HOWTO: Upgrade Annotations to DotImage 5.0 (From 4.x and Older)

Applicability

This article applies to users who may still have legacy code targeting DotImage from 4.0 and older. It should apply to very few users at this point as version 5.0 was released 20+ years ago.

The big take-away is that this is not a drop-in-place upgrade. This is because we had to completely rewrite/refactor our annotations components from the ground up.

This article will provide the reasons behind the design change, explain the advantages of this new design and give you an idea of the type of code changes required when upgrading.

Why was the design changed?

The primary reason for the design change was to improve the extensibility, specifically in terms of custom annotations and rendering, by breaking the annotations into separate data, rendering and UI classes. Having the data and rendering separate from the UI allow the annotations to be hosted in environments other than .NET Forms and controls. In fact, we did just that with our WebAnnotationViewer component, which uses the DotAnnotate data and rendering objects with its own UI elements.

Another advantage of separating the annotation functionality is the ability to change the look or "skin" of an annotation. In previous versions, the only way to make an annotation draw itself differently was to create your own custom annotation. With the new design, each annotation has a rendering engine associated with it. These engines can be swapped out with a custom renderer to instantly change the way an annotation is drawn.

How will this design change impact my (DotImage 4.0 or older) application?

If your application only uses the built-in annotation objects, the changes required will be minor. For instance, the annotation classes have been moved to a different namespace (Atalasoft.Annotate.UI). We've also changed from using .NET Pen, Brush, Font and Image classes to our own more generic classes, so constructor arguments and property settings will need to be modified.

Those who have created their own custom annotations will be affected the most with this design change. Each annotation will need to be separated into a few classes as outlined below:

Data Class

This class must derive from Atalasoft.Annotate.AnnotationData. The purpose of this class is to hold all information required to describe the annotation. This includes any pens, brushes or other objects used by the annotation

UI Class

This class must derive from Atalasoft.Annotate.UI.AnnotationUI. For common annotations there will be very little code in this class. User actions, such as mouse events, are passed from the AnnotationController to the UI class so it can provide any extra interactive functionality. The base AnnotationUI class can handle all standard actions for moving, resizing and rotating while allowing these methods to be overridden for custom functionality

Renderer Class

The renderer class is used to draw the annotation and must implement the IAnnotationRenderer interface. DotAnnotate has a base renderer (Atalasoft.Annotate.Renderer.AnnotationRenderingEngine) that all of our annotation rendering engines derive from. Custom renderers can also derive from this class to simplify the process.

Annotation UI Factory Class

(Optional but recommended)

The AnnotationController has a collection of classes called factories that implement the Atalasoft.Annotate.UI.IAnnotationUIFactory interface. The purpose of a factory is to create an AnnotationUI object from an AnnotationData object. The only time this is used is when an AnnotationData object is deserialized without its corresponding UI object and the AnnotationController needs to create an AnnotationUI object to work with.

In general this will involve a lot of code shuffling rather than rewriting. In most cases when converting our own annotations we found that the annotation code was reduced and simplified.

What benefits do I receive from this code change?

There are many benefits to the new design:

- Additional functionality will be given to your annotations, such as mirroring and rotating, without any additional coding.
- Rendering code is simplified by not having to deal with the orientation, rotation, scroll position or zoom level being used. You simply draw the annotation as if it's unmodified at location {0, 0} in the control and we do the rest.
- A skinning feature can be provided in your application by swapping out the rendering engines used for annotations.
- The data and rendering classes can be used for both Windows and web applications (when using our WebAnnotations control).
- Serializing annotations is greatly simplified by using standard .NET serialization with our XmpFormatter instead of requiring annotations to handle writing and parsing XMP data directly.

Custom Annotation Example

To provide a better understanding of what custom annotation code requires, we will step through the process of creating a triangle annotation. The full source code for this annotation is distributed with the SDK.

TriangleData

This class derives from AnnotationData and adds a Fill property. The requirements for an annotation data class are that it contain a default constructor and implement ICloneable. If this annotation needs to be serialized it must also implement the ISerializable interface.

In order for DotAnnotate to render this TriangleData object, it needs to know which rendering engine to use. This is done by adding the data type and renderer to the static AnnotationRenderers collection. An easy way to do this is to use a static constructor in your

data class.

```
tatic TriangleData() { Atalasoft.Annotate.Renderer.AnnotationRenderers.Add(
typeof(TriangleData), new TriangleRenderingEngine() ); }
```

The required Clone method makes use of the CloneBaseData method of the AnnotationData, so we only need to copy our Fill object.

```
ublic override object Clone() { TriangleData data = new TriangleData();
base.CloneBaseData(data); data._fill = (this._fill == null ? null : this._fill.Clone());
return data; }
```

Our Fill property must perform a few tasks in order to be fully functional. This includes raising the PropertyChanging event, removing and setting event handlers for the AnnotationBrush properties and sending an AnnotationUndo to the AnnotationController. Obviously if you don't care about the PropertyChanging events or undo feature, these can be skipped. We'll step through this:

The first thing you will want to do is raise the PropertyChanging event. This event notifies handlers about the change and allows them to modify or cancel the change.

```
nnotationPropertyChangingEventArgs e = new AnnotationPropertyChangingEventArgs(this, "Fill",
this._fill, value); if (!this.IgnoreDataChanges) { OnPropertyChanging(e); if (e.Cancel)
return; }
```

Next we create an AnnotationUndo object that will be passed to the AnnotationController after the property has been changed.

```
nnotationUndo undo = new AnnotationUndo(this, "Fill", this._fill, "Fill Change");
```

Now it's time to change the property value. For full undo support, event handlers need to be controlled for the AnnotationBrush. We provide simple methods to set and remove these handlers for you.

```
ase.RemoveBrushEvents(this._fill); this._fill = value; base.SetBrushEvents(this._fill);
```

Finally we will notify the AnnotationController about this change and provide the undo we

created earlier.

```
f (!this.IgnoreDataChanges) { OnAnnotationControllerNotification( new
AnnotationControllerNotificationEventArgs(
Atalasoft.Annotate.AnnotationControllerNotification.Invalidate, undo ) ) };
```

We'll now add a helper method that will define the points used by our TriangleData. The points are specified in annotation space, meaning from {0, 0} to the annotation size.

```
ublic PointF[] GetTrianglePoints() { PointF[] points = new PointF[3]; points[0] = new
PointF(0, this.Size.Height); points[1] = new PointF(this.Size.Width, this.Size.Height);
points[2] = new PointF(this.Size.Width / 2f, 0); return points; }
```

TriangleAnnotation

Next we create a TriangleAnnotation class, which derives from AnnotationUI. Each constructor for this class must pass a TriangleData object to the base constructor. This guarantees that each AnnotationUI has a corresponding data object.

```
ublic TriangleAnnotation() : base(new TriangleData()) { this._data = this.Data as
TriangleData; base.SetGrips(new RectangleGrips()); }
```

The SetGrips method used in the constructor sets the annotation grips this class will use, making it easy for a custom annotation to provide their own grips.

The only method we override in TriangleAnnotation is GetRegion. This method returns a region that is used for hit testing and cursor changes.

```
ublic override AnnotationRegion GetRegion(AnnotateSpace space) { AnnotationRegion region =
new AnnotationRegion(); SizeF size = this.Data.Size; // Specify the points in annotation
space. region.Path.AddPolygon(this._data.GetTrianglePoints()); // Be sure to add the grips to
the region. base.AddGripsToRegion(region); // Apply the required transformation.
base.ApplyRegionTransform(region, space); return region; }
```

TriangleRenderingEngine

The final class we will create is the TriangleRenderingEngine that will draw our annotation. We derive from AnnotationRenderingEngine in order to use some existing methods to simplify dealing with the transformation matrix of the annotation and viewer. The only

method we have to override is RenderAnnotation.

```
ublic override void RenderAnnotation(AnnotationData annotation,
Atalasoft.Annotate.Renderer.RenderEnvironment e) { TriangleData data = annotation as
TriangleData; if (data == null) return; if (data.Fill == null) return; //
SetGraphicsTransform handles combining multiple // transformation matrix objects so you can
render normally. base.SetGraphicsTransform(annotation, e); Brush b = CreateBrush(data.Fill);
if (b != null) { PointF[] points = data.GetTrianglePoints(); e.Graphics.FillPolygon(b,
points); b.Dispose(); } base.RestoreGraphicsTransform(e); }
```

There are a few lines of code that were left out of the above example, such as constructors and serialization code.

Serializing Annotations

Speaking of serialization, this part of DotAnnotate has been improved as well. The previous release required that your custom annotation add ToXmp and FromXmp methods that would create and parse the data. This was very problematic and prone to errors. In 5.0 we now have an XmpFormatter that uses standard .NET serialization to convert annotations to and from XMP. To make the TriangleData class serializable we would add a special constructor and override the GetObjectData method.

```
ublic TriangleData(SerializationInfo info, StreamingContext context) : base(info, context) {
this._fill = (AnnotationBrush)SerializationInfoHelper.GetValue( info, "Fill", new
AnnotationBrush(Color.Blue) ); base.SetBrushEvents(this._fill); }
[SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter=true)] public
override void GetObjectData(SerializationInfo info, StreamingContext context) {
base.GetObjectData(info, context); info.AddValue("Fill", this._fill); }
```

The TriangleAnnotation should also include the serialization constructor but does not need to override GetObjectData since we do not have additional information to add.

Original Article

Q10162 - HOWTO: Upgrading Annotations to DotImage 5.0

Last Update

2026-10-05 - TD - (Reformatted code fencing)

HOWTO: Upgrade Annotations to DotImage 5.0 (From 4.x and Older)

Atalasoft Knowledge Base
https://www.atalasoft.com/kb2/KB/50290/HOWTO-Upgrade-Annotations-to-DotImag...