RADIANT
VISION SYSTEMS

Getting Started

Requirements

- Visual Studio 2013+
- .NET Framework 4.0+
- TrueTest 1.7+
- General familiarity with TrueTest

Overview

Device Interfaces

When using the TrueTest API, we highly recommend developing your own Device Interface (DI) to insert custom functionality into TrueTest rather than developing a new application with your own graphical user interface (GUI). As you will see in later sections, by following this plugin approach you can fulfill almost any functional requirement.

Working with our API in this manner allows you to meet your requirements without having to worry about the user interface, since the standard TrueTest user interface will be used. This makes it easier to maintain and quicker to develop.

Our Plugin Approach

The TrueTest API provides mechanisms to develop your custom functionality and plug it directly into TrueTest. This is done by writing your custom logic inside either a device interface. A device interface is a logical

construct that TrueTest is aware of, solely for executing custom functionality.

Dynamic Loading via Reflection

This plugin ability for TrueTest is achieved through reflection and polymorphism. When developing your own DIs, you will compile them to a .NET DLL (dynamically-linked library). The TrueTest engine can then find these assemblies and the types therein via reflection.

Reflection is a mechanism that provides a way to access metadata about a given type (class) and its members.

- **Note**: See Microsoft Docs for more information on reflection.

When the TrueTest application starts, it first loads all DLLs found in the TrueTest installation directory, and then uses reflection to obtain information about the types located in those assemblies.

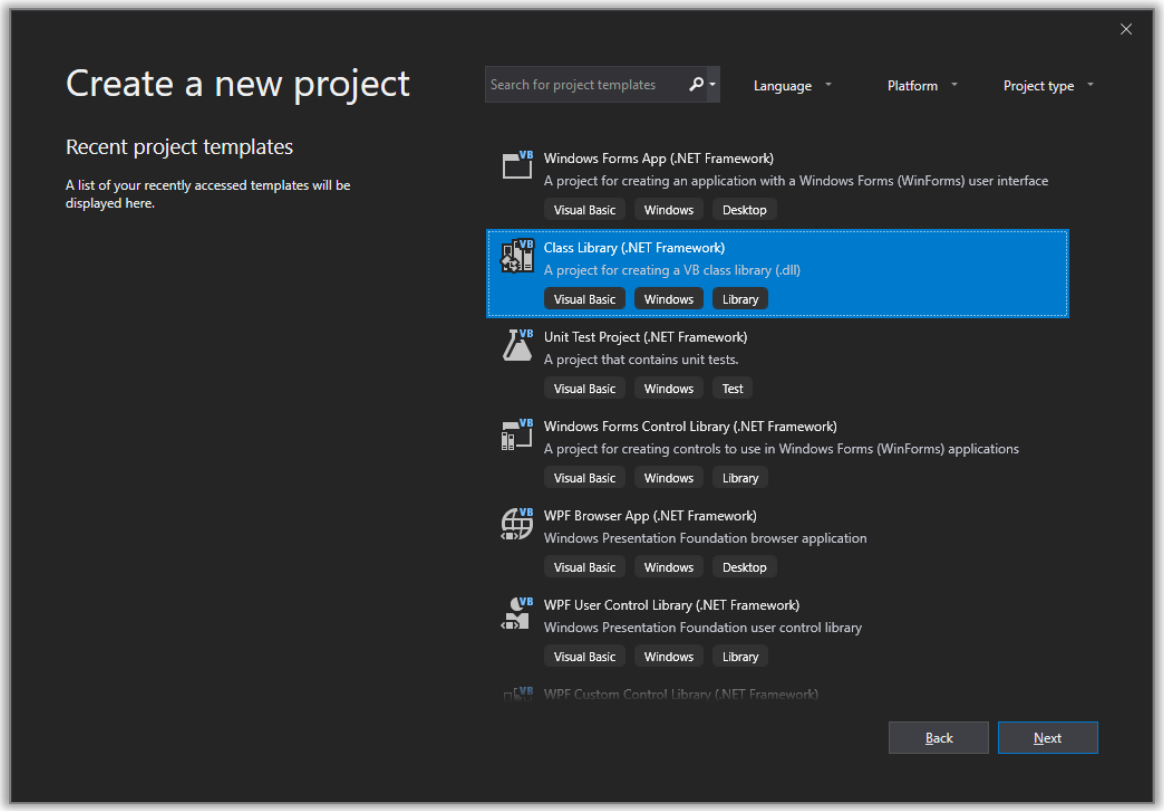
TrueTest uses this information to search for any types in the assembly that inherit from `PatternGeneratorBase`, which is an abstract classes designed for this plugin purpose. Any DI you develop must inherit from this base class, because by virtue of polymorphism, TrueTest can then find and instantiate all the custom device interfaces that it has loaded in memory. It will then populate the respective places in the application from which you can select and utilize your component.

Device Interfaces

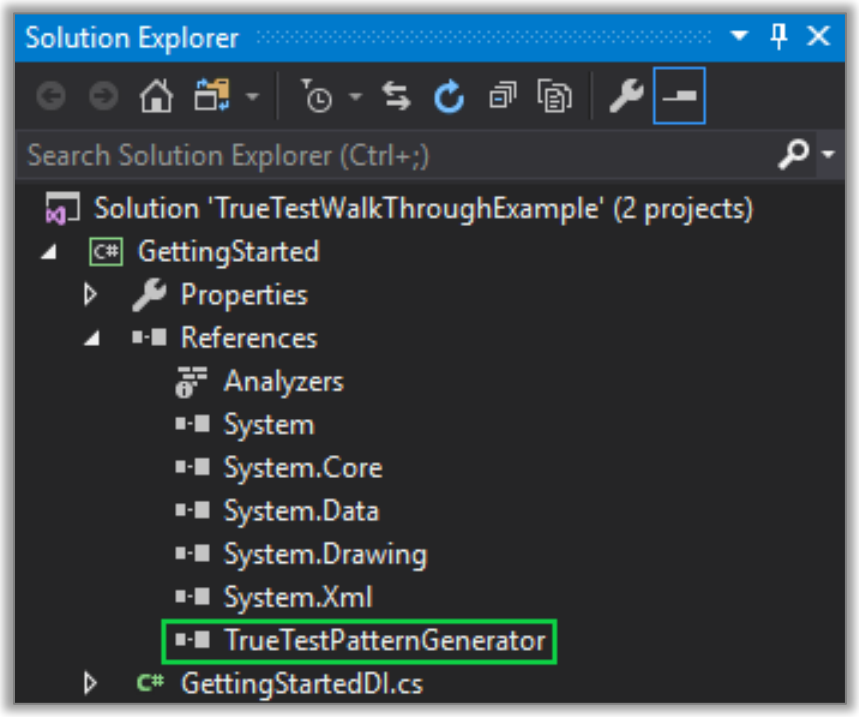
First Steps

Inheriting from PatternGeneratorBase

Open Visual Studio and create a new project. Select the **Class Library (.NET Framework)** template:


Create a new project
Search for project templates
Language
Platform
Project type
Recent project templates
A list of your recently accessed templates will be displayed here.
Windows Forms App (.NET Framework)
A project for creating an application with a Windows Forms (WinForms) user interface
Visual Basic
Windows
Desktop
Class Library (.NET Framework)
A project for creating a VB class library (.dll)
Visual Basic
Windows
Library
Unit Test Project (.NET Framework)
A project that contains unit tests.
Visual Basic
Windows
Test
Windows Forms Control Library (.NET Framework)
A project for creating controls to use in Windows Forms (WinForms) applications
Visual Basic
Windows
Library
WPF Browser App (.NET Framework)
Windows Presentation Foundation browser application
Visual Basic
Windows
Desktop
WPF User Control Library (.NET Framework)
Windows Presentation Foundation user control library
Visual Basic
Windows
Library
Back
Next

Next, add a reference to **TrueTestPatternGenerator.dll** from the TrueTest installation directory to your new project:



The first thing to do when designing your own class is to inherit from the `PatternGeneratorBase` class and override the abstract `ShowPattern` method.

This is essentially a fully functional device interface at this point. We'll look at the `ShowPattern` method in more detail a little later.

There is one more step to follow before TrueTest can load the DI, and that's marking your class with the `Serializable` attribute. Altogether, your DI skeleton will look like the following:

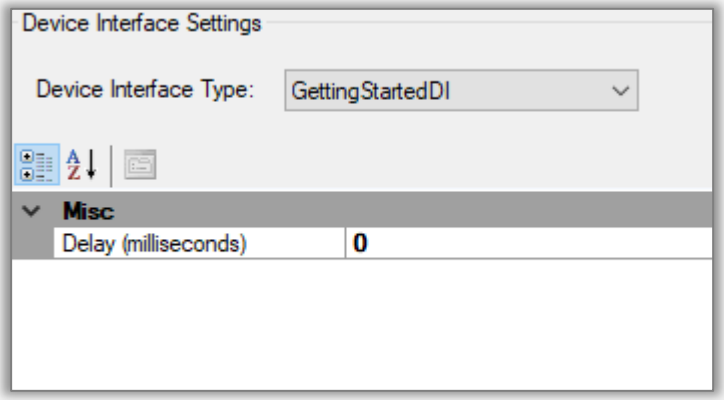
```
using System;
using TrueTestPatternGenerator;

namespace Radiant.SDK.GettingStarted
{
    [Serializable]
    public class GettingStartedDI : PatternGeneratorBase
    {
        protected override void ShowPattern(TrueTestPattern p)
        {
            throw new NotImplementedException();
        }
    }
}
```

Marking your class with **`Serializable`** allows TrueTest to save the browsable properties in your Device Interface using serialization. These properties can be modified by the user to configure certain options, such as a reporting directory; saving the configuration avoids the need to reconfigure them each time TrueTest is loaded.

- **Note**: see Microsoft Docs for more information on serialization

At this point you can do a quick sanity check to test the new DI with TrueTest to verify you have the initial setup in place:

1. Copy and paste the compiled assembly (*.dll) from your solution's bin\debug folder to the TrueTest installation directory
2. Start TrueTest and navigate to **Tools → Options** to open the **Global Settings** form
3. Select your DI from the **Device Interface Type** dropdown menu on the right:



4. Press **OK**

You should be able to select and save the chosen DI without any errors.

Notice the single property displayed in the property grid, **Delay (milliseconds)**. This is a *browsable* property, meaning the property is visible in the property grid. This is achieved by marking the property with the `Browsable(true)` attribute.

By default, properties will be browsable if the attribute is omitted, so if you need to hide a property then set `Browsable(false)`. Additionally, you can create read-only properties that are still visible to the user, but their values cannot be changed.

For example:

```
using System.ComponentModel;

public class MyClass
{
    [Browsable(false)]
    public Type MyHiddenProperty { get; set; }
}
```

TrueTest uses the **Browsable** attribute extensively for allowing the user to configure the properties of a device interface, analysis or pattern.

Now that you have a working DI, let's add some basic functionality.

Core Methods

Initialization and Shutdown

There are two virtual methods in the `PatternGeneratorBase` class, `Initialize` and `Shutdown`, that you will almost always want to override. These two methods are called by TrueTest as part of the device interface initialization routine. This routine is triggered either when the TrueTest software is launched (or closed for `Shutdown`), or after selecting **OK** from the **Tools → Options** menu.

The initialization routine is triggered by selecting **OK** because the righthand panel in the **Options** menu, **Device Interface Settings**, is where DI-specific properties are configured. After changing any of these properties, the user must click **OK** to save the changes.

It's very likely that your custom logic will depend on those properties, and you will want to handle these updated settings by creating new objects, reinitializing collections, etc. Therefore, the initialization routine is triggered at this time for the purpose of giving you a chance to handle the new state.

- **Note**: if a user clicks **Cancel** instead of **OK**, the initialization routine will be aborted and `Initialize` will not be called.

The `Initialize` method returns a Boolean to TrueTest engine, indicating whether the initialization routine was successful. `Shutdown` behaves the same, but is used for any teardown logic:

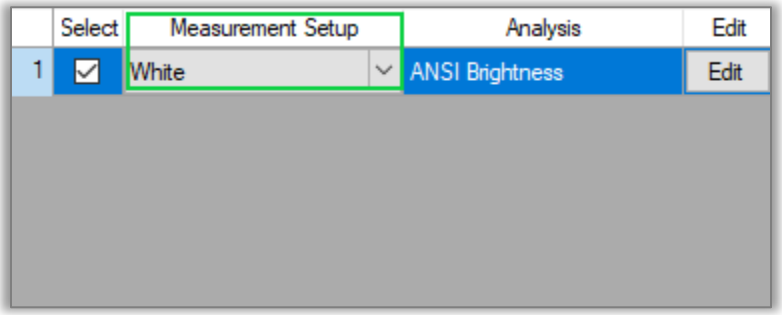
```
protected override bool Initialize()
{
    //Your setup logic here
    //Return a boolean indicating whether setup routine was successful
}

protected override bool ShutDown()
{
    //Your teardown logic here
    //Return a boolean indicating whether teardown routine was succesful
}
```

Along with `Initialize`, you should also override `IsInitializationRequired`. Most often you will want to return `True` to force reinitialization, since you will almost always have setup and teardown logic that you need to execute. In very simple DIs however, you may not need to perform any setup or teardown, in which case you can return `False` to avoid calling `Initialize`.

Show Pattern

`ShowPattern` is the only required member you must override from the base class (for historical reasons). Before we can fully explain how to utilize this method, some information on the TrueTest sequencer is in order.

A TrueTest sequence is a set of analysis steps that are to be executed when a test is initiated. Each of those steps has a **Pattern Setup** associated with it. In the TrueTest user interface, these are referred to as a **Measurement Setup**, but they are truly pattern setups behind the scenes since these measurement setups also have a **Pattern** associated with them. You can think of a pattern setup as a specialized measurement setup.



A measurement setup contains the configuration settings for the camera—focus distance, exposure times, calibrations, etc.

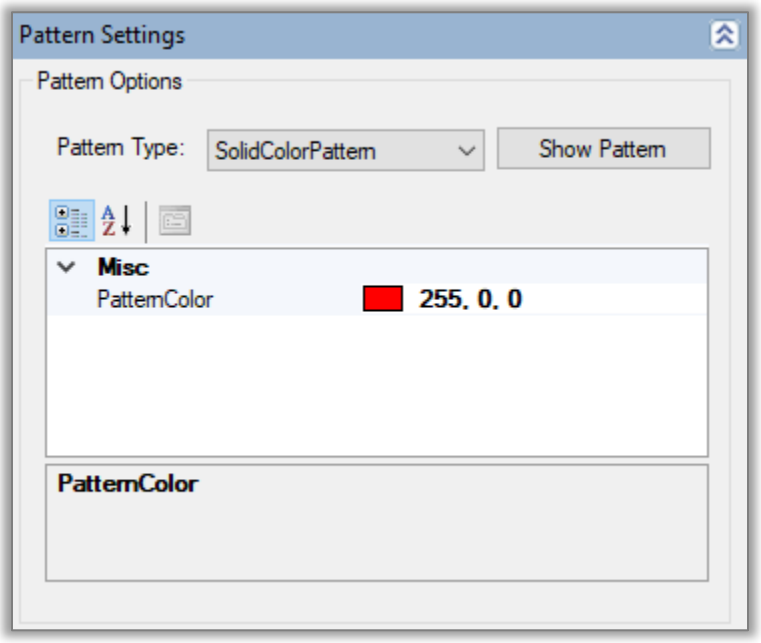
Patterns are the concept of a specific image that should be displayed on the device under test (DUT). This could be an image of a solid color, or a checkerboard pattern for example. Patterns can be associated with one or more pattern setups, and pattern setups can be associated with one or more analyses.

The TrueTest engine relies on this structure to ensure that a given analysis operates on a measurement that was captured with the correct pattern. The high-level overview of the process is as follows:

The TrueTest engine will call `ShowPattern` before executing a given analysis in the sequence. This provides a mechanism to allow the hardware to display the pattern that is required by the analysis.

The `ShowPattern` method takes a single parameter, **p**, of type `TrueTestPattern`. `TrueTestPattern` contains a reference to `Pattern`, which is of type `PatternBase`. PatternBase is an abstract class, and you must inherit from this class and design your own pattern type or rely on one of the built-in types.

`PatternBase` objects are meant to store information about the image that the DUT must show for a given analysis, from which the `ShowPattern` routine can then use for displaying the pattern. Let's look at an example with the `SolidColorPattern` type, which has the just a single browsable property, `PatternColor`. The property is of type `Color`, and this provides a configurable property in the user interface that a user can use to set the R, G and B values for a given color:



When providing a custom implementation for **ShowPattern**, a very simple implementation where all analyses use just a solid color pattern might look like the following:

```
protected override void ShowPattern(TrueTestPattern p)
{
    //First downcast to the specific type of pattern you wish to handle
    var pattern = p.Pattern as SolidColorPattern;

    //Return early when encountering a sequence step with
    //an unsupported pattern type, or perform some other logic
    if (pattern == null) return;

    //MyDevice is a hypothetical dependency that can control
    //the DUT
    MyDevice.ShowSolidColor(pattern.PatternColor);

    //Give the DUT a chance to fully render the pattern, if necessary,
    //by pausing execution by the specified amount
    System.Threading.Thread.Sleep(DelayMilliseconds);
}
```

It is very important to understand that **ShowPattern** is a blocking method. TrueTest calls this method synchronously, so that execution of the thread that the analysis is running on cannot proceed until the **ShowPattern** routine is finished. This behavior is enforced so that the DUT will always have the pattern ready before the camera captures the measurement.

Notice the use of **Thread.Sleep(DelayMilliseconds)**. DelayMilliseconds is a property defined in the base class **PatternGeneratorBase**. This property is an integer which specifies the time to wait in milliseconds, which can be useful if you need to wait a small amount of time for a pattern to fully render on the display before a measurement can begin.

One final, yet important note: there exists a default timeout for the `ShowPattern` routine. If it does not complete within 20 seconds, the sequence will automatically continue its execution. You can change this timeout value by accessing the `Settings` type:

```
//Change timeout to a minute instead of default of 20 seconds
TrueTest.AppSettings().ShowPatternRequestedTimeoutMs = 60000;
TrueTest.AppSettings().Save();
```

Next, we'll look at how to achieve some of the most common tasks with the API.

Common Tasks

Overview

For high level tasks such as retrieving analysis results or measurement data, you will typically achieve these by handling one of the corresponding static events defined in the `TrueTest` type. We'll cover some of the most useful events in detail.

AnalysisComplete

If you wish to do something with the results of each analysis, you can subscribe to the `AnalysisComplete` event. This event provides the list of `Result` objects that were generated by the analysis. This event is raised each time an analysis completes.

The event arguments contain additional information, such as the name of the analysis that completed and an overall pass/fail result for the analysis.

```
private void TrueTest_AnalysisComplete(object sender,
AnalysisCompleteEventArgs e)
{
    string analysisName = e.AnalysisName;

    TrueTest.AnalysisResultEnum overallPassFail = e.PassFail;

    List<Result> results = e.Results;
    foreach (Result result in results)
    {
        string resultName = result.Name;
        float resultValue = result.Value;
        //ValueString is used primarily for a list of results stored as a
        //CSV string, or error information
        string resultValueStr = result.ValueString;
        string resultUnit = result.Unit;
        TrueTest.AnalysisResultEnum resultPassFail = result.PassFail;
    }
}
```

Result is a simple type that contains information for a unique result, such as the name of the result, its value and units.

MeasurementComplete

Retrieving the measurement data from a camera capture can be done by handling the **MeasurementComplete** event. This event is raised for each measurement captured.

A **MeasurementBase** object contains access methods for retrieving the tristimulus data or the luminance and chromaticity data, using the **CIEColor** type.

Here's how to retrieve the data, either by accessing the tristimulus arrays or enumerating through each pixel and calculating the luminance and chromaticity, according to the CIE 1931 color space:

```
private void TrueTest_MeasurementComplete(object sender,
MeasurementCompleteEventArgs e)
{
    MeasurementBase meas = e.Measurement;

    //Tristimulus XYZ data
    var x = MeasurementBase.TristimlusType.TrisX;
    var y = MeasurementBase.TristimlusType.TrisY;
    var z = MeasurementBase.TristimlusType.TrisZ;

    float[,] trisX = meas.GetTristimulusArrayF(x);
    float[,] trisY = meas.GetTristimulusArrayF(y);
    float[,] trisZ = meas.GetTristimulusArrayF(z);

    //Luminance & Chromaticity data
    for (var col = 0; col < meas.NbrCols; col++)
    {
        for (var row = 0; row < meas.NbrRows; row++)
        {
            CIEColor color = meas.GetColor(col, row);
            float lv = color.Lv;
            float cx = color.Cx;
            float cy = color.Cy;
        }
    }
}
```

CIEColor is a useful type for working with measurement data and contains properties for retrieving the luminance and chromaticity in various color spaces (1931, 1976, etc.).

SequenceComplete

If you wish to do something once the sequence finishes, such as export a report, you can handle the **SequenceComplete** event.

If you wish to export all the results for the entire sequence, first aggregate the results into a collection by handling AnalysisComplete, and then enumerate through and write the results to a file:

```
private void TrueTest_SequenceComplete(object sender,
SequenceCompleteEventsArgs e)
{
    var sb = new StringBuilder();
    sb.AppendLine($"Sequence: {e.SequenceName}");
    sb.AppendLine($"Overall Pass/Fail: {e.PassFail}");
    sb.AppendLine();

    //Assuming _results is a field of type List<Result>
    //and populated from TrueTest_AnalysisComplete event handler
    foreach (Result result in _results)
    {
        sb.AppendLine($"Analysis: {result.AnalysisName}\r\n" +
                      $"Result: {result.Name}\r\n" +
                      $"Value: {result.Value}\r\n" +
                      $"Units: {result.Unit}\r\n" +
                      $"Pass/Fail: {result.PassFail}\r\n");

        sb.AppendLine();
    }

    string dateTime = DateTime.Now
        .ToString("G") //ex: 01/01/1960 12:00:00
        .Replace('/', '-')
        .Replace(':', '.');

    string filePath =
        Path.Combine(@"C:\tmp", $"{e.SerialNumber}_{dateTime}");

    File.WriteAllText(filePath, sb.ToString());
}
```

SequenceRunAllStarted

Handling the SequenceRunAllStarted event can be useful if you need to perform some logic that must occur when the sequence starts, but before the engine runs the sequence. A good example of this is for setting the serial number for a measurement, as this must happen before the sequence begins for TrueTest to successfully register the serial number.

Additionally, you can modify the sequence to configure which analyses are selected, possibly based on some filter criteria. Here's how to do both:

```
private void TrueTest_SequenceRunAllStarted(object sender, EventArgs e)
{
    //Pass argument by name for single channel (*.seqx)
    //as TrueTest.SerialNumber has an optional parameter for the
    //channel index, which is only used for multi-channel (*.mseqx),
    //so here we let TrueTest use the default value for the index
    TrueTest.set_SerialNumber(value: "abc1234");

    Sequence sequence = TrueTest.get_Sequence();
    foreach (var item in sequence.Items)
    {
        string patternSetupName = item.PatternSetupName;
        item.Selected = patternSetupName.ToLower().Contains("white");
    }
}
```

Next Steps

Standard DI API Example

The best next step for gaining familiarity with our API and in creating device interfaces is to open and run the example solution that was installed with the SDK package, titled **Device Interface Standard**.

One of the most common tasks you might perform when writing your own device interface is to integrate with a fixture. The example project demonstrates how to integrate with a fixture via serial communication (RS232) to send/receive commands and have TrueTest respond accordingly.

The example solution demonstrates the various concepts discussed in this guide and lets you run and modify the code to get a better understanding of how to use all the pieces together.

TrueTest API Specification

There are many operations and other types available to you through the API. For a detailed breakdown of these, refer to the TrueTest .NET API Specification document that was installed alongside this walkthrough guide.