately sets the **Handled** property to **true** so the list view will never get a chance to process it. The reason is that the right mouse button click will toggle the selection if allowed to rich the list view. This is a behavior I want to suppress. Next, there is a somewhat complex check to find the parent of the original source of the right mouse click and see if it's a **GridViewRowPresenter**. If this check succeeds it means that the right click was indeed on one of songs in the list and not just a click in an empty area. Of course that's when we want to play or stop the current song. First, I need to figure out the path to the song file, which I retrieve from the clicked item. Note that the **Content** property of the clicked item is a **TrackInfo** object.

Then, I stop the player in case it was playing (it may be stopped but it causes no harm). If the player **Source** property is null (no song) or different then the current clicked item then I set the **Source** property to the new song and starts playing. Otherwise (clicked on the same playing stop) I set the **Source** to null for cleanup purposes.

```
    private void files_PreviewMouseRightButtonDown(object sender,
                                                   MouseButtonEventArgs e)
    {
      e.Handled = true;
      var x = ((FrameworkElement)e.OriginalSource).Parent;
      if (x is GridViewRowPresenter)
      {
        var ti = ((GridViewRowPresenter)x).Content as TrackInfo;
        string path = System.IO.Path.Combine(tbTargetFolder.Text, ti.Name);
        player.Stop();
        if (player.Source == null || player.Source.AbsolutePath != path)
```

```
        {
          player.Source = new System.Uri(path);
          player.Play();
        }
        else
        {
          player.Source = null;
        }
      }
    }
```

Copy selected songs to a new folder

Okay, so you play with MDC you select a bunch of songs out of your source directory and now you want to burn them. Liat preferred to have all the selected songs in a dedicated folder so she can rename songs, rearrange them and examine previous CDs.

With MDC 1.0 it was a grueling manual process. Liat fired up an explorer window next to MDC, selected all the files she selected previously in MDC, create a new folder and copy the files there. In MDC 2.0 this task can be done directly. I added a "Make CD..." button (see Figure 4) that opens a folder selection dialog that allows you to browse to an existing folder or create a new one. Once you select a folder, the selected items are copied to the target folder.

[Click image to view at full size]

Figure 4

As usual XAML makes it easy to add a button. The button is docked to the right side of the bottom panel, it has a tooltip and it is disabled initially (can't make a CD before some songs are selected). The **Click** event is hooked up to the **MakeCD_Click()** event handler:

```
  <Button
    Name="makeCD" DockPanel.Dock="Right" Margin="3"
    ToolTip="Copy selected songs to a new folder"
    Click="MakeCD_Click" IsEnabled="False"
  >

 Make CD...

 </Button>
```

There button is enabled/disabled based on the selection state of the files list. This can be done with data binding, but I opted in this case to use imperative code in the files**_SelectionChanged()** event handler that's being called whenever the selection changes. It is just one line and I handle in the same place enabling/disabling also the 'Select All' and 'Unselect All' buttons. The 'Select All' button is enabled if not all the items are selected. The 'Unselect All' and 'Make CD...' buttons are enabled if at least one item is selected.

```
  selectAll.IsEnabled = files.SelectedItems.Count < files.Items.Count;

 unselectAll.IsEnabled = files.SelectedItems.Count > 0;
  makeCD.IsEnabled = files.SelectedItems.Count > 0;
```

The **MakeCD_Click()** event handler implements two requested features: copying selected files to a folder and also exporting the song list to a CSV file.

The first step is to select the target folder. I reuse here there **_folderBrowserDialog** I use for selecting the source folder. In order to remember the source folder I save it in a temporary variable and restore it after the user selected a target folder. If the user selects the the same folder as the source folder MDC displays a message box and takes no action (no point in copying the files to the same folder, there are already there.

```
    private void MakeCD_Click(object sender, RoutedEventArgs e)
    {
      string savedPath = _folderBrowserDialog.SelectedPath;
      _folderBrowserDialog.ShowNewFolderButton = true;
      DialogResult r = _folderBrowserDialog.ShowDialog();
      if (r == System.Windows.Forms.DialogResult.OK)
      {
        string path = _folderBrowserDialog.SelectedPath;
        // check that the target folder is different than the source folder
        if (path == savedPath)
        {
          System.Windows.MessageBox.Show(
              "Target folder is the same as source folder. no action taken");
          return;
        }
        // restore selected path
        _folderBrowserDialog.SelectedPath = savedPath;
```

Now, it's time to actually copy all the selected files to the target folder. The .NET Framework provides in the **System.IO** namespace several classes to help with path manipulation and file operations. Many of these operations are implemented as static methods, which makes a lot of sense for operations like **Path.Combine** or **File.Copy** that don't have any state worth

remembering for future operations:

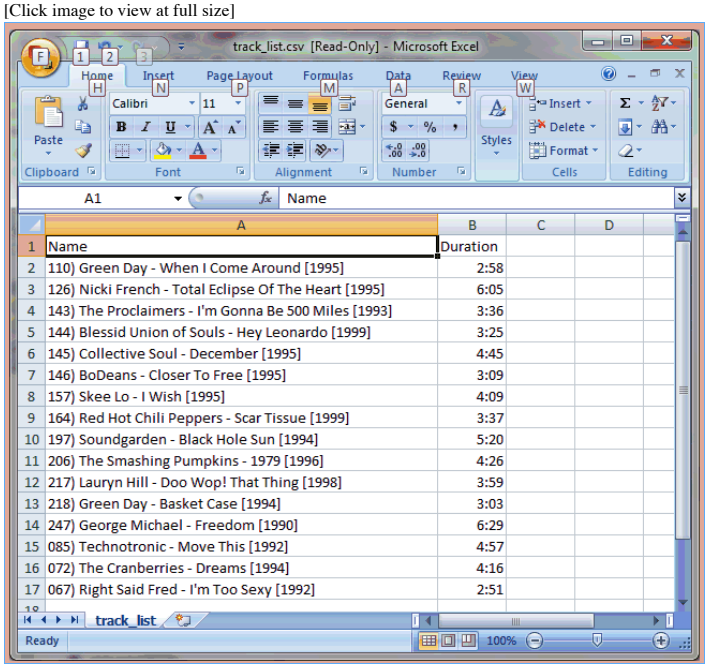
```
  // copy selected songs to target folder

 foreach (TrackInfo t in files.SelectedItems)
 {
   var src = System.IO.Path.Combine(savedPath, t.Name);
   var dst = System.IO.Path.Combine(path, t.Name);
   System.IO.File.Copy(src, dst);
 }
```

Exporting the song list to a CSV file

This is another feature Liat requested and it is also implemented in the **MakeCD_Click()** event handler immediately after copying the selected songs. The format is very simple. It is just a two-column list of **Name**, **Duration** for each selected item (see Figure 5). Liat can edit it later in Excel and add her class program next to the songs. It goes something like: go fast for 5 minutes, then accelerate for 8 more minutes and when you can't go any faster increase the resistance until you start crying.

[Click image to view at full size]



	A	B
1	Name	Duration
2	110) Green Day - When I Come Around [1995]	2:58
3	126) Nicki French - Total Eclipse Of The Heart [1995]	6:05
4	143) The Proclaimers - I'm Gonna Be 500 Miles [1993]	3:36
5	144) Blessid Union of Souls - Hey Leonardo [1999]	3:25
6	145) Collective Soul - December [1995]	4:45
7	146) BoDeans - Closer To Free [1995]	3:09
8	157) Skee Lo - I Wish [1995]	4:09
9	164) Red Hot Chili Peppers - Scar Tissue [1999]	3:37
10	197) Soundgarden - Black Hole Sun [1994]	5:20
11	206) The Smashing Pumpkins - 1979 [1996]	4:26
12	217) Lauryn Hill - Doo Wop! That Thing [1998]	3:59
13	218) Green Day - Basket Case [1994]	3:03
14	247) George Michael - Freedom [1990]	6:29
15	085) Technotronic - Move This [1992]	4:57
16	072) The Cranberries - Dreams [1994]	4:16
17	067) Right Said Fred - I'm Too Sexy [1992]	2:51

Figure 5

The Excel native file format is very complicated because it supports a lot of functionality. But, just for importing tabular data into Excel you can use a CSV (Comma-separated values) file. It is very simple to create such a file. You have to be careful to make sure your values don't contain a comma (I don't do it here).

The filename is track_list.csv and it is stored in the target folder together with all the songs. I utilize the 'using' statement that ensures correct use of **IDisposable** objects. These object have a **Dispose()** method that should be called to ensure proper cleanup and the **using** statement does it at the end of the block even if an exception is thrown. I like to think of it as the C# equivalent of the C++ RAII design pattern (http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization).

The **using** statement creates the track_list.csv text file, which is guarantied to be closed properly at the end of the block. Inside the **using** block the code writes a header "Name", "Duration" to the **sw StreamWriter** object and then iterates over the selected **TrackInfo** items and writes each item's name and duration with some formatting to make it look nicer.

```
  // Create CSV file with track list
  var filename = System.IO.Path.Combine(path, "track_list.csv");
 using (var sw = File.CreateText(filename))
 {
   sw.WriteLine("Name,Duration");
   foreach (TrackInfo t in files.SelectedItems)
   {
     var d = new DateTime(t.Duration.Ticks);
     var name = System.IO.Path.GetFileNameWithoutExtension(t.Name);
     sw.WriteLine(name + "," + d.ToString("mm:ss"));
   }
 }
```

Remember Last Folder

Programs that remember where the user left off are much nicer to use. The .NET Framework provides standard infrastructure to persist application settings and it's well integrated with Visual Studio (see Figure 6). I only use it to remember the last source folder, but it can be extended to store any number of things like the size and position of the main window, the selected items in the current session and more.

[Click image to view at full size]

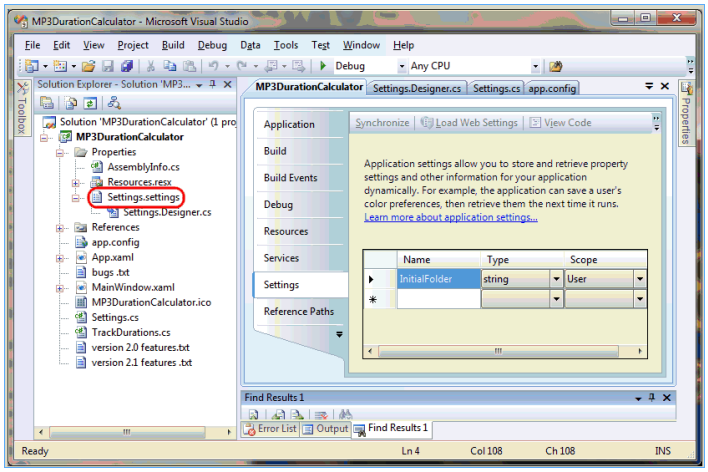


Figure 6

There are several other issues you have to take into account when thinking about persistent application settings such as write access and shared settings between all users vs. per-user settings.

Once you figured it out, it is pretty easy to use application settings (especially in Visual Studio). You define your properties in the settings tab. Visual Studio takes care of adding two files: Settings.Designer.cs and Settings.cs. These files together define the sealed Settings class (via partial classes in each file) under the **MP3DurationCalculator.Properties** namespace. The Settings.Designer.cs file contains a static instance of the default settings and get/set properties for each setting:

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.3603
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace MP3DurationCalculator.Properties {


 [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.VisualStudio.Editors.SettingsDesigner.SettingsSingleFileGenerator", "9.0.0.0")]

 internal sealed partial class Settings :
   global::System.Configuration.ApplicationSettingsBase {
    private static Settings defaultInstance = ((Settings)
(global::System.Configuration.ApplicationSettingsBase.Synchronized(
        new Settings())));
    public static Settings Default {
      get { return defaultInstance; }
[global::System.Configuration.UserScopedSettingAttribute()]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Configuration.DefaultSettingValueAttribute("")]
     public string InitialFolder {
      get { return ((string)(this["InitialFolder"])); }
       set { this["InitialFolder"] = value; }
    }
   }
}
```

The Settings.cs file contains optional event handlers you can implement to respond to a change in settings and to intercept the saving process:

```
namespace MP3DurationCalculator.Properties
{
  // This class allows you to handle specific events on the settings class:
  // The SettingChanging event is raised before a setting's value is changed.
 //  The PropertyChanged event is raised after a setting's value is changed.
 //  The SettingsLoaded event is raised after the setting values are loaded.
 //  The SettingsSaving event is raised before the setting values are saved.
 internal sealed partial class Settings
  {
   public Settings()
    {
      // To add event handlers for saving and changing settings, uncomment the
       //lines below:
      //
      // this.SettingChanging += this.SettingChangingEventHandler;
      //
      // this.SettingsSaving += this.SettingsSavingEventHandler;
      //
   }
   private void SettingChangingEventHandler(
      object sender,
      System.Configuration.SettingChangingEventArgs e)
    {
     // Add code to handle the SettingChangingEvent event here.
    }
    private void SettingsSavingEventHandler(
      object sender,        System.ComponentModel.CancelEventArgs e)
    {
     // Add code to handle the SettingsSaving event here.
   }
 }
}
```

MDC saves its settings in the **OnExit** method of the **App** class:

```
namespace MP3DurationCalculator
{
  public partial class App : Application
```

```
  {
    protected override void OnExit(ExitEventArgs e)
    {
      base.OnExit(e);
  MP3DurationCalculator.Properties.Settings.Default.Save();
    }
  }
}
```

When MDC is loaded the settings are loaded automatically and are available in the **MainWindow** constructor. If the initial folder doesn't exist than the user's MyMusic special folder is used as an initial folder:

```
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
       ...
      var initialFolder = Properties.Settings.Default.InitialFolder;
      if (!Directory.Exists(initialFolder))
        initialFolder =
             Environment.GetFolderPath(Environment.SpecialFolder.MyMusic);
           ...
    }
   ...
   }
```

Give Me More...

Liat was pleased with all the changes and new features, but with the food comes the appetite. Once the initial excitement over MDC 2.0 subsided Liat wanted more features:

- Make CD from multiple source folders (will require merging of track_list.csv files)
- A full-fledged media control bar: play, pause, stop
- A seek bar that allows skipping and also displays the current position of the song
- Maybe a little database to manage prepared CDs and mark songs that were used already.

I guess this is a good sign. There are only two types of programs: the programs that users complain about and the programs they don't use.

Conclusion

The .NET Framework and WPF are a superb platform for developing rich client applications. It is designed well and supports both enterprise-grade applications as well as small fun utilities like MDC. I was able to extend MDC with a significant amount of user-requested functionality with minimal changes to the XAML and the code. The separation between UI description and data binding in XAML and event handling in code works very well. I'm looking forward to explore WPF further in the future.