UiPath The Foundation of Innovation™

Workflow Analyzer
カスタムルール開発ガイド

リビジョン履歴

Date	Author	Description
22nd June 2023	Xinyu. D	初版

商標について

- UiPath のソフトウェア、製品、サービス、(これには、UiPath Orchestrator、 UiPath Robot、UiPath Studio が含まれますが、これらに限りません) はアメリカ合衆国で登録された UiPath Inc.、および 他の国・地域で登録された UiPath の関係会社の商標または登録商標です。 UiPath のロゴは UiPath Inc., が所有するものであり、UiPath の事前の明示的な許可なく、お客様及びその他の方が使用することはできません。
- その他、記載されている製品名、会社名およびサービス名はそれぞれの各社の商標または登録商標です。

免責事項

- 本ガイドの内容は 2023 年 6 月現在の情報であり、本ガイドは、下記の製品機能（以下「本製品機能」といいます。）を説明するものです。
  - Workflow Analyzer
- 本製品機能、本製品機能を構成するプログラムまたは本製品機能が依存しもしくは本製品機能と連携する外部サービスがアップデートまたは修正される等によって、本製品機能が本ガイドの説明と異なる動作をする、または仕様となる可能性がありますので、予めご留意ください。当該アップデートまたは修正等の後に本ガイドが改訂された場合には、当該改訂後の本ガイドが本製品機能を説明するものとなります。
- 本ガイドに含まれる情報は可能な限り正確を期しておりますが、本ガイドに記載された内容の正確性、充分性等に関して UiPath 株式会社（以下「UiPath」といいます。）は何ら保証していません。従って、本ガイドに含まれる情報の利用はお客様の責任においてなされるものであり、UiPath はガイドの内容によってお客様が受けたいかなる損害に関して何らの補償をするものではございません。
- 本ガイドは UiPath を法的に拘束する書類ではありません。UiPath はお客様に通知なくして、本ガイドの内容の一部または全部を修正およびアップデートできます。
- 本ガイドの著作権などの一切の知的財産権は、UiPath またはその関係会社に帰属します。お客様は UiPath の書面の承諾なしで本ガイドを複製、修正、頒布、公衆送信等できません。

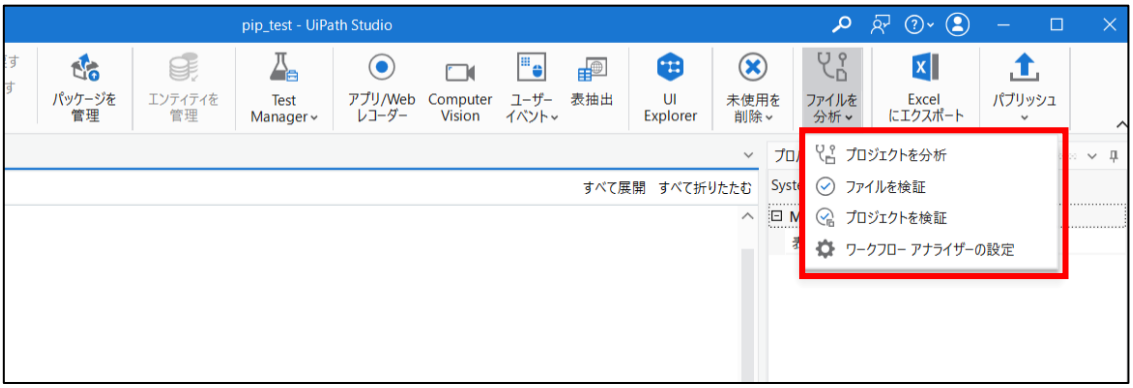
内容

はじめに

- 本文書では Workflow Analyzer のカスタムルールの作成方法について例示・解説しております。なお、記述内容は 23.4.1 バージョンをもとに構成しています。
- 「Windows レガシ」プロジェクトと「Windows」プロジェクトは.NET のランタイム（それぞれバージョン 4.6.1 と 6）が異なるため、開発手順の差分についても解説しています。

UiPath Workflow Analyzer について

UiPath Workflow Analyzer は Studio/StudioX によって開発されたプロジェクトの実装内容を自動チェックできるツールです。Workflow Analyzer は Studio/StudioX に同梱されていて、定義されたルールに基づいて、実行とパブリッシュ時に自動的に実装内容をチェックできます。



より詳しい説明は 公式ドキュメント よりご確認できます。

カスタムルールとは

UiPath Workflow Analyzer では、実装のベストプラクティスに基づいたルールを既定で提供していますが、既定のルールで対応できないチェック内容に対して、カスタムのルールを作成することが可能です。カスタムのルールはプロジェクト、ワークフローファイル、アクティビティのいずれかに対して作成でき、以下の表で示された項目に関連する内容をチェックできます。なお、チェックできる内容は UiPath アクティビティ SDK の内容に従うため、変更される可能性があります。最新情報は SDK の公式ドキュメント をご参照ください。

チェック対象	SDK エンティティ名	動作概要	チェックできる内容（例）
プロジェクト	IProjectModel	プロジェクトを分析したときに 1 回だけ実行	● プロジェクトフォルダ構成 ● 起点処理 ● エラーハンドリング ● 依存関係 ● プロジェクト名 ● 実行時のユーザー入力の有無 ● モダンデザインの利用有無 ● 利用テンプレート
ワークフローファイル	IWorkflowModel	ワークフローファイルひとつにつき 1 回実行	● ルートアクティビティ ● アクティビティの詳細エディターの記述内容 ● 利用アクティビティ ● 引数 ● 利用変数
アクティビティ	IActivityModel	アクティビティひとつにつき 1 回実行。Studio で、アクティビティスコープのエラーメッセージをダブルクリックすると、当該のアクティビティにジャンプする。	● アクティビティ名 ● 親要素 ● 子要素 ● 利用変数 ● 引数 ● プロパティ ● 注釈内容 ● オブジェクト参照

開発環境の構築

UiPath Workflow Analyzer のカスタムルールは C#言語と.NET フレームワークを利用して開発されます。カスタムルールを開発するためには、C#言語と.NET フレームワークをサポートする開発ツールを利用する必要があります。なお、本資料は Microsoft Visual Studio を利用して解説しています。開発に着手する前に準備しておく必要があるツールは以下の表をご参照ください。

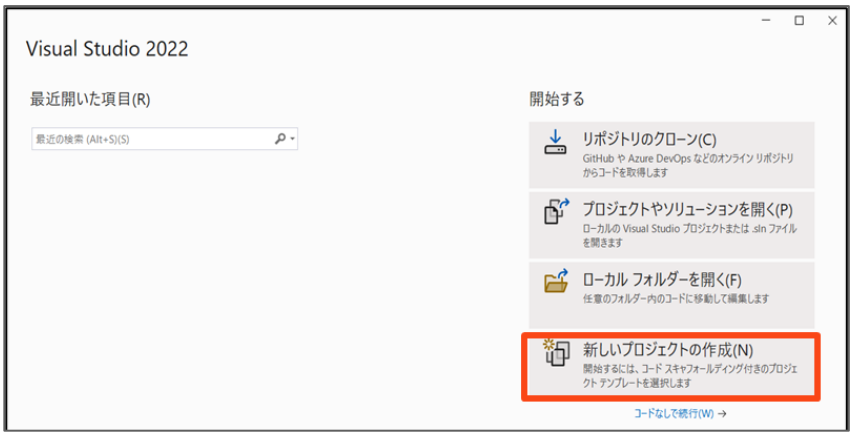
名称	補足
.NET Framework	● プロジェクトの OS によって利用する.NET Framework のバージョンが異なります。Windows レガシプロジェクトの場合は 4.X, Windows の場合は 6 を利用する必要があります。 ● 開発 IDE として Visual Studio を利用する場合、.NET Framework は Visual Studio と同時にインストールできます。
開発 IDE	● 本資料では Microsoft Visual Studio を利用して解説しています。 ● Visual Studio インストーラーの「個別コンポーネント」画面で.NET 4.X の SDK と Targeting Pack、.NET 6 のランタイムにチェックを入れてインストールすれば、開発用の.NET フレームワークをインストールできます。

ルールの実装

.NET フレームワークを利用したクラスライブラリ プロジェクトを作成してカスタムルールの実装を行います。実装は C# 言語と UiPath アクティビティ SDK を利用します。UiPath アクティビティ SDK の詳細は SDK の公式ドキュメント より確認できます。

① プロジェクト作成

Microsoft Visual Studio を起動して、「新しいプロジェクトの作成」をクリックして、カスタムルール用のプロジェクトを新規作成します。

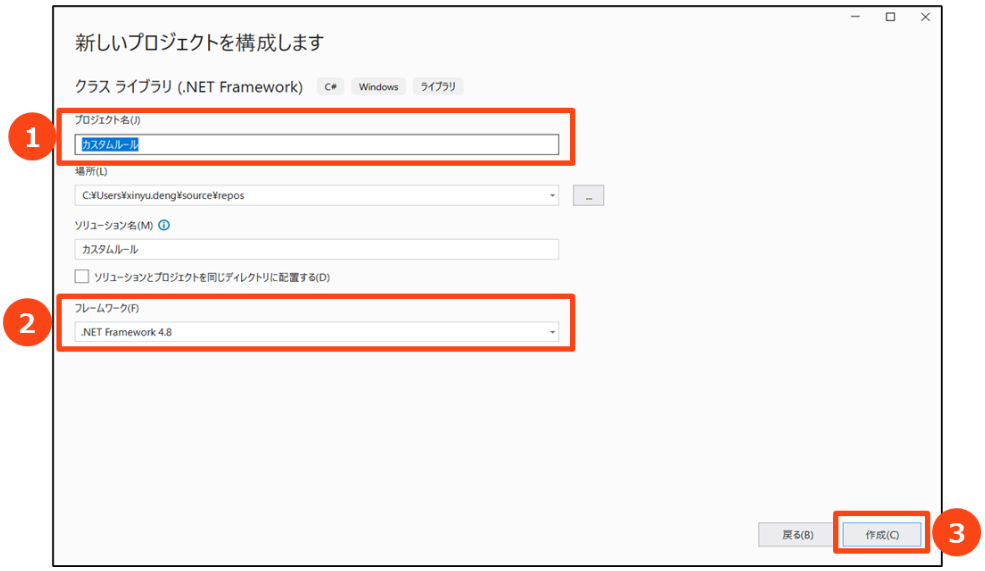


複数のカスタムルールを作成する予定であっても、1 つのプロジェクトにまとめることが可能ですが、カスタムルールの適用対象の RPA プロジェクトの対応 OS ごとにプロジェクトを分ける必要があります。なお、対応 OS が Windows レガシか Windows かによってプロジェクトの作成方法が異なるため、個別に解説します。

- Windows レガシの場合

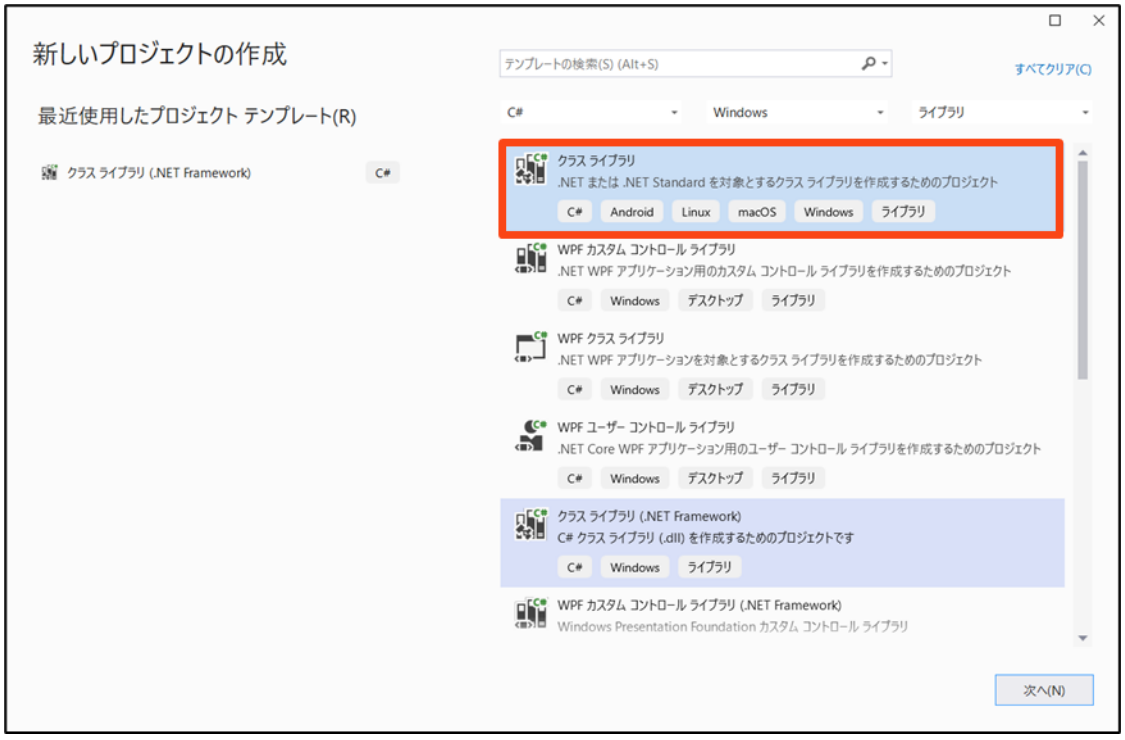
(1)言語が C# の「クラス ライブラリ (.NET Framework)」テンプレートを選択して、「次へ」をクリックします。

(2)プロジェクト名を入力し、フレームワークを選択して「作成」を実行します。「フレームワーク」の選択では実行環境と同様の.NET バージョンを選択してください。



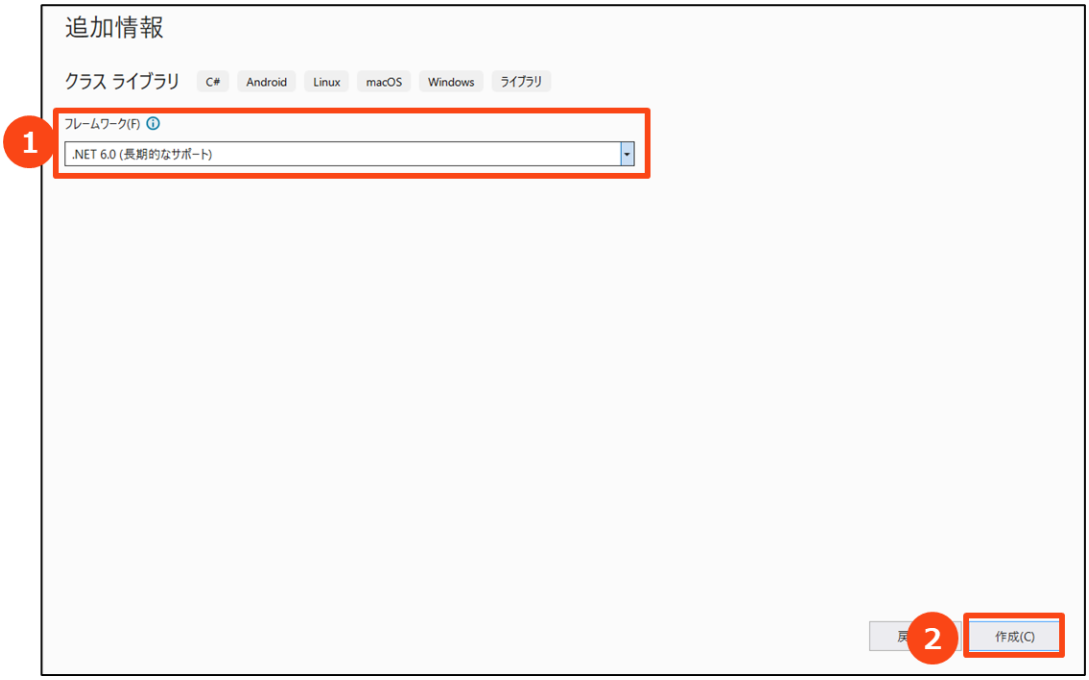
- Windows の場合

(1)言語が C# の「クラス ライブラリ」テンプレートを選択して、「次へ」をクリックします。



(2)プロジェクト名とソリューション名を入力して、「次へ」をクリックします。

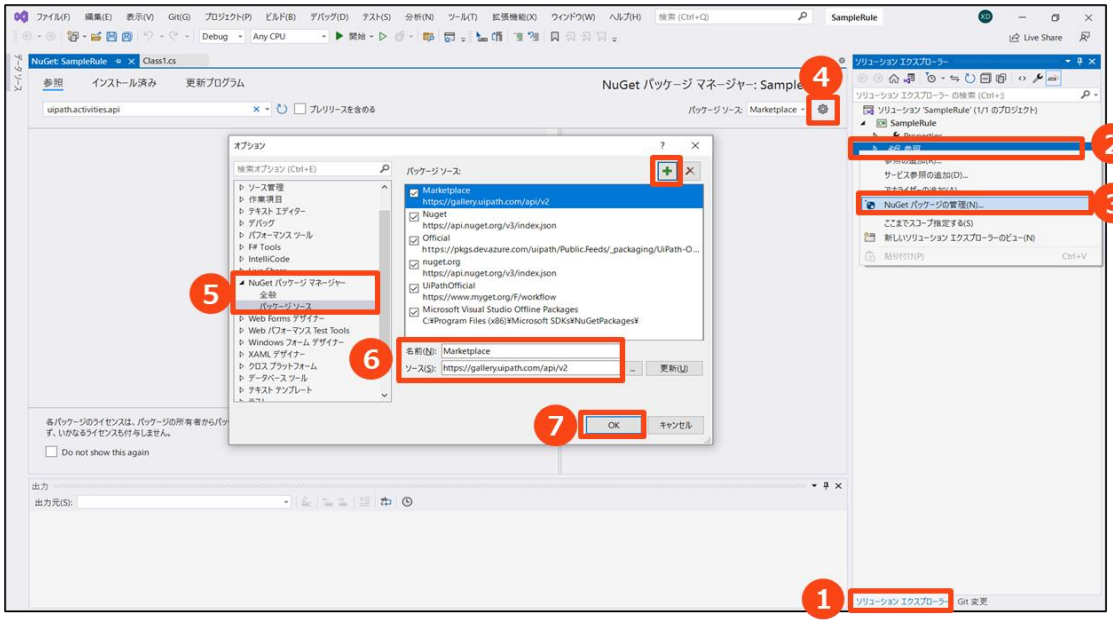
(3)フレームワークで「.NET6.0(長期的なサポート)」を選択して、「作成」をクリックします。



② UiPath アクティビティ SDK の追加

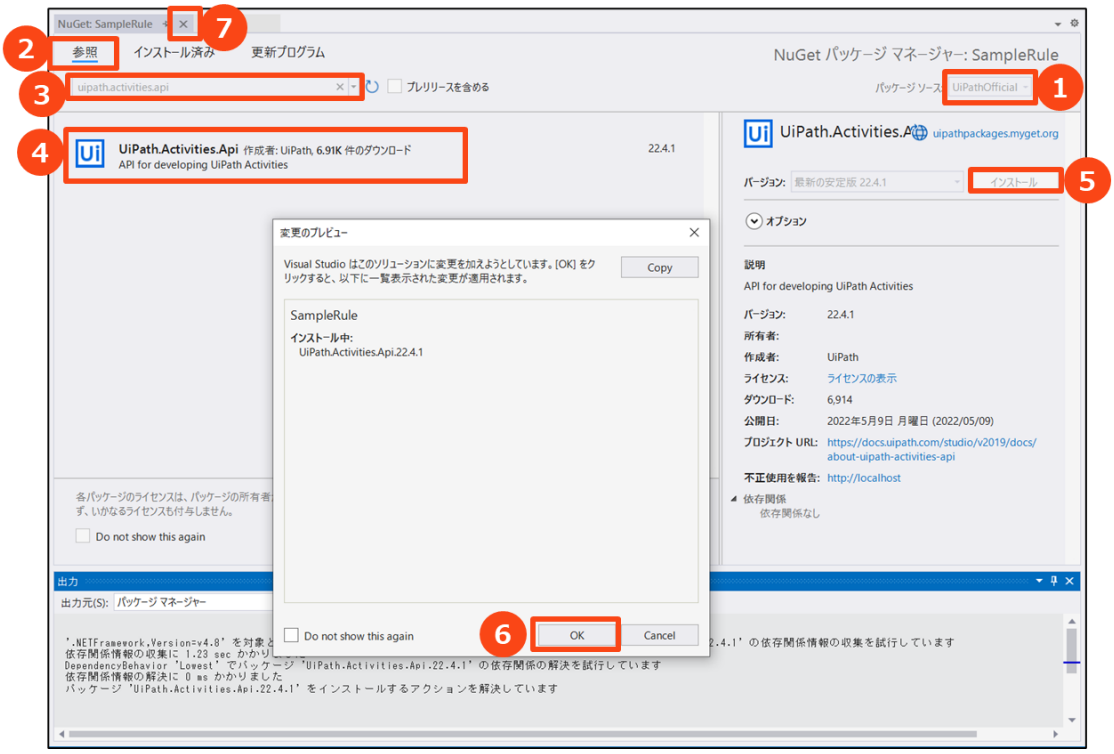
(1)右下の「ソリューションエクスプローラー」タブを開いて、「参照」に右クリックして「NuGet パッケージの管理」を選択し、次の名前とソースでパッケージソースを追加します。なお、対象 URL がすでに追加済みの場合は本ページの内容をスキップしてください。

- 名前：UiPathOfficial
- ソース：https://pkgs.dev.azure.com/uipath/Public.Feeds/_packaging/UiPath-Official/nuget/v3/index.json



(2)「パッケージソース」で「UiPathOfficial」を選択し、「参照」タブで「uipath.activities.api」を検索して選択します。

「インストール」を実行し、表示されたダイアログで「OK」をします。インストール完了後に NuGet 管理画面を閉じます。



③ 実装

(1)Class1.cs ファイルを開き、既存の「public class Class1」を削除します。

(2)以下の図で示した通り、namespace の直下に以下のクラスを追加し、クラス内に Get、Inspect メソッドを追加します。クラス「TODO」の部分はチェック対象の SDK エンティティ名（IProjectModel、IWorkflowModel、IActivityModel のいずれか）に置き換えてください。なお、本資料はアクティビティ（IActivityModel）を利用して解説しています。

- クラス

  internal static class {ルールの名前}

- メソッド

  internal static Rule<TODO> Get()

  private static InspectionResult Inspect(TODO inspectionModel, Rule ruleInstance)

```
namespace SampleRule_2305
{
    0 個の参照
    internal static class Rule1
    {
        0 個の参照
        internal static Rule<TODO> Get()
        {

        }

        0 個の参照
        private static InspectionResult Inspect(TODO inspectionModel, Rule ruleInstance)
        {

        }
    }

}
```

【補足】

- 作成されたクラスの内部でカスタムルールの内容を実装していきます。後ほど実装方法を詳しく説明しますが、Get メソッドでカスタムルールの名前と引数などの定義情報を定義し、Inspect メソッドでチェックロジックを実装していきます。
- Inspect メソッドの引数として、inspectionModel と ruleInstance があります。inspectionModel にはチェック対象のエンティティ(アクティビティ、ワークフローなど)の内容が格納されていて、ruleInstance には get メソッドで定義されたルールの定義情報が格納されています。

(3)Get メソッド内にカスタムルールの定義情報を追加し、以下の図の赤枠に囲まれた箇所は適切な内容に置き換えてください。

【コード】

```
var rule = new Rule<IActivityModel>("ルールの日本語名", "SAMPLE-RULE-01", Inspect)
{
    RecommendationMessage = "ルールの説明文",
    ErrorLevel = System.Diagnostics.TraceLevel.Warning,
    ApplicableScopes = new List<string> { RuleConstants.BusinessRuleConstant }
};
```

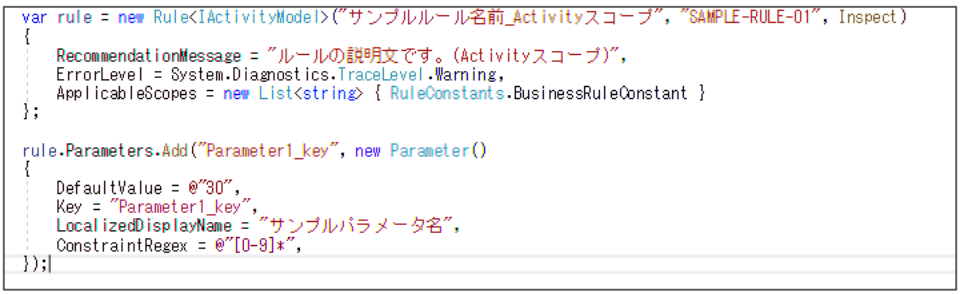
【補足】

- IActivityModel の部分はステップ(2)と同様の SDK エンティティ名に置き換えてください。
- “SAMPLE-RULE-01”の部分は英数字のコードです。（既定のルールの命名規則については「④実装の Tips」をご参照ください。
- ErrorLevel は Error、Warning、Info のいずれかを選択できます。Error にする場合、ルール違反のワークフローをパブリッシュできません。
- ApplicableScopes を追加しますと、該当ルールは StudioX でも利用できるようになります。Studio のみのルールは ApplicableScopes 行を追加しないでください。
- 既定のルールは以下の規則に従って命名されています。カスタムルールを作成する際にそのまま流用する必要はありませんが、参考情報として掲載させていただきます。
  - ST-NMG-00x 命名規則（Naming Rules）
  - ST-DBP-00x デザインのベストプラクティス（Design Best Practices）
  - ST-ANA-00x プロジェクト構造（Project Anatomy Rules）
  - ST-MRD-00x 保守性とわかりやすさ（Maintainability and Readability Rules）
  - ST-USG-00x 使用法（Usage Rules）
  - ST-PRR-00x パフォーマンスと再利用性（Performance and Reusability Rules）
  - ST-REL-00x 信頼性（Reliability Rules）
  - ST-SEC-00x セキュリティ（Security Rules）

(4)カスタムルールに引数を設定したい場合、引数の定義情報を追加します。引数の設定が不要な場合は本ステップをスキップしてください。

【コード】

```
 rule.Parameters.Add("Parameter1_key", new Parameter()
{
    DefaultValue = "30",
    Key = "Parameter1_key",
    LocalizedDisplayName = "サンプルパラメータ名",
    ConstraintRegex = @"[0-9]*",
});
```



```
var rule = new Rule<IActivityModel>("サンプルルール名前_Activityスコープ", "SAMPLE-RULE-01", Inspect)
{
    RecommendationMessage = "ルールの説明文です。(Activityスコープ)",
    ErrorLevel = System.Diagnostics.TraceLevel.Warning,
    ApplicableScopes = new List<string> { RuleConstants.BusinessRuleConstant }
};

rule.Parameters.Add("Parameter1_key", new Parameter()
{
    DefaultValue = @"30",
    Key = "Parameter1_key",
    LocalizedDisplayName = "サンプルパラメータ名",
    ConstraintRegex = @"[0-9]*",
});
```

【補足】

- 複数の引数を設定する場合、上記のブロックを引数ごとに追加してください。
- 引数の各項目の意味
  - DefaultValue：既定値
  - Key：英文字の一意キー
  - LocalizedDisplayName：日本語表示名
  - ConstraintRegex：引数値の入力制限（正規表現で記述）
- ConstraintRegex と DefaultValue で正規表現を利用する場合、文字の前に「@」を追加すると各記号のエスケープが不要になります。
- 既定値と入力制限が不要な場合、DefaultValue と ConstraintRegex を削除してください。

(5)定義したルールのエンティティを戻り値として設定します。

【コード】

return rule;

```
rule.Parameters.Add("Parameter1_key", new Parameter()
{
    DefaultValue = @"30",
    Key = "Parameter1_key",
    LocalizedDisplayName = "サンプルパラメータ名",
    ConstraintRegex = @"[0-9]*",
});

return rule;
```

(6)Inspect メソッドの中に、引数の値を取得する処理を追加します。引数がない場合はこのステップをスキップしてください。

【コード】

```
var sampleParameter = ruleInstance.Parameters["Parameter1_key"]?.Value;
if (string.IsNullOrWhiteSpace(sampleParameter))
{
    return new InspectionResult() { HasErrors = false };
}
```

```
private static InspectionResult Inspect(IActivityModel inspectionModel, Rule ruleInstance)
{
    var sampleParameter = ruleInstance.Parameters["Parameter1_key"]?.Value;
    if (string.IsNullOrWhiteSpace(sampleParameter))
    {
        return new InspectionResult() { HasErrors = false };
    }

}
```

【補足】

- 複数の引数がある場合、一つずつ取得してください。
- 上記のコードでは、引数が空白の場合にチェック処理をせずに終了していますが、引数が空白の場合でもチェック処理を実行する場合は if ブロックを削除してください。

(7)チェック結果を格納する List を作成します。

【コード】

var messageList = new List<string>();

```
var sampleParameter = ruleInstance.Parameters["Parameter1_key"]?.Value;
if (string.IsNullOrWhiteSpace(sampleParameter))
{
    return new InspectionResult() { HasErrors = false };
}

var messageList = new List<string>();
```

(8)inspectionModel からチェック処理に利用する情報を取得し、チェック処理を実装します。なお、以下の実装例では、アクティビティのタイムアウト時間をチェックし、ルールの引数で設定された数値以上の場合に警告（Warning）を表示しています。

【コード（実装例）】

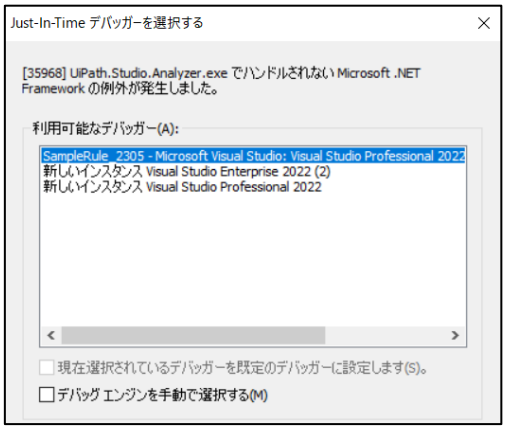
var arguments = inspectionModel.Arguments;

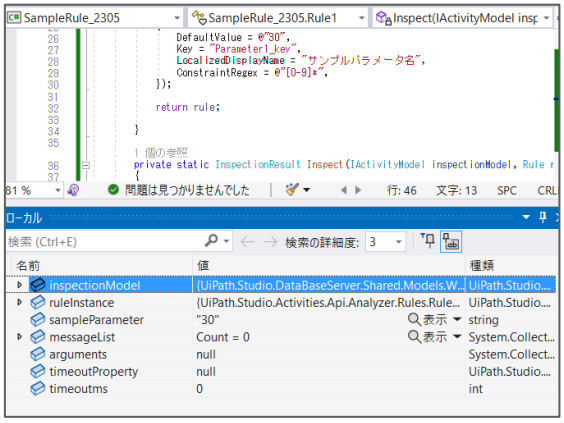
var timeoutProperty=arguments.FirstOrDefault(p => { return p.DisplayName == "タイムアウト (ミリ秒)"; });
if (timeoutProperty != null && Int32.TryParse(timeoutProperty.DefinedExpression, out var timeoutms))
{
    if (timeoutms > Int32.Parse(sampleParameter))
    {
        messageList.Add("タイムアウト時間が推奨の上限を超えています。");
    }
}

```
var messageList = new List<string>();

var arguments = inspectionModel.Arguments;

var timeoutProperty = arguments.FirstOrDefault(p => { return p.DisplayName == "タイムアウト (ミリ秒)"; });
if (timeoutProperty != null && Int32.TryParse(timeoutProperty.DefinedExpression, out var timeoutms))
{
    if (timeoutms > Int32.Parse(sampleParameter))
    {
        messageList.Add("タイムアウト時間が推奨の上限を超えています。");
    }
}
```

【補足】

- inspectionModel から取得できる情報はチェック対象のエンティティによって異なります。詳細は SDK の公式ドキュメントより確認できます。
- inspectionModel から取得できる情報は基本的に SDK ドキュメントを参照すれば理解できますが、IActivityModel の Arguments と Properties について、簡単に補足いたします。
  - Arguments：アクティビティのプロパティパネルで表示されている項目
  - Properties：表示名、ID などの内部プロパティ。なお、UIAutomation アクティビティのターゲットも Properties から取得する必要があります
- 実装内容の動作をテストしたい場合、ステップ(9)~(13)・パブリッシュ・展開を完了させた上で、Studio から Workflow Analyzer を実行する必要があります。
- テスト中にチェック対象のエンティティの詳細をリアルタイムに確認したい場合、チェック処理の前に以下のコードを一行追加してください。このコードを追加すると、パブリッシュ・展開されたルールが実行されると、Visual Studio のデバッグツールが起動され、エンティティの内容などの実行情報を確認できます。なお、開発が完了したタイミングで追加したデバッグコードを削除する必要があります。
  - System.Diagnostics.Debugger.Launch();

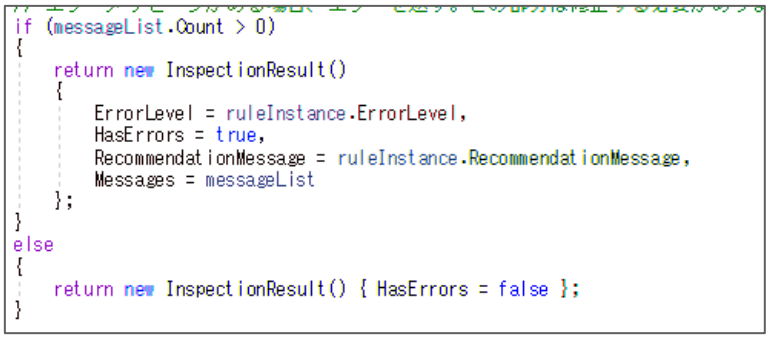




(9)チェック結果を戻り値に設定します。エラーがない場合、空のチェック結果を戻り値に設定します。

【コード】

```
if (messageList.Count > 0)
{
    return new InspectionResult()
    {
        ErrorLevel = ruleInstance.ErrorLevel,
        HasErrors = true,
        RecommendationMessage = ruleInstance.RecommendationMessage,
        Messages = messageList
    };
}
else
{
    return new InspectionResult() { HasErrors = false };
}
```



【補足】

- エラーがある場合、ErrorLevel と RecommendationMessage をエラーメッセージとともに戻り値に設定する必要がありますが、事前に get メソッドで定義した内容を ruleInstance から取得して利用してください。
- messageList を戻り値に設定するため、複数のエラーメッセージがある場合でも対応できます。

(10)　複数のカスタムルールを作成する場合、各ルールに対してステップ(2)~(9)を実施してください。

(11) 以下のいずれかの方法でカスタムルールを登録します。

- 登録インターフェイスメソッド（推奨）
  - Studio バージョン 2019.10 以降でのみ使用可能
  - ステップ(12)以降で詳しく説明
- IRegisterMetadata メソッド
  - Studio バージョン 2019.6 以降で使用可能
  - 登録インターフェイスメソッドほど適切ではないため、本資料では割愛いたします。詳細は公式ドキュメントをご参照ください。

(12) 以下のクラスを追加し、その中に以下のメソッドを追加します。このメソッドにカスタムルールを Studio に認識させるための処理を追加していきます。

- クラス

public class RegisterAnalyzerConfiguration : IRegisterAnalyzerConfiguration

- メソッド

public void Initialize (IAnalyzerConfigurationService workflowAnalyzerConfigService)

```
        else
        {
            return new InspectionResult() { HasErrors = false };
        }

    }
}


0 個の参照
public class RegisterAnalyzerConfiguration : IRegisterAnalyzerConfiguration
{
    0 個の参照
    public void Initialize(IAnalyzerConfigurationService workflowAnalyzerConfigService)
    {

    }
}
```

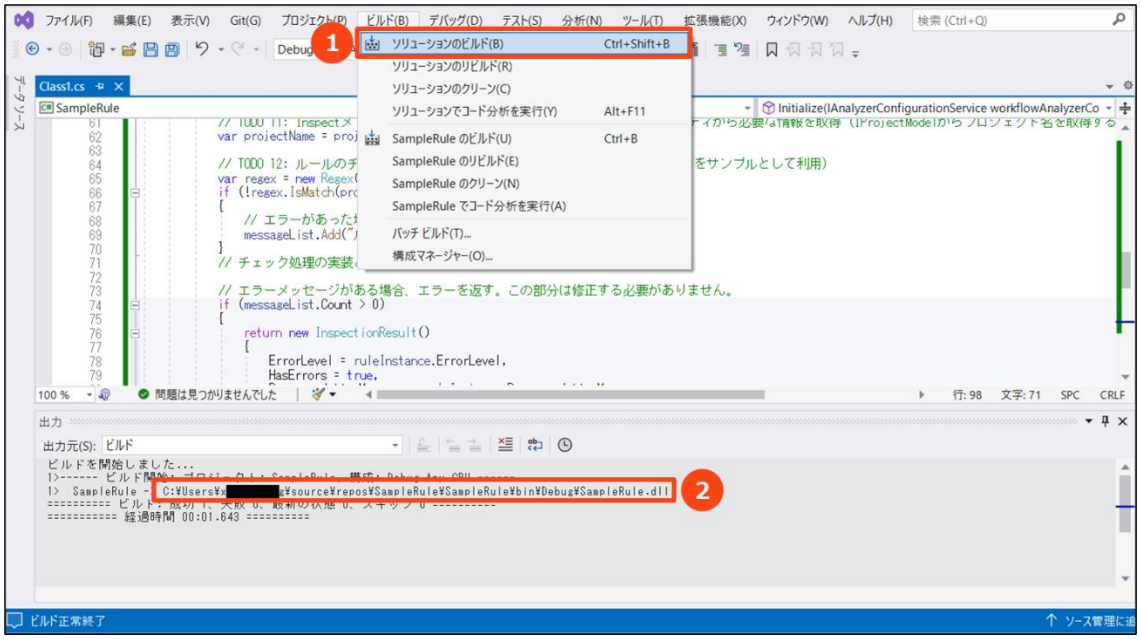
(13) 作成したルールを Studio に認識させる処理を追加します。なお、「Rule1」の部分はステップ(2)で追加したクラスの名前に置き換えてください。複数のルールがある場合、1 行ずつ Initialize メソッド内に追加する必要があります。

```
0 個の参照
public class RegisterAnalyzerConfiguration : IRegisterAnalyzerConfiguration
{
    0 個の参照
    public void Initialize(IAnalyzerConfigurationService workflowAnalyzerConfigService)
    {
        workflowAnalyzerConfigService.AddRule(Rule1.Get());
    }
}
```

パブリッシュ

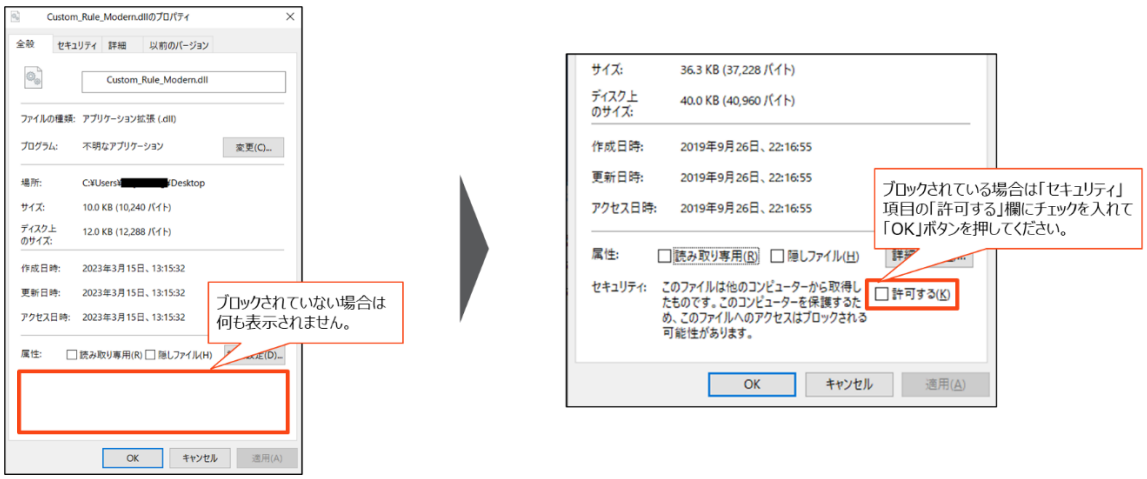
実装されたカスタムルールをテスト・利用するために、実装物を.dll ファイルにパッケージ化し、所定のフォルダーに格納する必要があります。また、該当のルールを開発テンプレートとともに展開する場合、.dll ファイルの他に、NuGet パッケージを出力する必要があります。NuGet パッケージの出力手順は本資料の「追加手順」セクションをご参照ください。
なお、開発テンプレートとともにルールを展開する場合、ルールの NuGet パッケージがないと実行できなくなるため、RPA 実行環境でも該当の NuGet パッケージが必要になります。保守運用のコストが増えるリスクがあるため、積極的な利用は推奨しません。

(1)Visual Studio の「ビルド」 > 「ソリューションのビルド」をクリックして、.dll ファイルを作成します。作成が成功した場合、.dll ファイルの出力先が表示されます。NuGet パッケージ作成する場合、この出力先は NuGet パッケージの作成でも利用されます。



【補足】

- 一度ビルドを実行してパブリッシュした後に、実装内容を修正して再パブリッシュする場合、「ソリューションのビルド」ではなく、「ソリューションのリビルド」を実行する必要があります。
- 他人が作成した.dll ファイルをインターネット経由でダウンロードした場合、Windows のセキュリティ機能により実行がブロックされる場合があるので、展開する前にブロックを解除する必要があります。該当.dll ファイルがブロックされている場合、右クリックの「プロパティ」メニューの「全般」タブに「セキュリティ」という名前の項目が表示されます。その右側の「許可する」欄にチェックを入れて「OK」ボタンを押すとブロックを解除できます。



展開

パブリッシュされた.dll ファイルを各対象端末の適切なフォルダーに格納して Studio/StudioX を再起動すれば、.dll ファイルに含まれるルールは対象端末にインストールされた Studio/StudioX で表示されるようになります。

【既定の.dll ファイルの格納フォルダー】

- バージョン 2021.10 以前の Studio の場合
  - マシン単位のインストールの場合
    %ProgramFiles%¥UiPath¥Studio¥Rules
  - ユーザー単位のインストールの場合
    %LocalAppData%¥Programs¥UiPath¥Studio¥Rules
- バージョン 2021.10.6 以降の Studio の場合
  - マシン単位のインストールの場合
    - 適用対象の RPA プロジェクトの OS が Windows かクロスプラットフォームの場合
      %ProgramFiles%¥UiPath¥Studio¥Rules¥net6.0
    - 適用対象の RPA プロジェクトの OS が Windows レガシの場合
      %ProgramFiles%¥UiPath¥Studio¥net461¥Rules
  - ユーザー単位のインストールの場合
    - 適用対象の RPA プロジェクトの OS が Windows かクロスプラットフォームの場合
      %LocalAppData%¥Programs¥UiPath¥Studio¥Rules¥net6.0
    - 適用対象の RPA プロジェクトの OS が Windows レガシの場合
      %LocalAppData%¥Programs¥UiPath¥Studio¥net461¥Rules

【補足】

- カスタムルールの格納先フォルダーを既定値から変更したい場合、Studio/StudioX の「ホーム > 設定 > 場所」画面を開いて、「ワークフローアナライザーのカスタムルールの場所」より設定できます。この機能はバージョン 2020.10 以降の Studio/StudioX に搭載されています。
- 32bit 版インストーラーの場合、%ProgramFiles%を%ProgramFiles(x84)%に置き換えてください。

追加手順：開発テンプレートとともにカスタムルールを展開する場合

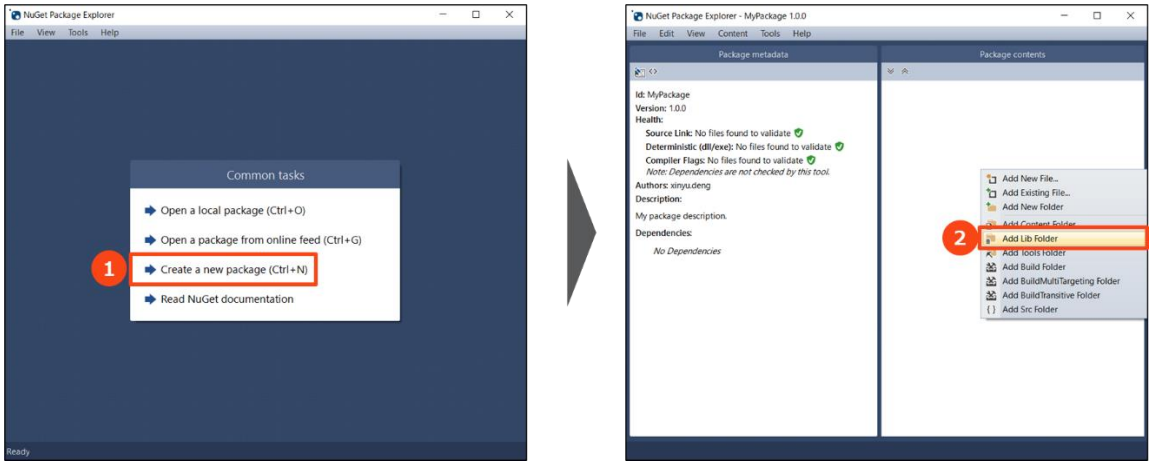
各開発用の端末に.dll ファイルを配布せずに、カスタムルールの NuGet パッケージを開発テンプレートにインストールして展開することが可能です。なお、開発テンプレートとともにルールを展開する場合、ルールの NuGet パッケージがないとワークフローが実行できなくなるため、RPA 実行環境でも該当の NuGet パッケージが必要になります。保守運用のコストが増えるリスクがあるため、積極的な利用は推奨しません。

(1)環境セットアップ

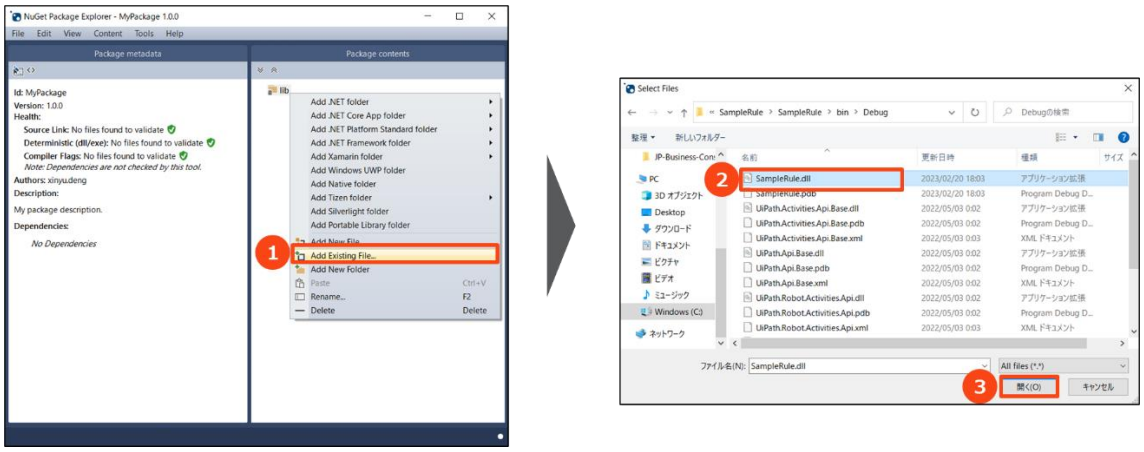
NuGet パッケージを作成するために、以下のツールをインストールする必要があります。

名称	補足
NuGet パッケージ作成ツール	● 本資料では Microsoft Store より無料で入手可能な NuGet Package Explorer を利用して解説しています。 ● コマンドラインで作成することも可能ですが、本資料では割愛させていただきます。

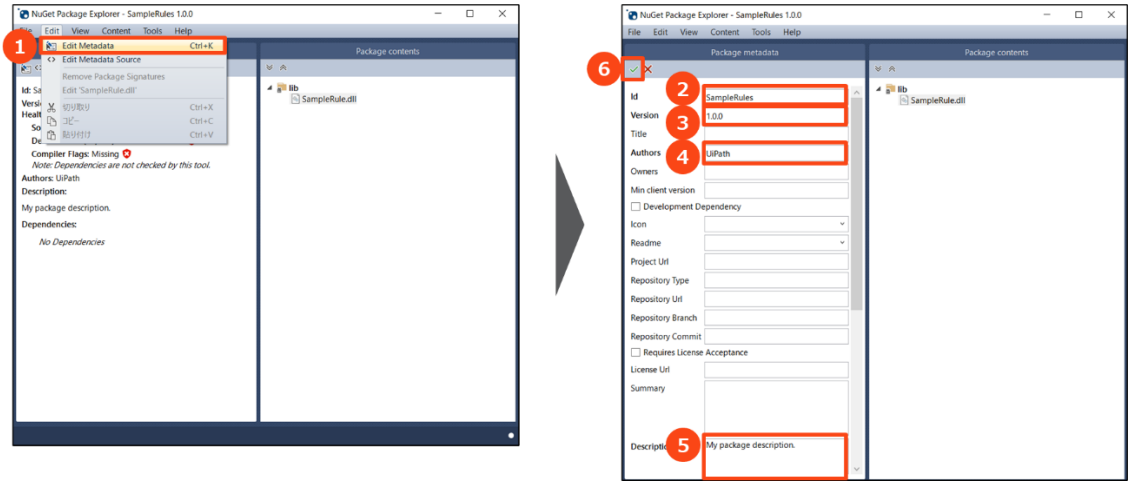
(2)NuGet Package Explorer を起動し、「Create a new package」を選択します。表示された画面の右側の「Package contents」エリアに右クリックし、「Add Lib Folder」を選択します。



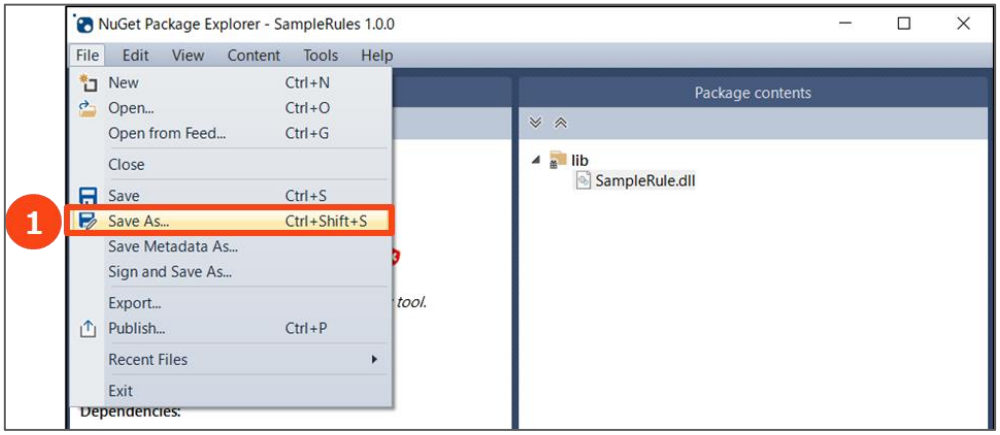
(3)「Package contents」に表示された「lib」に右クリックし、「Add Existing File」を選択します。出力された.dll ファイルを選択し、「開く」を実行します。



(4)「Edit」 > 「Edit Metadata」を実行し、パッケージの ID、Version（バージョン）、Author（作成者）、Description（説明文）を適切に編集して保存します。



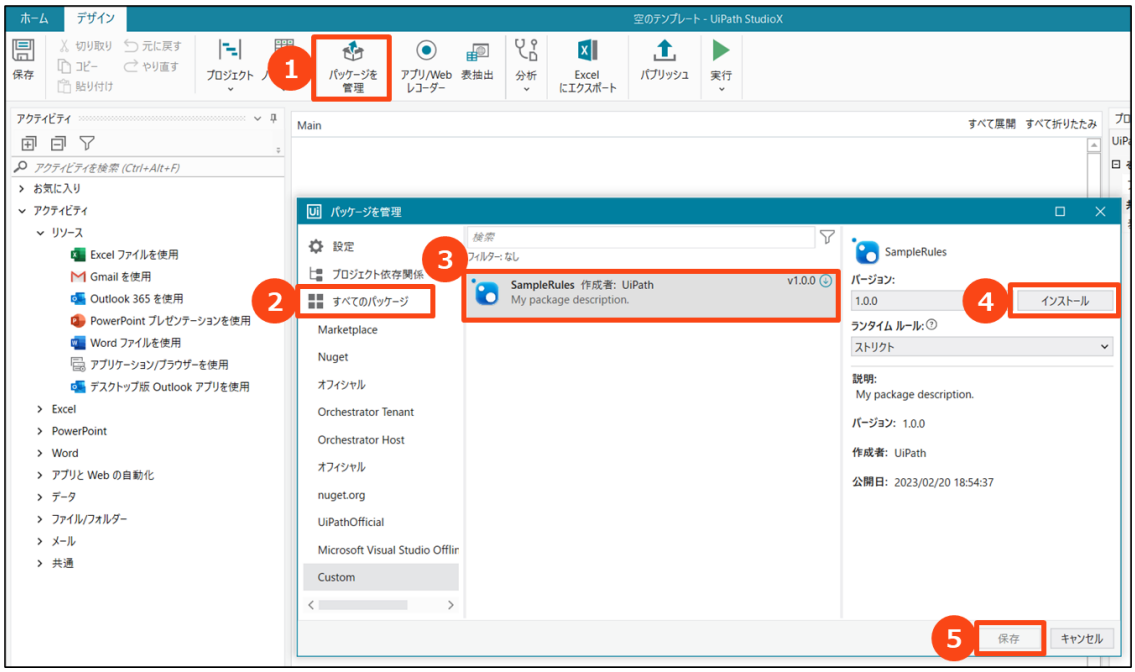
(5)「File」 > 「Save As」を実行し、パッケージを PC ローカルに保存します。



(6)Studio/StudioX を起動し、テンプレートを新規作成します。なお、新規作成時の「対応 OS」の選択では、カスタムのルールと同じ対応 OS を選ぶ必要があります。.NET 6 でルールを作成した場合は「Windows」、.NET 4.X でルールを作成した場合は「Windows レガシ」を選択してください。

(7)ルールの NuGet パッケージをテンプレート利用者がアクセスしやすいように、該当パッケージを以下のページを参照して、Orchestrator にアップロードすることを推奨します。
https://docs.uipath.com/orchestrator/lang-ja/docs/managing-packages

(8)Studio/StudioX で「パッケージを管理」を起動し、追加したいカスタムルールのパッケージを検索して、インストールします。なお、「パッケージを管理」の仕様は Studio と StudioX 共通なので、Studio 版のスクリーンショットは割愛します。



(9)Studio/StudioX で「パブリッシュ」画面を起動し、必要な情報を入力して、「パブリッシュ」を実行します。
自動化プロジェクトを新規作成する際に該当テンプレートを選択すると、カスタムルールが自動的にプロジェクトに追加されます。