How To Implement A Modern Progress Dialog For WPF Applications

Jürgen Bäurle

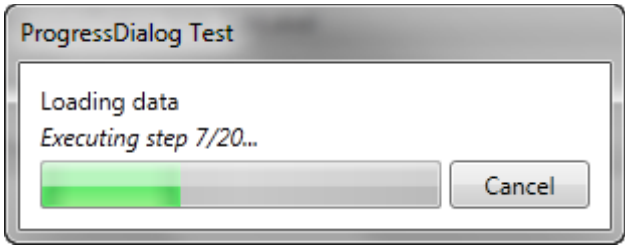
April 2011

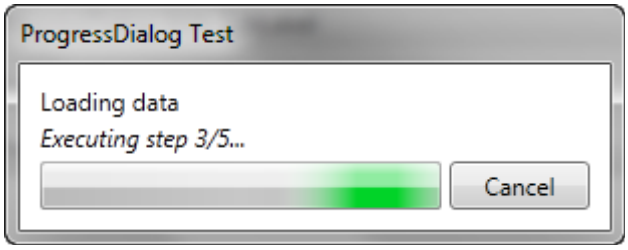
Parago Media GmbH & Co. KG

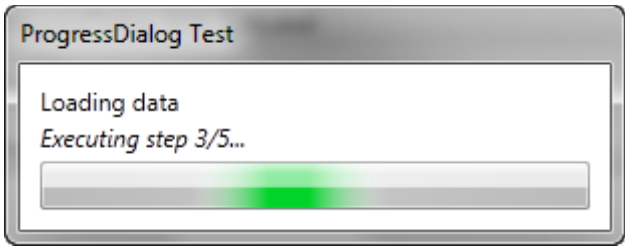
Introduction

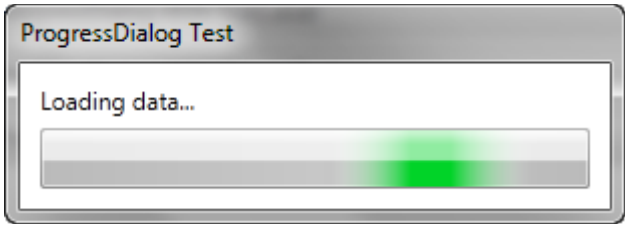
Developing a Windows Presentation Foundation (WPF) application requires sometimes to execute an asynchronous task with long-running execution steps, e.g. calling a web service. If those steps are triggered by user interactions, you want show a progress dialog to block the user interface and display detail step information. In some cases you also want to allow the user to stop the long-running task by clicking a Cancel button.

The following screenshots show some samples of progress dialogs implemented in this article:









This article will show how to implement such progress dialogs in WPF with C#. All of the above dialogs can be used with one implementation. The solution will also show how to hide the close button of a window, which is officially not supported by WPF.

The next section describes the usage of the progress dialog.

The Usage Of The Progress Dialog

The code snippet below shows how to use and display such a progress dialog:

```
ProgressDialogResult result = ProgressDialog.Execute(this, "Loading data...", () => {

  // TODO: Do your work here!

});

if(result.OperationFailed)
  MessageBox.Show("ProgressDialog failed.");
else
  MessageBox.Show("ProgressDialog successfully executed.");
```

The ProcessDialog class provides a number of static Execute methods (overrides) to easily setup a long-running task with a dialog window. The first parameter always defines the parent window in order to center the process dialog relative to the parent window. The second parameter is the text message displayed in all dialogs. The third parameter is the asynchronous method itself.

The fourth parameter allows to pass additional settings. Those dialog settings, represented by an instance of the ProcessDialogSettings class, define if the dialog shows a sub label, has a Cancel button or displays the progress bar itself in percentage or indeterminate. Predefined settings are also defined as static properties.

Using lambda expressions in C#, it is very comfortable to start a long-running task and displaying the dialog. You can also communicate with the progress dialog to report messages to the user by calling the Report method of the ProgressDialog class:

```
ProgressDialogResult result = ProgressDialog.Execute(this, "Loading data...", (bw) => {

  ProgressDialog.Report(bw, "Connecting to the server...");

  // TODO: Connect to the server

  ProgressDialog.Report(bw, "Reading metadata...");

  // TODO: Reading metadata

}, ProgressDialogSettings.WithSubLabel);
```

The two samples above did not show a Cancel button. The next code shows a progress dialog with a Cancel button including code to check if the long-running code must be cancelled:

```
int millisecondsTimeout = 1500;

ProgressDialogResult result = ProgressDialog.Execute(this, "Loading data", (bw, we) => {

  for(int i = 1; i <= 5; i++)
  {
    if(ProgressDialog.ReportWithCancellationCheck(bw, we, "Executing step {0}/5...", i))
      return;

    Thread.Sleep(millisecondsTimeout);
  }

  // So this check in order to avoid default processing after the Cancel button has been
  // pressed. This call will set the Cancelled flag on the result structure.
  ProgressDialog.CheckForPendingCancellation(bw, we);

}, ProgressDialogSettings.WithSubLabelAndCancel);

if(result.Cancelled)
  MessageBox.Show("ProgressDialog cancelled.");
else if(result.OperationFailed)
  MessageBox.Show("ProgressDialog failed.");
else
  MessageBox.Show("ProgressDialog successfully executed.");
```

Calling the ReportWithCancellationCheck method will check for a pending cancellation request (from the UI) and will may set the Cancel property of the DoWorkEventArgs class passed from the underlying BackgroundWorker object to True. Otherwise the method will display the message and continue processing.

The Exceute method of the ProgressDialog class will return an instance of the ProgressDialogResult class that returns the status of the execution. Thrown exceptions in the task method will be stored in the Error property if the OperationFailed property is set to True. For more samples see the Visual Studio 2010 solution you can download from my homepage.

Implementation

The ProgressDialog window contains the main application logic. The above described static Execute methods will internally call the ExecuteInternal method to create an instance of the ProgressDialog window passing and setting all necessary values. Then the Execute method with the asynchronous method as parameter is called:

```
internal static ProgressDialogResult ExecuteInternal(Window owner, string label,
  object operation, ProgressDialogSettings settings)
{ {
  ProgressDialog dialog = new ProgressDialog(settings);
  dialog.Owner = owner;

  if(!string.IsNullOrEmpty(label))
    dialog.Label = label;

  return dialog.Execute(operation);
}
```

The operation method can be a delegate of the following type:

- Action
- Action<BackgroundWorker>
- Action<BackgroundWorker, DoWorkEventArgs>
- Func<object>
- Func<BackgroundWorker, object>
- Func<BackgroundWorker, DoWorkEventArgs, object>

The Func types can return a value that will be used to set the Result property of the ProgressDialogResult class.

The Excute method is implemented as follows:

```
internal ProgressDialogResult Execute(object operation)
{ {
  if(operation == null)
    throw new ArgumentNullException("operation");

  ProgressDialogResult result = null;

  _isBusy = true;

  _worker = new BackgroundWorker();
  _worker.WorkerReportsProgress = true;
  _worker.WorkerSupportsCancellation = true;

  _worker.DoWork +=
    (s, e) => {
      if(operation is Action)
        ((Action)operation)();
      else if(operation is Action<BackgroundWorker>)
        ((Action<BackgroundWorker>)operation)(s as BackgroundWorker);
      else if(operation is Action<BackgroundWorker, DoWorkEventArgs>)
        ((Action<BackgroundWorker, DoWorkEventArgs>)operation)(s as BackgroundWorker, e);
      else if(operation is Func<object>)
        e.Result = ((Func<object>)operation)();
      else if(operation is Func<BackgroundWorker, object>)
        e.Result = ((Func<BackgroundWorker, object>)operation)(s as BackgroundWorker);
      else if(operation is Func<BackgroundWorker, DoWorkEventArgs, object>)
        e.Result = ((Func<BackgroundWorker, DoWorkEventArgs, object>)operation)(s as
BackgroundWorker, e);
      else
        throw new InvalidOperationException("Operation type is not supoorted");
    };

  _worker.RunWorkerCompleted +=
    (s, e) => {
      result = new ProgressDialogResult(e);
      Dispatcher.BeginInvoke(DispatcherPriority.Send, (SendOrPostCallback)delegate {
        _isBusy = false;
```

```
        Close();
      }, null);
    };

  _worker.ProgressChanged +=
    (s, e) => {
      if(!_worker.CancellationPending)
      {
        SubLabel = (e.UserState as string) ?? string.Empty;
        ProgressBar.Value = e.ProgressPercentage;
      }
    };

  _worker.RunWorkerAsync();

  ShowDialog();

  return result;
}
```

The ProgressDialog class is using internally a BackgroundWorker object to handle the asynchronous execution of the task or operation method.

For more details of the implementation see the source code.

Summary

Using a progress dialog to handle the UI for long-running task can be implemented quite simple. The effect is an impressive and clean user communication experience.

Contact Information

If you have any feedback or suggestions, please feel free to contact me:

Jürgen Bäurle
jbaurle@parago.de
http://www.parago.de/jbaurle

Parago Media GmbH & Co. KG
Im Wengert 3 | 71336 Waiblingen, Germany | Phone +49.7146.861803 | Internet http://www.parago.de