DE10000 USER MANUAL

PLUGIN DEVELOPMENT

TestPlanner

testplanner

DEICO

Document number: SBL-0038 Rev.0 2023

Contents

INTRODUCTION

Overview

TestPlanner provides a robust framework for Plugin Development based on OpenTAP plugins, enabling the creation of plugins with extended features. This API includes enhancements such as extended test steps and mixins, designed to offer more flexibility and functionality to plugin developers.

Target Audience

This document is intended for plugin developers and users of the developed plugins. It aims to provide comprehensive information about using TestPlanner's plugin development API, ensuring both developers and end-users can effectively utilize and integrate these extended features into their workflows.

Note This document assumes that developers are familiar with the basics of OpenTAP plugins.

Prerequisite Knowledge

For those new to OpenTAP or in need of a refresher, reviewing the OpenTAP basics documentation is recommended. This foundational knowledge will help developers understand the core concepts and functionalities essential for developing plugins with TestPlanner. The OpenTAP developer guide can be accessed here.

Note By ensuring familiarity with OpenTAP, users will be better equipped to leverage the extended features provided by TestPlanner.

GETTING STARTED

Plugin Development Components

TestPlanner plugin development includes two essential DLL files:

- **TestPlanner.PluginDevelopment:** This DLL is developed using .NET Standard 2.0, ensuring compatibility with both .NET Core and .NET Framework. It contains the core functionalities required for plugin development.
- **TestPlanner.PluginDevelopment.Gui:** This DLL is specifically designed for the .NET Framework and includes GUI components necessary for creating views within the framework.

These components provide the foundational elements required for developing and integrating plugins with extended features in TestPlanner.

Installation and Setup Instructions

TestPlanner plugin development components are installed automatically when TestPlanner is installed using its setup file. To verify successful installation, the OpenTAP's packages directory at C:Files must be checked. The DLL files should be present in this directory.

Note Alternatively, the plugin development components can be installed manually using the .TapPackage file TestPlanner.PluginDevelopment.X.X.X, where XXX represents the version number.

To install the .TapPackage file manually, the steps below should be followed.

- **Using the TestPlanner PackageInstaller:**

  1. Double-click on the .TapPackage file to launch the PackageInstaller and complete the installation.

- **Using Command Line:**

  1. Open a command prompt.
  2. Navigate to the OpenTAP directory:
     cd C:\Program Files\OpenTAP
  3. Run the following command to install the package:
     tap package install "<path to .TapPackage file>"

Accessing and Using DLL Files

There are 2 primary ways to access and use the DLL files for TestPlanner plugin development.

- **Using Visual Studio TestPlanner Template:** During the package installation, a Visual Studio plugin project template is automatically installed, streamlining the development process for plugin developers. By following the instructions for creating a new plugin project with this template, developers can:
  - Quickly set up a new project with the necessary configuration,
  - Automatically import the required DLLs,
  - Generate initial plugin files, reducing the setup time and potential configuration errors.

- **Manually Importing DLLs:** For those who prefer manual configuration, the DLL files can be directly imported into an OpenTAP plugin project in Visual Studio by following the steps below.

  1. Navigate to **C:Files.PluginDevelopment**.
  2. Add the DLL files to the Visual Studio project.
  3. Once imported, the functionalities provided by the DLLs can be used in the OpenTAP plugin project.

Note Both methods ensure that developers have access to the core functionalities and GUI components necessary for developing TestPlanner plugins.

Creating TestPlanner Plugin Project from Template

To create a plugin project using the TestPlanner template, the steps below should be followed.

1. Open Visual Studio and select **Create a new project** from the start window.
2. Search for TestPlanner template using the search bar.
3. Select **OpenTap TestPlanner Project** from the list of templates and click on **Next** to proceed.

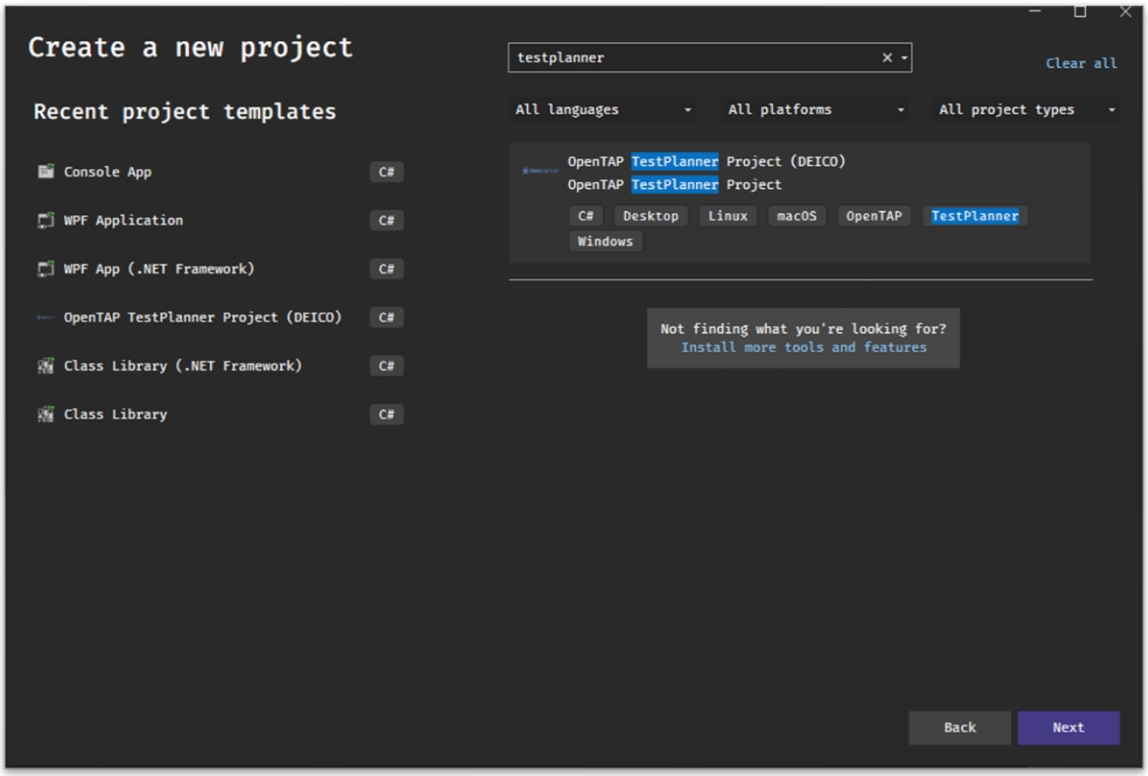


Figure 1: Creating a New Project

4. Configure the new project, provide a name and choose the directory where the project will be saved.
5. During the project setup, developers will have the option to select specific plugins that they want to generate. Choose the desired plugins and Editor for debugging the plugin.
6. Click **Create** to finalize the project creation.

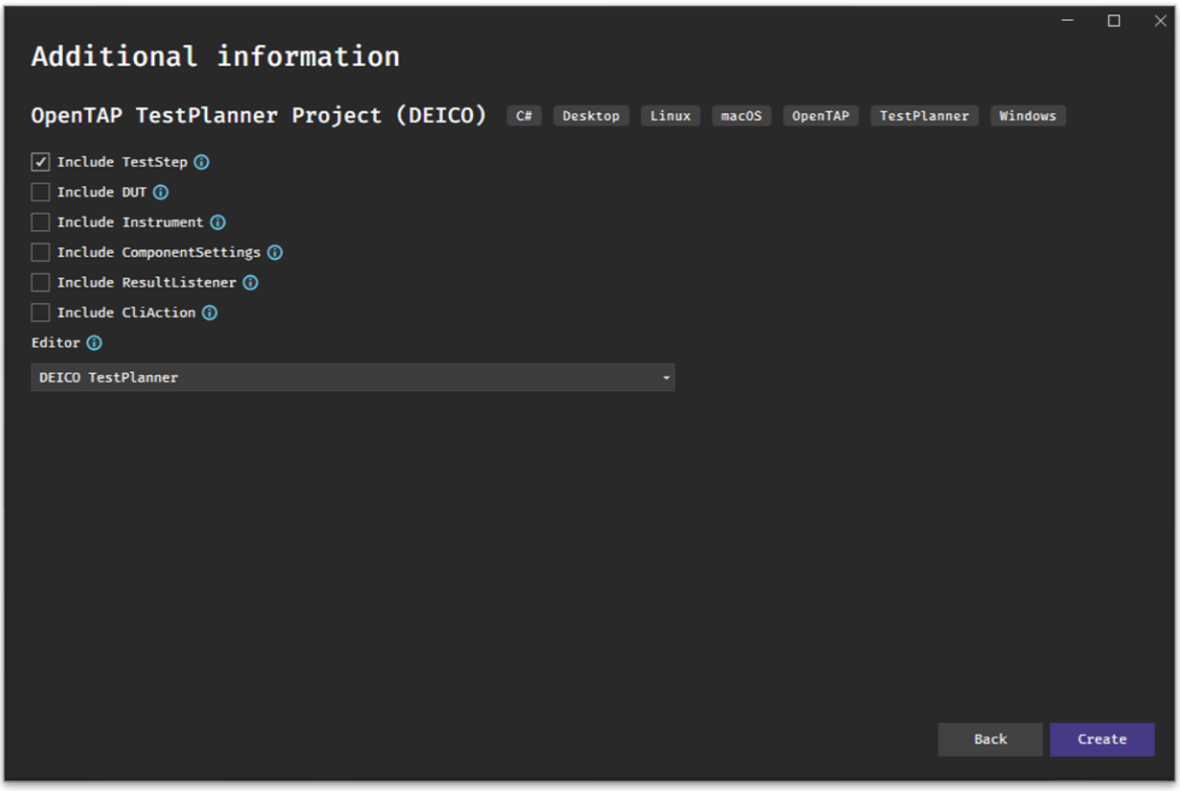


Figure 2: The New Project

Following these steps will set up a new TestPlanner plugin project with the necessary configuration and initial files, allowing developers to start developing their plugins quickly and efficiently.

EXTENDED FEATURES

The Plugin Development API extends OpenTAP's test steps to incorporate new features, enhancing the functionality and flexibility of the plugins.

Note TestPlanner plugins are designed to work cohesively based on the Plugin Development API. For instance, the following features are utilized by PDF test report plugins. This means that the Plugin Development API serves as a foundational interface for plugins, and developed plugins interact with each other through this API.

Extended Test Step

TestPlanner Plugin Development API offers a base class named **ExtTestStep** for creating extended test steps. This class introduces several new test step settings under the **Results and Limits** group.

- ⇨ **Selectable Results:** Allows users to select specific results they want to capture during the test execution. Selectable results are listed in a combobox. Developers must add Output attribute to their test settings during development.

- **Enable/Disable Result Publishing:** Provides the option to enable or disable the publishing of test results. When this option is enabled, the selected test result will be published to result listeners.

- **Enable/Disable Limit Check:** Allows users to enable or disable limit checks for the test results. When this option is enabled, the result will be compared with the limits to decide the result (pass or fail) of the test step.

- **Limit & Result Format:** Enables the configuration of the format for limits and results. The format is specified similar to C#'s numeric format. Users can use **0** for mandatory place and **#** as an optional place.

- **Upper and Lower Limits:** Sets the upper and lower boundaries for acceptable test results.

- **Unit:** Specifies the unit of the measurement for the test results.

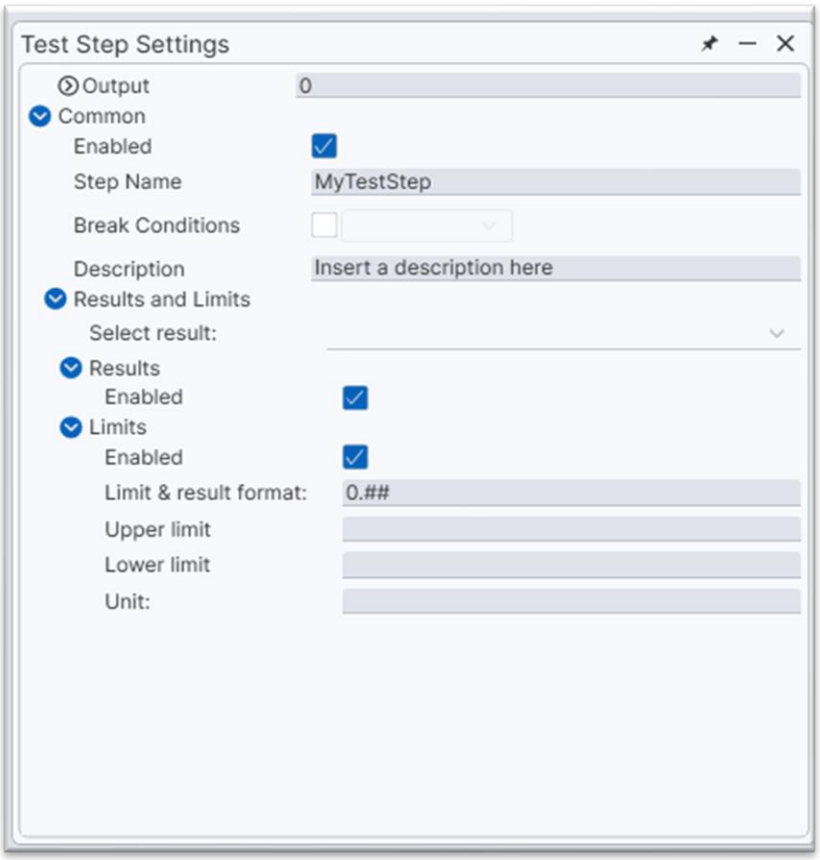


Figure 3: Test Step Settings

These settings provide control over the result of test steps by offering users limit values and enabling them to select which results will be displayed in their test reports. This ensures that users can tailor the output to meet their specific needs and maintain clarity in their test documentation.

ExtTestStep does not automatically compares result with limits or publishes results to result listeners. Developers need to call two methods in the **Run()** method. These methods are **LimitCheck()** and **PublishResults()**.

Note Order of calling these methods are important. **LimitCheck()** must be called before **PublishResults()**.

To create an extended test step, developers must inherit from the **ExtTestStep** class. An example of an extended test step is given below:

```
[Display("Create Random Number", Description: "Creates a random double between min and max values", Group: "New Plugin")]

public class MyTestStep : ExtTestStep
{
  #region Settings
  // ToDo: Add property here for each parameter the end user should be able to change

  [Display("Max", Order: 1)] public int Maximum { get; set; } = 10;

  [Display("Output", Order:3)]
  [Output]
  public double Output { get; set; }

  #endregion

  private Random random;
  public MyTestStep()
  {
    // ToDo: Set default values for properties / settings.
    random = new Random();
  }

  public override void PrePlanRun()
  {
    base.PrePlanRun();
    // ToDo: Optionally add any setup code this step needs to run before the testplan starts
  }

  public override void Run()
  {
    // ToDo: Add test case code here
    Output = random.NextDouble() * Maximum;

    LimitCheck();
```

```
            PublishResults();
        }

        public override void PostPlanRun()
        {
            // ToDo: Optionally add any cleanup code this step needs to run after the entire t
estplan has finished
            base.PostPlanRun();
        }
    }
```

Execution output of this test step is given below:

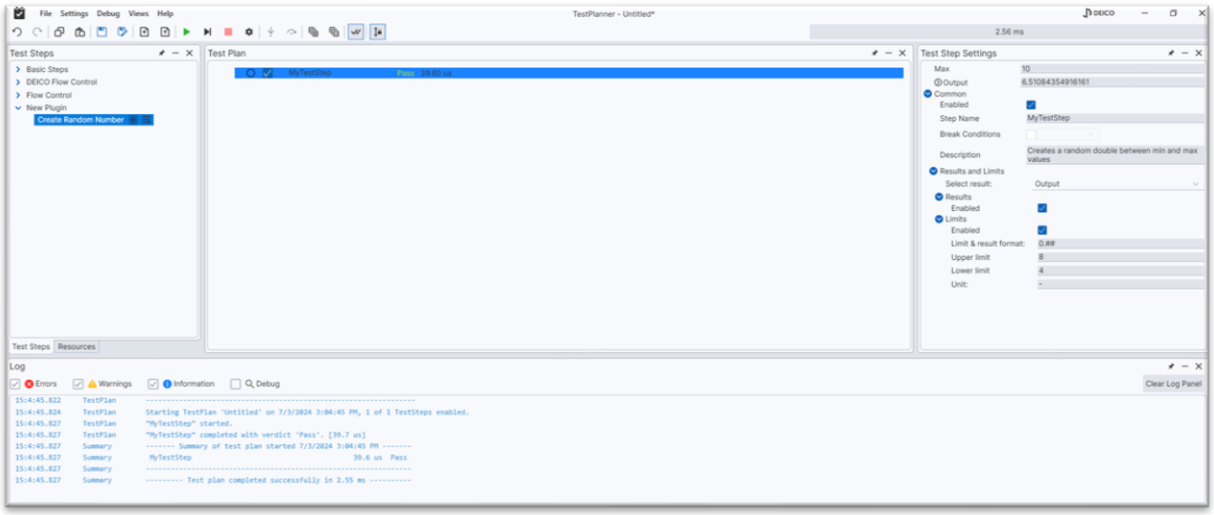


Figure 4: Execution Output

Limit Types

Comparisons are made according to the limit type. Developers must know about which data type corresponds to which limit type. The following table shows this relation.

Table 1: Data & Limit Types

Data Type(s)	Limit Type(s)
double, int, uint, float, byte	Point limit
double[], int[], uint[], float[], byte[]	Numeric array limit
string	String limit
string[]	String array limit
bool	Boolean limit
bool[]	Boolean array limit

LimitCheck() Method

Developers might need to understand how LimitCheck() operates when developing their own plugins. The pseudocode for this method is provided below.

1. If limit check is disabled, return without comparing output and limits.
2. Find the selected output member chosen by the user.
3. Determine the limit type (point limit, array limit, boolean limit or string limit).
4. Call comparison methods according to the limit type to compare output with the limit(s).
5. Upgrade the test step's verdict according to the comparison result.
6. If the limit type is a double array and **show graph** option is enabled, display limits and output on a graph.

Note This is a straightforward method and developers just need to call the **LimitCheck()** method in their **Run()** method.

PublishResults() Method

Result listeners must be developed in conjunction with the data table published by this method. This method is crucial for how result listeners receive output data.

Note Before delving into details, developers should understand the result publishing mechanism of OpenTAP.

The results are published according to the limit type.

Point Limit

The following method shows how results are published using **Results.Publish()** method of OpenTAP.

```
Results.Publish(Name, new List<string>() {"Lower limit", "Result", "Upper limit", "Unit", "Verdict", "Format" },
    PointLowerLimit, res, PointUpperLimit, unit, ComparePointLimit(res), DoubleFormat);
```

The parameters are described below:

- **Name:** Test step's name.
- **new list(){…}:** Data table's headers. Columns are Lower Limit, Result, Upper Limit, Unit, Verdict, and Format in order.

Note The remaining parameters are the information matching the column headers.

Numeric Array Limit

The following method shows how results are published using **Results.PublishTable()** method of OpenTAP.

```
Results.PublishTable(Name, new List<string>() { "Lower limit", "Result", "Upper limit"
, "Unit", "Verdict", "X-Axis", "Format"},.....);
```

Results.PublishTable() provides its parameters to be arrays. All results and limits are provided to result listeners in an array. Other parameters like **Unit** and **Format** are provided in arrays also.

The only different header from the point limits is the **X-axis**. The **X-axis** column contains the x-axis data of a plot if developers want to display a plot on their report.

Remaining Limit Types

Remaining limit types have the exact same column headers as the point limit.

Note This information is given to developers because if they are developing a result listener, they need to know what kind of a data is provided to the result listeners by **ExtTestStep**s.

Image Publishing Test Step

Another extended test step provided by the TestPlanner Plugin Development API is the **ImagePublishingStep**, which is used to publish images to TestPlanner result listeners. There is an existing plugin called **Advanced Test Steps** that implements **ImagePublishingStep**.

If developers wish to implement their own custom behavior for publishing an image to result listeners, they need to create a test step that inherits from **ImagePublishingStep**.

Note A crucial requirement is that the test step must include a property with type **OpenTap.Picture** named **Picture** and set the verdict of the test step to any value.

PublishResults() method is not required in the **Run()** method to publish image to result listeners. An example implementation is provided below:

```
[Display("Publish Image", Description: "Publishes an image to the TestPlanner result l
isteners", Group: "New Plugin")]
public class PublishImage : ImagePublishingStep
{
   public Picture Picture { get; } = new Picture();

   [Display("Source", "The source of the picture. Can be a URL or a file path.", "Pictur
e", Order: 2, Collapsed: true)]
```

```
    [FilePath(FilePathAttribute.BehaviorChoice.Open)]
    public string PictureSource
    {
        get => Picture.Source;
        set => Picture.Source = value;
    }

    public override void Run()
    {
        UpgradeVerdict(Verdict.Pass);
    }
}
```

Below is the test step settings seen on TestPlanner Editor.

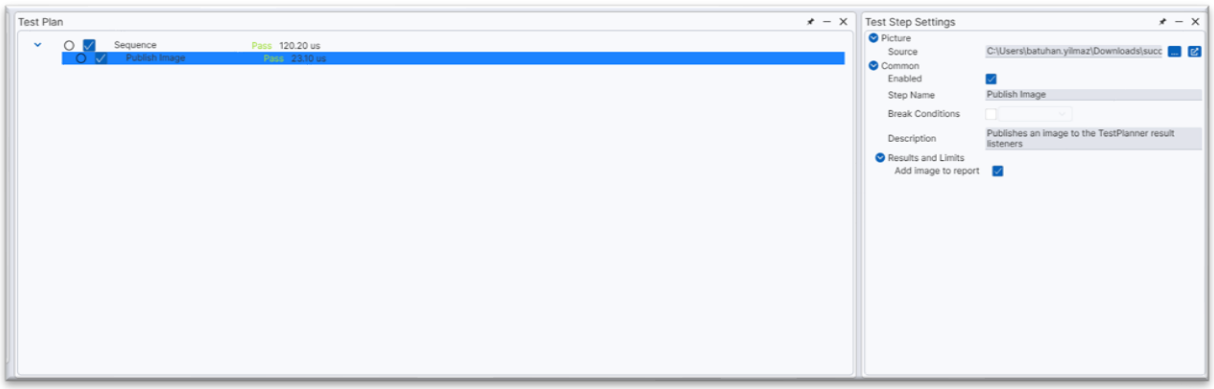


Figure 5: Test Step Settings on TestPlanner Editor

Note This test step does not automatically add an image to the test reports. It must be used in conjunction with a result listener plugin to include the image in the test report.

Informative Test Step

Informative test step is very similar to image publishing steps. Plugin development API provides a base class called **InformativeTestStepBase** for adding extra information to test reports. There is an existing plugin called **Advanced Test Steps** that implements **InformativeTestStepBase**.

If developers aim to implement their own behavior for including additional information on their test reports, they need to create a test step that inherits from **InformativeTestStepBase**.

Note A crucial requirement is that in the **Run()** method of the test step, additional information must be assigned to the property called **Information** provided by the base class.

PublishResults() method is not required in the **Run()** method to publish image to result listeners. An example implementation is provided below:

```
[Display("Publish Info", Description: "Publishes additional custom information to the TestPlanner result listeners", Group: "New Plugin")]
public class InfoStep : InformativeTestStepBase
{
    [Display("Extra info")]
    public string ExtraInfo { get; set; }

    public override void Run()
    {
        Information = ExtraInfo;
        if(this.EnablePublish)
            Log.Info(Information);
    }
}
```

Below is the test step settings seen on TestPlanner Editor.

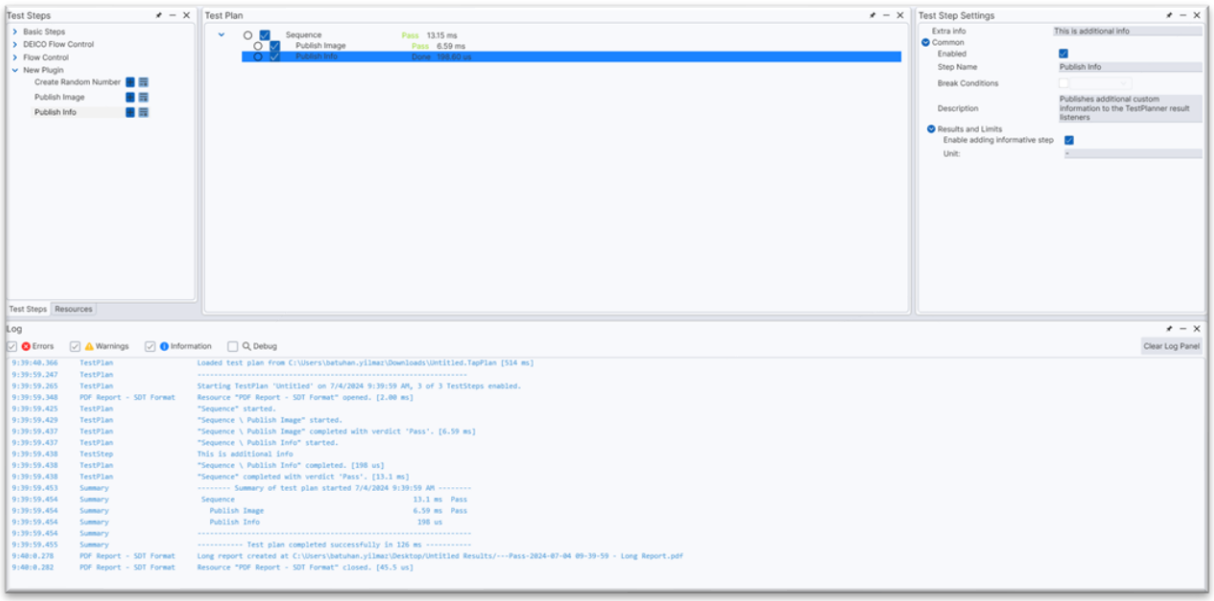
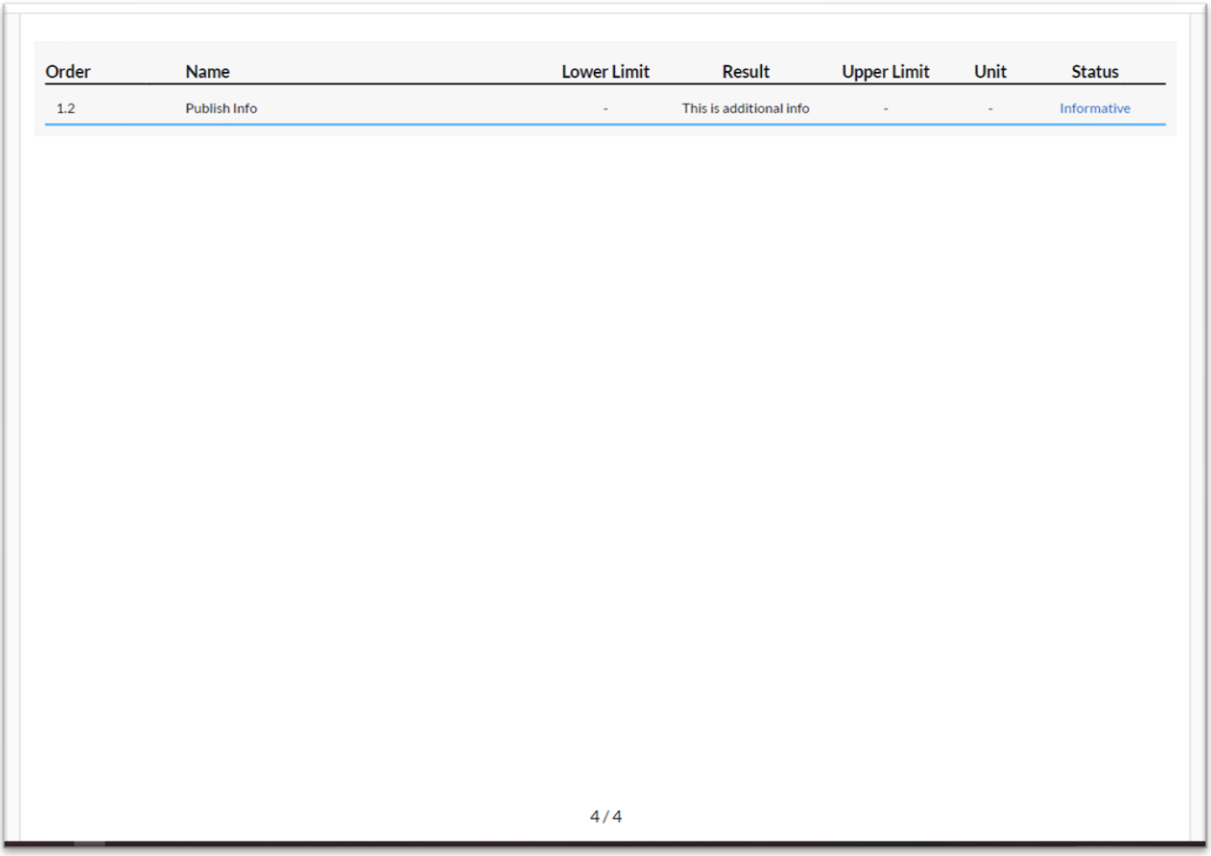


Figure 6: Test Steps Settings on TestPlanner Editor

Additional information is seen like below in a test report.



Order	Name	Lower Limit	Result	Upper Limit	Unit	Status
1.2	Publish Info	-	This is additional info	-	-	Informative

4 / 4

Figure 7: Additional Information in a Test Report

Note This test step does not automatically add additional information to the test reports. It must be used in conjunction with a result listener plugin to include the additional information in the test report. Developing a result listener plugin is out of scope of this document, but briefly **public override void OnResultPublished(Guid stepRun, ResultTable result)** method contains published results in the **ResultTable** parameter.

MIXINS

OpenTAP's Mixins are small units of functionality that can be integrated or 'mixed in' with an object. With the Plugin Development API, developers can achieve both methods of integration, allowing for flexible and dynamic enhancement of test steps.

Mixins can be integrated to plugins during development or in the runtime by providing a **IMixinBuilder** for that mixin. With Plugin Development API, developers can accomplish both.

There are three mixins provided by TestPlanner Plugin Development API.

Repeat Mixin

The Repeat Mixin adds repeating functionality to test steps, allowing a test step to be repeated either until a desired verdict is obtained or for a specified number of repetitions.

When the Repeat Mixin is added to a test step, its settings appear within the test step settings. To configure the Repeat Mixin, the below steps should be followed:

1. **Select repeat condition:** Choose whether to repeat for a specific count or until the expected verdict is satisfied.
2. **Enter repeat count:** If the *repeat for a specific count* option is selected, enter the desired number of repetitions.
3. **Specify expected verdict:** If the *repeat until expected verdict is satisfied* option is selected, specify the expected verdict.
4. **Enter maximum repeat count:** This limits the number of repetitions to prevent infinite loops or excessive repeats.
5. **Enable verdict upgrading (optional):** This option allows users to set the test step's final verdict based on the number of pass/fail counts.

   For example, if this option is enabled and the expected pass count is 3 out of 10 attempts, the test step's verdict will be set as desired once a total of 3 pass verdicts is obtained, and the remaining repeats will be discarded.

Note Repeat mixin can either be integrated to the test step during development or added during run time.

Adding repeat mixin during run time

To add repeat mixin during run time, the steps below must be followed.

1. In TestPlanner Editor, right click on the settings of the test step that repeat mixin will be added to.

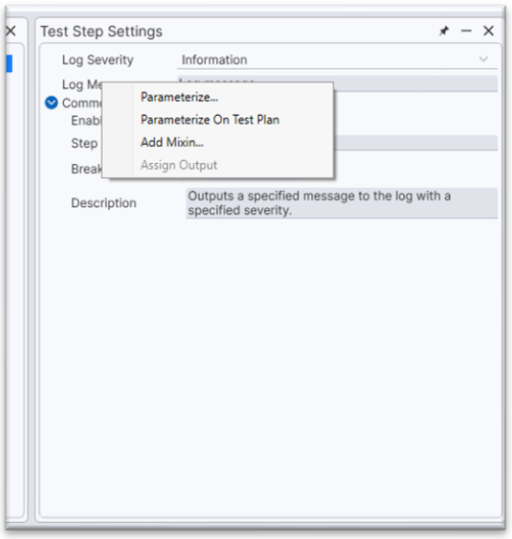


Figure 8: Adding Mixin

2. Click on **Add Mixin** menu item.
3. Select **Repeat Mixin** from the available mixins on the displayed window.

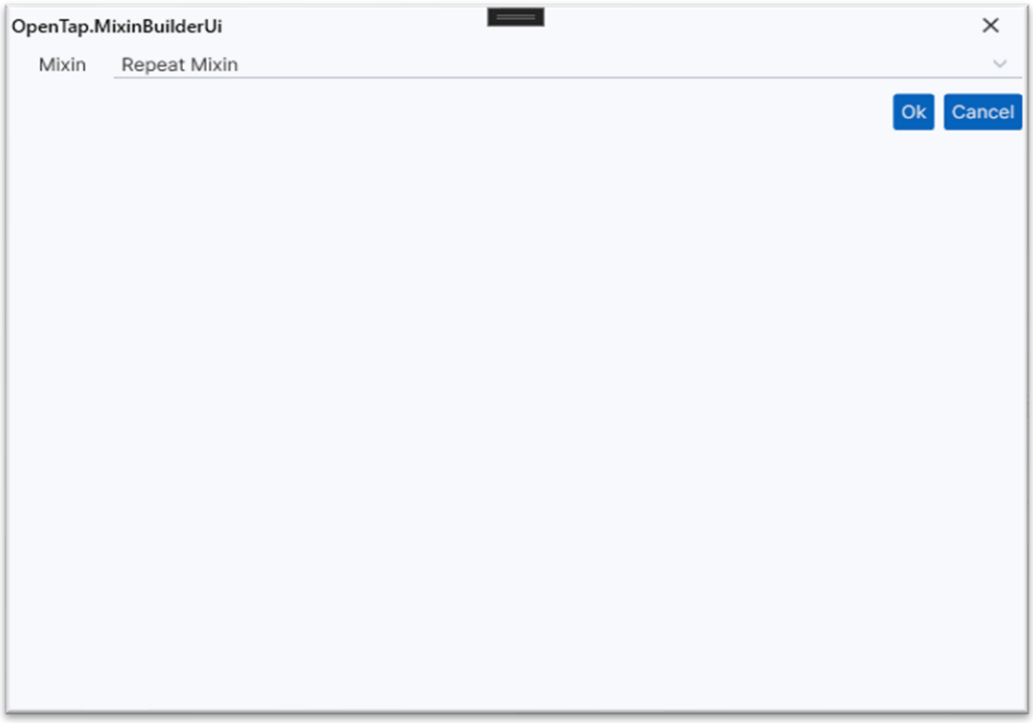


Figure 9: Repeat Mixin Selection

4. Click on **OK** to add the mixin.
5. Configure the mixin settings added to the test step settings.

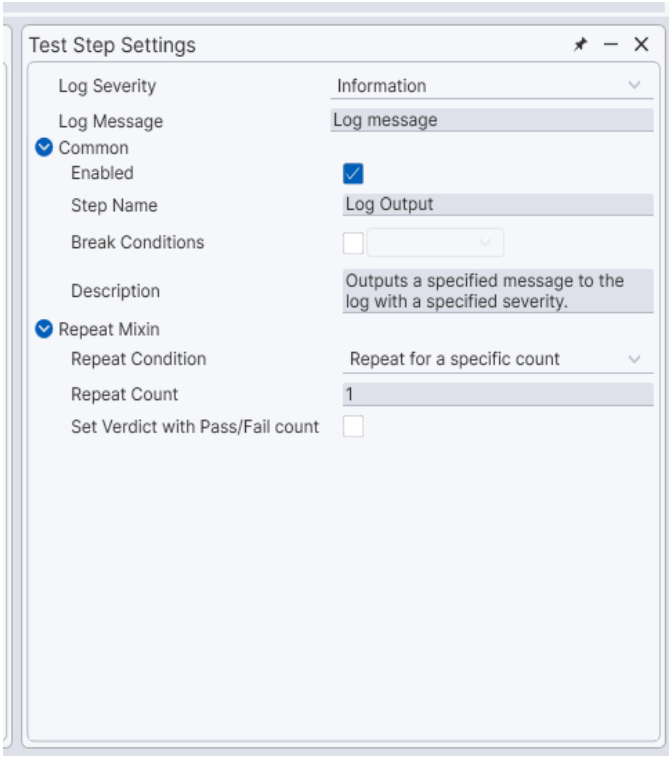


Figure 10: Mixin Settings Configuration

Integrating repeat mixin during development

To integrate repeat mixin during development, the steps below must be followed.

1. Add the following property with the EmbedProperties attribute to the code.

```
[EmbedProperties]
public RepeatMixin Repeat { get; set; } = new RepeatMixin();
```

2. Configure the mixin settings added to the test step settings.

Developers or users can add mixins using either method. Both approaches will achieve the same result, allowing the test step to leverage the repeating functionality provided by the Repeat Mixin.

Note Mixin development is out of scope of this document but adding mixins on run time is a more flexible way of adding mixins. If developers want to develop a mixin, OpenTAP's guide on Mixins can be referred to.

Jump Mixin

The Jump Mixin adds jumping functionality to test steps, allowing the test plan execution to jump to another test step if a specific condition is met.

When the Jump Mixin is added to a test step, its settings appear within the test step settings. To configure the Jump Mixin, the steps below should be followed:

1. **Select expected verdict:** Choose the expected verdict. If this verdict is satisfied, the jump operation will be performed.
2. **Select which step to jump:** Choose the test step to which the jump will be performed. *Only test steps on the same level* in the test plan are jumpable.
3. **Enable maximum jump count (optional):** This option allows users to set a limit on the number of jumps to prevent infinite loops.
4. **Enter maximum jump count (optional):** If the maximum jump count is enabled, specify the maximum number of jumps that can be performed.

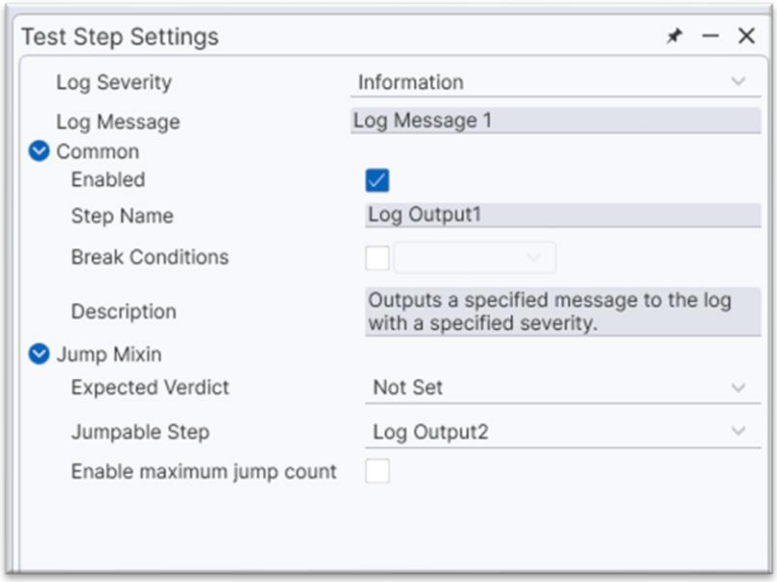


Figure 11: Jump Mixin Configuration

Note Jump Mixin can either be integrated to the test step during development or added during run time, similar to the Repeat Mixin.

Set Last Verdict Mixin

The Set Last Verdict Mixin is used to override a parent test step's verdict. Typically, the verdict of a parent step is determined based on its child steps' verdicts, meaning if any child step fails, the parent step's verdict is also set to fail. However, when test steps are executed repetitively under a parent step and the last execution's verdict is **pass**, the parent step's verdict might still be set to **fail**.

By adding this mixin to a parent step, the parent step's verdict is updated after all the child steps have been executed. This ensures that the final verdict reflects the outcome of the last executed child steps, allowing for more accurate representation of the test results.

Similar to all mixins, when the Set Last Verdict Mixin is added to a test step, its settings appear within the test step settings. Set Last Verdict Mixin contains *only one setting* and it is used to enable or disable the behavior.

Note Set Last Verdict Mixin can either be integrated to the test step during development or added during run time, similar to the other mixins.

Log output of the mixin's operation is provided below:

```
7:53:22.589     Summary                 -------- Summary of test plan started 7/5/2024 7:53:22 AM --------
7:53:22.589     Summary                  Sequence                                          710 us  Pass
7:53:22.589     Summary                    Create Random Number                            53.1 us  Fail
7:53:22.589     Summary                    Create Random Number                            43.4 us  Pass
7:53:22.589     Summary                    Create Random Number                            79.5 us  Fail
7:53:22.589     Summary                    Create Random Number                            78.8 us  Pass
7:53:22.589     Summary                    Create Random Number                            45.4 us  Pass
7:53:22.589     Summary                 ------------------------------------------------------------------
7:53:22.589     Summary                 ---------- Test plan completed successfully in 24.0 ms ----------
```

Figure 12: Log Output of the Mixin's Operation

Note Without Set Last Verdict Mixin, verdict of the **Sequence** step would be set to **fail**.

BEST PRACTICES

Understand the Basics

Developers need to ensure they have a solid understanding of OpenTAP's basics before diving into TestPlanner plugin development. OpenTAP documentation can be referred to for a comprehensive overview.

Error Handling and Debugging

Developers need to ensure their plugins handle errors gracefully, provide meaningful error messages to users, and incorporate logging into their test steps to help with debugging and tracking execution flow.

Plugin Versioning

Devlopers need to ensure to assign appropriate versions to their plugin using the package.xml file. This helps in managing updates and maintaining compatibility.

Additionally, developers must verify that the TestPlanner Plugin Development dependency is correctly specified in the package.xml file. This ensures that their plugin can properly integrate and function with the TestPlanner framework.

TROUBLESHOOTING

Common Issues and Solutions

Visual Studio Plugin Project Template is Missing

- ⇨ **Cause:** In some cases, the plugin project template may not be installed due to the installed version of .NET. Although TestPlanner's setup file automatically installs the required version, this issue can still occur.
- ⇨ **Solution:** Ensure the latest .NET version is installed. Additionally, if multiple versions of Visual Studio are installed on the computer, the project template will only be installed to the latest version. Verify that the latest version of Visual Studio is being used.

Selectable Output List is Empty in ExtTestStep

- ⇨ **Cause:** The **Output** attribute may not be properly added to a property in the test step.
- ⇨ **Solution:** Ensure the **Output** attribute is added to a property with proper accessors (get & set accessors) in the test step.

OpenTAP Version Compatibility

- ⇨ **Cause:** The installed version of OpenTAP may be incompatible with TestPlanner Plugin Development.
- ⇨ **Solution:** TestPlanner Plugin Development is compatible with OpenTAP version 9.22.3 and later. Ensure that version 9.22.3 or later is installed in the plugin project.

String Array Limit

- ⇨ **Cause:** The string array limit is not yet implemented in the TestPlanner Plugin Development API.
- ⇨ **Solution:** This limit type will be added in future versions. Currently, there is no workaround for this limitation.

Contact Support

- ⇨ **Action:** If any issue is encountered that cannot be resolved, contact support. Before doing so, gather relevant information such as error messages, log files, and steps to reproduce the issue. Use the designated support channels provided in the next section to report issues.

Note Providing detailed information will help support staff diagnose and resolve the problem more quickly.

CONTACT & SUPPORT

If encountered any issue, support@deico.com.tr should be contacted, providing the following information:

- ⇨ Detailed description of the issue including steps to reproduce it,
- ⇨ Any error messages and relevant log files,
- ⇨ Specified versions of TestPlanner, OpenTAP, .NET, and any other relevant software,
- ⇨ Project files or code snippets that demonstrate the issue (*if possible*).

DEICO

Contact

DEICO Head Office

Teknopark Ankara, Serhat Mah.,
2224 Cad., No:1 F Blok, Z-12,
Yenimahalle, Ankara, Türkiye

support@deico.com.tr

+90 312 395 68 44

www.deico.com.tr


DEICO