Progress-O-Doom

BoneSoft

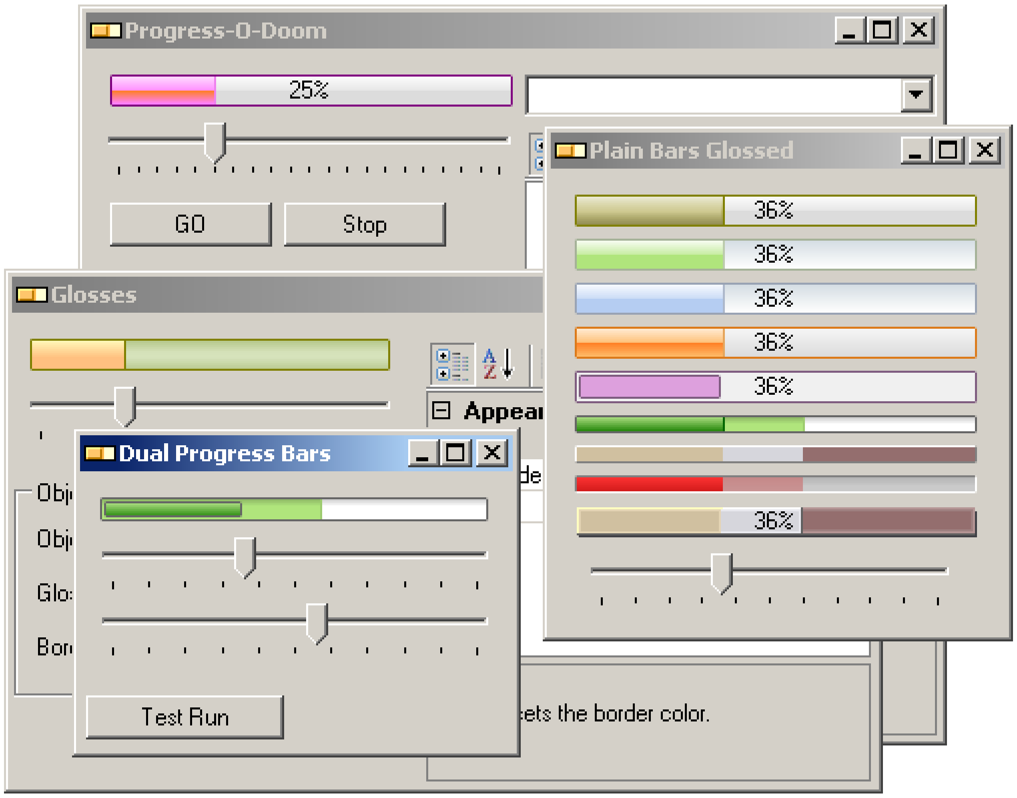
21 Jan 2009 CPOL 15 min read 380.2K 13.5K 549 120

Rate me: 4.97/5 (251 votes)

A set of pluggable progress bar components

Download demo - 42.91 KB

Download source - 103.96 KB



Introduction

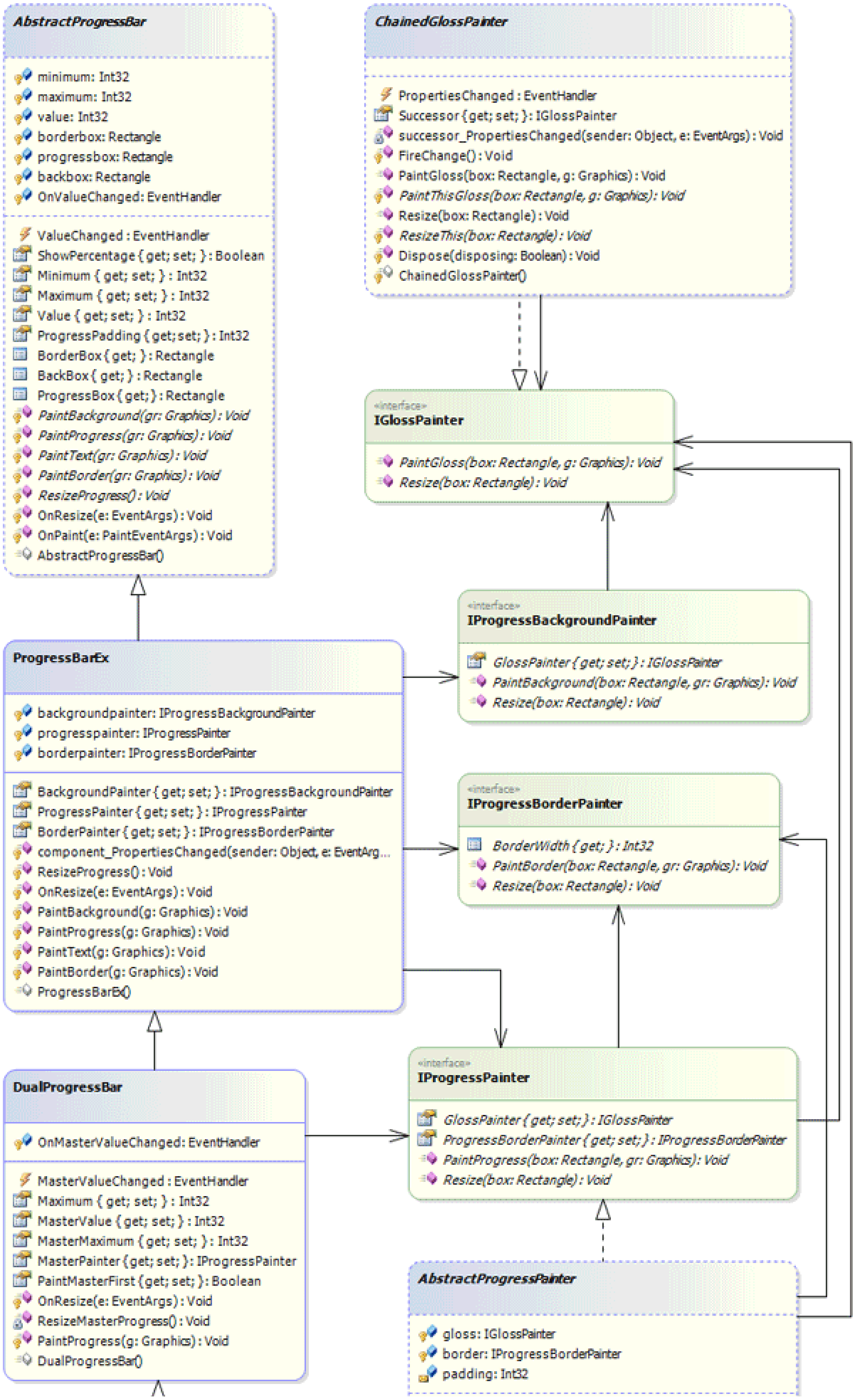
For reasons I can't explain, I have an irrational fascination with progress bars. Over the years, I'd see a progress bar I liked, and would try to recreate it in C# for my own use. This resulted in several concrete implementations that all shared a relatively small set of functionality in a single parent class. Then, not long ago, I saw several that I wanted to implement, but I wasn't interested in writing them from the ground up. However, I noticed that they all shared a lot of characteristics. Then, it finally dawned on me that I could build almost all of those examples and many more with a set of pluggable components that would paint certain parts of the progress bars. This project is the result of that revelation.

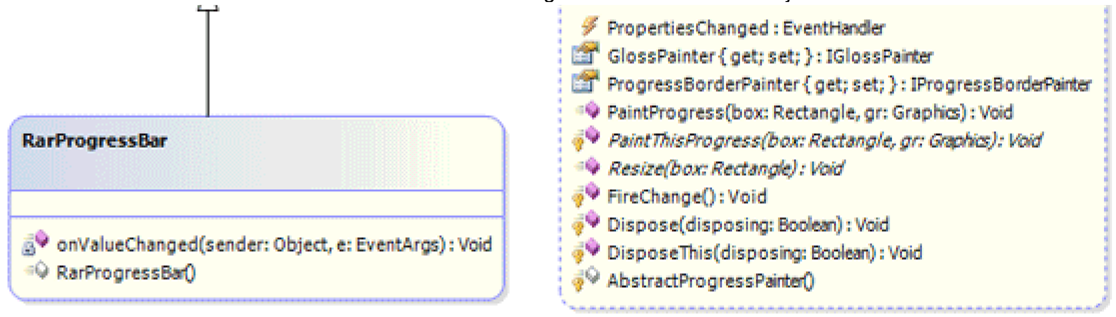
Component Structure

The idea, basically, is this... There are two concrete implementations for progress bars (plus, one specifically to recreate the WinRar dual progress bar), each building on the more general implementation. Those implementations use an instance of `IProgressBackgroundPainter` to paint a background, an instance of `IProgressPainter`, and an instance of `IProgressBorderPainter` to paint the border. Then, optionally, you can chain any number of `IGlossPainter` instances together to doctor up your progress and/or background painters. `IProgressPainter` can also use its own `IProgressBorderPainter`. `IProgressPainter` and `IGlossPainter` both have `abstract` implementations that provide some basic functionality. `ChainedGlossPainter`, the `abstract` implementation of `IGlossPainter`, is a decorator that allows you to chain `IGlossPainter`s together for multiple effects.

By making all of these functional parts `Component`s, you can drop them into your designer, and set their relationships and properties at design time. This makes it easy to experiment with different possibilities to find what best suits your needs and tastes.

Shown below is a diagram of the structure of the progress bar implementations, the painter interfaces, and their abstract implementations:

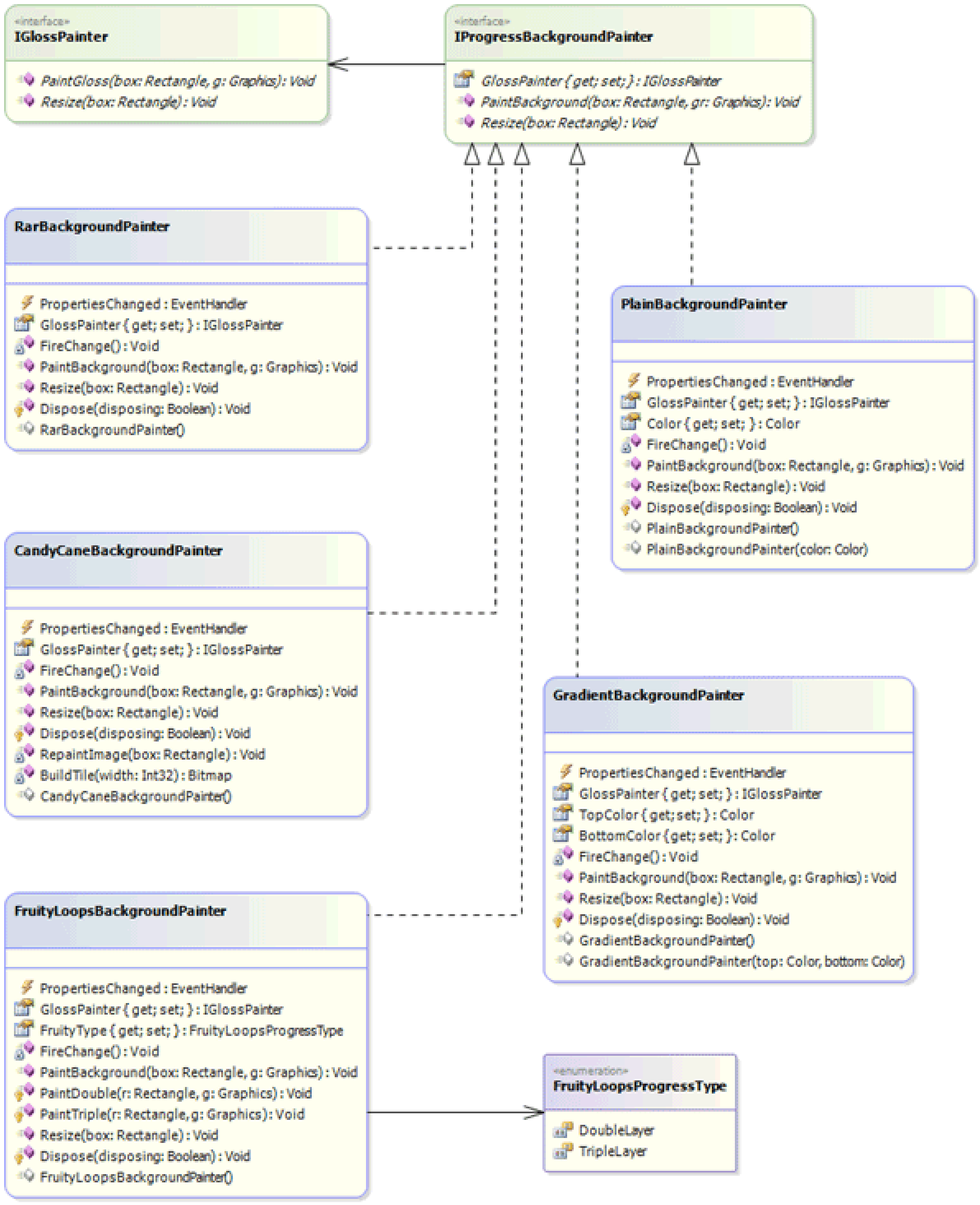

AbstractProgressBar
minimum: Int32
maximum: Int32
value: Int32
borderbox: Rectangle
progressbox: Rectangle
backbox: Rectangle
OnValueChanged: EventHandler
ValueChanged : EventHandler
ShowPercentage { get; set; } : Boolean
Minimum { get; set; } : Int32
Maximum { get; set; } : Int32
Value { get; set; } : Int32
ProgressPadding { get; set; } : Int32
BorderBox { get; } : Rectangle
BackBox { get; } : Rectangle
ProgressBox { get; } : Rectangle
PaintBackground(gr: Graphics) : Void
PaintProgress(gr: Graphics) : Void
PaintText(gr: Graphics) : Void
PaintBorder(gr: Graphics) : Void
ResizeProgress() : Void
OnResize(e: EventArgs) : Void
OnPaint(e: PaintEventArgs) : Void
AbstractProgressBar()
ChainedGlossPainter
PropertiesChanged : EventHandler
Successor { get; set; } : IGlossPainter
successor_PropertiesChanged(sender: Object, e: EventArgs) : Void
FireChange() : Void
PaintGloss(box: Rectangle, g: Graphics) : Void
PaintThisGloss(box: Rectangle, g: Graphics) : Void
Resize(box: Rectangle) : Void
ResizeThis(box: Rectangle) : Void
Dispose(disposing: Boolean) : Void
ChainedGlossPainter()
«interface»
IGlossPainter
PaintGloss(box: Rectangle, g: Graphics) : Void
Resize(box: Rectangle) : Void
«interface»
IProgressBackgroundPainter
GlossPainter { get; set; } : IGlossPainter
PaintBackground(box: Rectangle, gr: Graphics) : Void
Resize(box: Rectangle) : Void
ProgressBarEx
backgroundpainter: IProgressBackgroundPainter
progresspainter: IProgressPainter
borderpainter: IProgressBorderPainter
BackgroundPainter { get; set; } : IProgressBackgroundPainter
ProgressPainter { get; set; } : IProgressPainter
BorderPainter { get; set; } : IProgressBorderPainter
component_PropertiesChanged(sender: Object, e: EventArg...
ResizeProgress() : Void
OnResize(e: EventArgs) : Void
PaintBackground(g: Graphics) : Void
PaintProgress(g: Graphics) : Void
PaintText(g: Graphics) : Void
PaintBorder(g: Graphics) : Void
ProgressBarEx()
«interface»
IProgressBorderPainter
BorderWidth { get; } : Int32
PaintBorder(box: Rectangle, gr: Graphics) : Void
Resize(box: Rectangle) : Void
DualProgressBar
OnMasterValueChanged: EventHandler
MasterValueChanged : EventHandler
Maximum { get; set; } : Int32
MasterValue { get; set; } : Int32
MasterMaximum { get; set; } : Int32
MasterPainter { get; set; } : IProgressPainter
PaintMasterFirst { get; set; } : Boolean
OnResize(e: EventArgs) : Void
ResizeMasterProgress() : Void
PaintProgress(g: Graphics) : Void
DualProgressBar()
«interface»
IProgressPainter
GlossPainter { get; set; } : IGlossPainter
ProgressBorderPainter { get; set; } : IProgressBorderPainter
PaintProgress(box: Rectangle, gr: Graphics) : Void
Resize(box: Rectangle) : Void
AbstractProgressPainter
gloss: IGlossPainter
border: IProgressBorderPainter
padding: Int32


RarProgressBar
onValueChanged(sender: Object, e: EventArgs) : Void
RarProgressBar()
PropertiesChanged : EventHandler
GlossPainter { get; set; } : IGlossPainter
ProgressBorderPainter { get; set; } : IProgressBorderPainter
PaintProgress(box: Rectangle, gr: Graphics) : Void
PaintThisProgress(box: Rectangle, gr: Graphics): Void
Resize(box: Rectangle) : Void
FireChange() : Void
Dispose(disposing: Boolean) : Void
DisposeThis(disposing: Boolean) : Void
AbstractProgressPainter()

All of the progress bar implementations have the basic properties you would expect to see (i.e., `Maximum`, `Minimum`, `Value`). They also have `ShowPercentage`, which is a boolean property that determines whether or not to paint the percentage text on the bar, and `ProgressPadding` which is an `int` representing the number of pixels to space between the progress painter and the borders. `DualProgressBar` adds `MasterMaximum`, `MasterValue`, `MasterPainter`, and `PaintMasterFirst` for controlling the master progress. The `DualProgressBar` was designed to function like the WinRar progress bar, which shows a silver progress for the overall progress, and a gold progress (always smaller) superimposed to represent the overall compression ratio. However, this bar can be used to show individual and overall progress, especially when the non-master progress has padding so that the master progress can be seen under it.

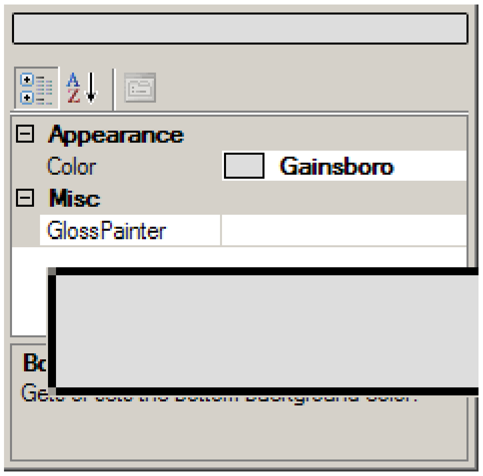
Background Painters

I currently have five implementations of `IProgressBackgroundPainter`, three of which were built for specific progress bars (WinRar, FruityLoops, and Candy Cane).


«interface»
IGlossPainter
PaintGloss(box: Rectangle, g: Graphics): Void
Resize(box: Rectangle): Void
«interface»
IProgressBackgroundPainter
GlossPainter { get; set; } : IGlossPainter
PaintBackground(box: Rectangle, gr: Graphics): Void
Resize(box: Rectangle): Void
RarBackgroundPainter
PropertiesChanged : EventHandler
GlossPainter { get; set; } : IGlossPainter
FireChange() : Void
PaintBackground(box: Rectangle, g: Graphics) : Void
Resize(box: Rectangle) : Void
Dispose(disposing: Boolean) : Void
RarBackgroundPainter()
PlainBackgroundPainter
PropertiesChanged : EventHandler
GlossPainter { get; set; } : IGlossPainter
Color { get; set; } : Color
FireChange() : Void
PaintBackground(box: Rectangle, g: Graphics) : Void
Resize(box: Rectangle) : Void
Dispose(disposing: Boolean) : Void
PlainBackgroundPainter()
PlainBackgroundPainter(color: Color)
CandyCaneBackgroundPainter
PropertiesChanged : EventHandler
GlossPainter { get; set; } : IGlossPainter
FireChange() : Void
PaintBackground(box: Rectangle, g: Graphics) : Void
Resize(box: Rectangle) : Void
Dispose(disposing: Boolean) : Void
RepaintImage(box: Rectangle) : Void
BuildTile(width: Int32) : Bitmap
CandyCaneBackgroundPainter()
GradientBackgroundPainter
PropertiesChanged : EventHandler
GlossPainter { get; set; } : IGlossPainter
TopColor { get; set; } : Color
BottomColor { get; set; } : Color
FireChange() : Void
PaintBackground(box: Rectangle, g: Graphics) : Void
Resize(box: Rectangle) : Void
Dispose(disposing: Boolean) : Void
GradientBackgroundPainter()
GradientBackgroundPainter(top: Color, bottom: Color)
FruityLoopsBackgroundPainter
PropertiesChanged : EventHandler
GlossPainter { get; set; } : IGlossPainter
FruityType { get; set; } : FruityLoopsProgressType
FireChange() : Void
PaintBackground(box: Rectangle, g: Graphics) : Void
PaintDouble(r: Rectangle, g: Graphics) : Void
PaintTriple(r: Rectangle, g: Graphics): Void
Resize(box: Rectangle) : Void
Dispose(disposing: Boolean) : Void
FruityLoopsBackgroundPainter()
«enumeration»
FruityLoopsProgressType
DoubleLayer
TripleLayer

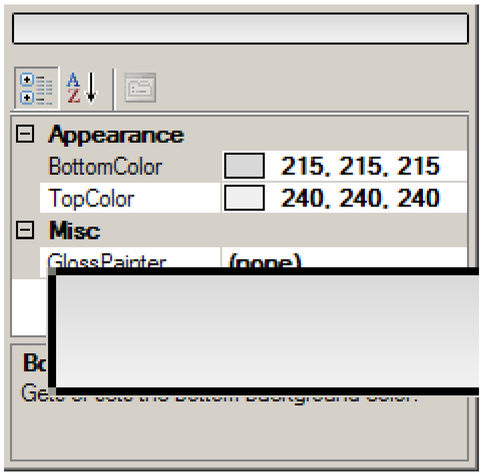
The Plain Background Painter

`PlainBackgroundPainter` just paints the background with a single color. I use this background painter the most in combination with `IGlossPainter`s. On its own, it's far from interesting, but it's the perfect base for adding glosses to.


Appearance
Color Gainsboro
Misc
GlossPainter

The Gradient Background Painter

GradientBackgroundPainter was built before the glosses were conceived of. It allows you to specify a top and bottom color, and fills the background with a gradient between those two colors. The same effect can be attained with the PlainBackgroundPainter and the GradientGlossPainter. So, this background painter is really just a convenient shortcut.


Appearance
BottomColor 215, 215, 215
TopColor 240, 240, 240
Misc
GlossPainter (none)

The Candy Cane Background Painter

CandyCaneBackgroundPainter was created from an existing bar in my old collection of bars. It can probably also be recreated from the PlainBackgroundPainter and glosses. It does match the CandyCaneProgressPainter, but it also works well with virtually any other progress painter.

The FruityLoops Background Painter

FruityLoopsBackgroundPainter was also built from an old bar I had that was based on the progress bars in the FruityLoops application. It has a matching progress painter. This background painter isn't very configurable, it was built for a specific set to emulate the FruityLoops bars.

The WinRar Background Painter

`RarBackgroundPainter` was also built from an old bar based on the WinRar progress bars. It also has matching progress and border painters, which are used by the `RarProgressBar`. Because it was built specifically to emulate the WinRar bars, there isn't much that's configurable; you can't change the color of it, for instance.

Progress Painters

There are nine implementations of `IProgressPainter`, most of which were based on my old collection of progress bars. They can all be decorated with glosses, but the plain implementation was designed to be the common canvas for glosses.

The Plain Progress Painter

`PlainProgressPainter`, like the plain background painter, only fills the progress with a solid color, which makes it a great starting point for glosses. In addition to the `Color` property, you can set a color to the `LeadingEdge` property. Without setting a `ProgressPadding` value on the progress bar, you can get the same effect by setting the plain progress painter's `ProgressBorderPainter` property to an instance of `IProgressBorderPainter`.

The Beveled Painter

`BevelledProgressPainter` also fills the progress with a single color, but it also bevels the progress with a two pixel border based on the base color set. Since the `IProgressPainter` interface defines it, all progress painters can have an `IProgressBorderPainter` set to its `ProgressBorderPainter` property.

The Beveled Gradient Painter

`BevelledGradientProgressPainter` is basically the same as the `BevelledProgressPainter` except that it has the `MinColor` and `MaxColor` properties that it uses to fill the progress with a gradient.

The Metal Progress Painter

`MetalProgressPainter` is a subtle but elegant custom design that builds from a single base color.

The Barber Pole

`BarberPoleProgressPainter` is another custom design. This is one of the two progress painters that have a problem with the `ProgressPadding` property of all bars. When the control is resized, the painter rebuilds an image that fits the entire bar so that value changes only require the image to move. This is a limitation that I'm not overly concerned with at the moment.

A Java Painter

`JavaProgressPainter` is a custom design based on a progress bar I saw in a Java app once upon a time. It's always padded, so setting padding will just alter it further, though you could also give it a negative padding to force it to fill the progress space.

The Candy Cane

`CandyCaneProgressPainter` shares the same strategy, and therefore the same limitations, as the `BarberPoleProgressPainter`. It builds a partially transparent design based on a single base color.

The FruityLoops Painter

`FruityLoopsProgressPainter` only has one configurable property of its own, the `FruityType` property which has two possible values, `DoubleLayer` and `TripleLayer`. It pads well, and can be glossed and/or bordered. But, it works best with the `FruityLoopsBackgroundPainter` it was designed to match.

The WinRar Progress Painter

`RarProgressPainter` is a very simple progress painter. Though designed to work with the `RarBackgroundPainter` and the `RarBorderPainter`, it works well with other implementations because it is so simple. The `ProgressType` property corresponds to the two colors WinRar uses, `Silver` and `Gold`. The `ShowEdge` property draws a leading edge specific to the WinRar painter.

Border Painters

Border painters can be used to border the entire progress bar, and they can also be used to border a progress painter. The `IProgressBorderPainter` interface requires implementations to expose a `BorderWidth` property so that the progress bars using it can determine what size the progress `Rectangle` should be. As with the other components, it has methods to paint and resize itself.

The Plain Border Painter

With the `PlainBorderPainter`, you can specify a color for a flat border, or use its implementation of `Sunken` or `Raised`. The `RoundedCorners` property just makes the corner pixels partially transparent, which gives it a subtle rounding.

The Styled Border Painter

The `StyledBorderPainter` uses the `System.Windows.Forms.Border3DStyle` values with the `ControlPaint.DrawBorder3D()` method to paint a border. As you can see with some of the examples below, some of these `Border3DStyle` borders have a single pixel border width on the left and two pixel widths for the other sides. Since the `BorderWidth` property on `IProgressBorderPainter` only allows you to specify a single border width, this border painter has some undesirable side effects, but they are subtle, and in most cases hardly noticeable.

The WinRar Border Painter

The `RarBorderPainter` was designed specifically for use with the `RarProgressPainter` and `RarBackgroundPainter`, but it does give a nice raised outer look to the bars.

Gloss Painters

The glosses were designed to allow you to put various gradients with varying transparency on top of progress bars and backgrounds. There are four gloss implementations, all of which have a `Successor` property that allows you to chain them together for multiple effects.

Flat Gloss

The `FlatGlossPainter` just blankets an area with a color. You can control that area by setting the `Style` (`Top` or `Bottom`) property and the `PercentageCovered` property. It allows you to specify a color and an alpha value for that color.

Gradient Gloss

`GradientGlossPainter` gives you a gradient over a percentage of the top or bottom, or over the entire surface. It uses the base color combined with the `AlphaHigh` and `AlphaLow` properties for the gradient. You can also control the angle. The example uses an angle of 90; 270 would invert what's shown.

Middle Gloss

The `MiddleGlossPainter` gradients from the middle out. Again, through the `Style` property, you can gloss the top, bottom, or full surface. Like the `GradientGlossPainter`, it uses one color and two alpha values for the gradient.

Round Gloss

The `RoundGlossPainter` gradients off the top and/or bottom edges. Like the `GradientGlossPainter` and the `MiddleGlossPainter`, it uses one color and two alpha values for the gradient. Instead of a percentage, like the other glosses, this gloss has a `TaperHeight` property that allows you to set the number of pixels from the edge to the gradient. In the example below, the `TaperHeight` is set to 8.

Some Code

Most of the magic is just simple GDI+ stuff. The real strength here is in the structure of how components are arranged and how they control the `Rectangle`s passed to the other components. However, there are a couple of things that might be interesting to note.

The PropertiesChanged Events

The following was used to facilitate access to the `PropertiesChanged` event to avoid registering a listener more than once. Perhaps a little lazy on my part, but it avoids issues in the long run. With this many components talking to each other, I wanted to ensure I wasn't repeating redraws or other function calls unnecessarily.

C#

```
private EventHandler onPropertiesChanged;
/// <summary></summary>
public event EventHandler PropertiesChanged {
    add {
        if (onPropertiesChanged != null) {
            foreach (Delegate d in onPropertiesChanged.GetInvocationList()) {
                if (object.ReferenceEquals(d, value)) { return; }
            }
        }
        onPropertiesChanged = (EventHandler)Delegate.Combine(onPropertiesChanged, value);
    }
    remove { onPropertiesChanged = (EventHandler)Delegate.Remove
                (onPropertiesChanged, value); }
}
```

Using the DualProgressBar

The concept of the dual progress bar is somewhat different from what most people are used to working with. To help understand what it's doing, the following code is what was used in the DualTests form of the demo app. It sets the master max to 10,000, and the max to 2000 (because I knew ahead of time that I wanted five iterations). Then, as it loops five times, it resets the `Value` property, and increments both the `Value` and the `MasterValue` properties. This example shows how the dual progress bar can be used to display an overall progress with individual progresses.

C#

```
private bool go = false;
private void button1_Click(object sender, EventArgs e) {
    go = true;
    dualProgressBar1.Value = 0;
    dualProgressBar1.MasterValue = 0;
    dualProgressBar1.Maximum = 2000;
    dualProgressBar1.MasterMaximum = 10000;
    for (int i = 0; i < 5; i++) {
        if (!go) { break; }
        dualProgressBar1.Value = 0;
        for (int j = 0; j < 2000; j++) {
            if (!go) { break; }
            dualProgressBar1.Value = j;
            dualProgressBar1.MasterValue++;
            Application.DoEvents();
        }
    }
    dualProgressBar1.Value = 0;
    dualProgressBar1.MasterValue = 0;
}
```

Though the master progress shares the `Minimum` value of the non-master progress, through the `MasterMaximum` and `MasterValue` properties, it behaves just like any other progress.

Something to be Aware of

`IGlossPainter`'s `Successor` property does test to ensure that you're not setting a gloss as its own successor, but it does not test for a larger reference loop. Since each gloss asks its successor (if not `null`) to repaint itself, a loop in gloss references will result in an infinite repaint loop. Since this happens at design time, Visual Studio will die if you do this. If anyone has a suggestion on testing for referential loops like this, I'd love to hear it.

C#

```
public abstract class ChainedGlossPainter : Component, IGlossPainter, IDisposable {
    private IGlossPainter successor = null;

    /// <summary></summary>

    public IGlossPainter Successor {
        get { return successor; }
        set {
            if (object.ReferenceEquals(this, value)) {
                throw new ArgumentException("Gloss cannot be it's own successor,
                    an infinite loop will result");
            }
            successor = value;
            if (successor != null) {
                successor.PropertiesChanged +=
        new EventHandler(successor_PropertiesChanged);
            }
            FireChange();
        }
    }
    ...
```

Using the Components

First, you'll want to drop one of the progress bars on your form (`ProgressBarEx`, `DualProgressBar`, or `RarProgressBar`), then also add a background painter, border painter, and progress painter to your form, along with any gloss instances you might need.

Once you've added the components, set the `ProgressPainter`, `BorderPainter`, and `BackgroundPainter` properties of your progress bar to the appropriate components.

Then, if you've added glosses to the form, you can set the `GlossPainter` properties of your other painters. Each gloss painter can be used on more than one component too, so the background and progress painters could share a gloss.

And, if you want to chain glosses together, set each of the `Successor` property of each one you wish to chain, to the next in the chain. Just remember that they are painted in the order they are chained. Also, remember that you must avoid having a chain that results in a gloss referencing itself anywhere in its chain, which will result in an infinite redraw loop that will kill Visual Studio.

Conclusion

This project was done for fun, and I have very little doubt that there are still some bugs in there. I'm no expert in GDI+ or custom controls, so there's likely plenty of room for improvement here, and I would love to hear suggestions from those of you who have more expertise than me (or even from those who haven't :). But, it is functional, and it makes for some nice, versatile looking bars. Below are some examples created during development and testing. As you can see, there are a lot of possibilities with these components.

All of the examples on the left were done with plain controls and gloss. The examples on the right were done with the specific painters, with the exception of the non-WinRar dual bars, which were all done with plain painters and gloss.

Future Plans

- I'd like to add something to facilitate sectioned progress bars, like the standard Windows bars.
- I'd also like to provide more options for rounding borders and bars.
- And, of course, I'd also like to find solutions to the limitations mentioned above (variable width borders, the padding issue with image based bars, etc.).

History

- **12/19/2008**

  Initial release

- **12/23/2008**

- As suggested by Acshi, the `Successor` accessor has been updated in `ChainedGlossPainter` to better detect circular references.
- Also, this update includes functionality for Marquee bars. The `AbstractProgressBar` class has four new properties: `ProgressType`, `MarqueeSpeed`, `MarqueeStep` & `MarqueePercentage` (I'll try to update diagrams at a later time). `ProgressType` is an `enum` with the values `Smooth` (normal), `MarqueeWrap` (runs off frame to the right, then comes back in frame from left), `MarqueeBounce` (bounces between right and left) & `MarqueeBounceDeep` (also bounces, but goes out of frame before rebounding). Speed is an int for milliseconds between updates. Step is an int for the number of pixels to move between updates. And `MarqueePercentage` determines the width of the marquee bar.
- `AbstractProgressBar` also includes three `abstract` methods for marquee functionality, which are implemented in `ProgressBarEx`: `MarqueeStart()`, `MarqueePause()` & `MarqueeStop()`. I've never liked how the default progress bar handles marquee operations, and have seen many people confused by and/or complaining about it. So I decided to use methods for starting and stopping animation.
- Since `DualProgressBar` extends `ProgressBarEx`, it inherits this functionality. And you can also still use the master progress normally. This might come in handy if you have long operations where you want to show the actual progress, but still have animation during the wait. This scenation is simulated in the new Marquee Test form. Just be sure to set the `PaintMasterFirst` property to `true`.

  I'm not sure that I'm completely satisfied with this implementation, but it does establish a usable interface for this functionality, and it seems to work well enough (with the notable exceptions of the `BarberPoleProgressPainter` & the `CandyCaneProgressPainter`, which don't behave correctly due to previously mentioned limitations).

- **01/18/2009**

  - The `ProgressType` enum has a new value `Animated`.
  - The `IAnimatedProgressPainter` interface was added.

- The `ProgressBarEx.ProgressPainter` property now does some validation when being set. The `ProgressType.Animated` value is only valid when the `ProgressBarEx.ProgressPainter` is an instance of `IAnimatedProgressPainter`. If an non-animated progress painter is assigned to `ProgressBarEx.ProgressPainter`, and it currently has the `ProgressType.Animated` value, the `ProgressType` will be changed to `ProgressType.Smooth`.
- If the `ProgressBarEx.ProgressType` is set to `ProgressType.Animated` while `ProgressBarEx.ProgressPainter` is set to a non-animated progress painter, an `ArgumentException` will be thrown.
- `ProgressBarEx` has two new methods, `StartAnimation()` and `StopAnimation()`. It manages an internal timer that it uses (with the `MarqueeSpeed`) to update animation. When animating, it sets the `Animating` property of it's animated progress painter, then it's timer callback method just invalidates the control to be repainted (the animated implementations update an internal variable to keep track of framing, if it's `Animating` property is `false`, it does not use the frame counter).

Added Classes

There are two new progress painters that implement `IAnimatedProgressPainter`: `StripedProgressPainter` and `WaveProgressPainter`. The `StripedProgressPainter` is very similar in appearance to the `BarberPoleProgressPainter`, but it doesn't share it's padding limitations because it uses `GraphicsPaths` instead of painting an image that is moved. It also allows you to set both colors. The `WaveProgressPainter` is similar in functionality except that it uses bezier curves instead of stripes. (I'm actually a little disappointed in the wave painter, it's not quite as cool as I thought it would be. Maybe you can do more with it, and maybe you'll like it more than I did) The demo sets the progress bar's `Value` property to max to display the animation, but this is not required.

To use the animation, all you have to do use the `ProgressBarEx.StartAnimation()` and `ProgressBarEx.StopAnimation()` methods. If a non-animated progress painter is the current progress painter, these methods with do nothing.

License

Written By

BoneSoft

Software Developer (Senior) BoneSoft Software

United States

I've been in software development for more than a decade now. Originally with ASP 2.0 and VB6. I worked in Japan for a year doing Java. And have been with C# ever since.

In 2005 I founded BoneSoft Software where I sell a small number of developer tools.