

Reusable Components Methodology

Table of Contents

- 1 [Introduction](#)
- 2 [Principles](#)
- 3 [Leveraging Object Repository](#)
- 4 [Scaling up RPA by combining Workflow Libraries and Object Repository](#)
 - 4.1 [About Libraries](#)
 - 4.2 [Step by Step Method](#)
 - 4.3 [General Practices](#)
 - 4.3.1 [Arguments](#)
 - 4.3.2 [Error handling](#)
 - 4.3.3 [Structure and naming](#)
 - 4.4 [Approach for Larger Solutions](#)
 - 4.5 [Benefits](#)
- 5 [Examples](#)

Introduction

Any UiPath project, or for that matter any RPA solution, has two main elements: **A logic layer** which contains all the data validation and processing and any other logical instructions needed for the process flow and **a GUI interaction layer** which is used to extract and input data into the application. Usually, a tendency to intertwine these layers exist in the case of less experienced RPA developers, but this creates issues regarding the maintainability and the reusability of the components.

With the help of the **Libraries** feature, the separation of the GUI Interaction Layer can be done by creating a series of libraries with reusable components dedicated exclusively to the interaction with the GUI. This way, we can create a versatile and very robust library that is easy to maintain (since it only contains GUI interaction and no other process dependent logic) and extremely reusable (it is very likely that a different process that automates the same application(s) will reuse at least some of the components).

Furthermore the release of 20.10 brought the **Object Repository**, another feature which disrupts the old way Ui Automations were being built and brings a completely new way of creating and managing UI Selectors that make it a gamechanger when it comes to building scalable and highly reusable UI automations.

In this guide we'll try to define a methodology in which we can use these 2 features in order to build easily maintainable and reusable code, present all of the architectural principles and best practices behind it, and offer a detailed breakdown of the steps needed to write our reusable components accompanied by lots of examples.

Principles

There are a variety of general principles in software development that can be applied regardless of the programming language or the platform we are working on and that show a lot of potential when used in the context of RPA and of UiPath.

The most important principle we need to address is **separation on concerns**. This principle states that a program must be separated into different sections, so that each section addresses a separate concern. In our case, we need to layer our solution, meaning that each section should become a layer. Separating a solution on layers makes the layers be testable individually. For a normal software application, we usually have three distinct layers: the presentation layer, the domain layer and the persistence layer.

- the presentation layer is concerned with displaying that data and presenting it to the user. Thus, it's the layer concerned with the interaction with the user. The user interface module should not contain business logic.
- the domain layer is concerned with the domain logic, referring to the general business/problem domain that the application solves. In our case, this layer is concerned with the logic of the application.
- the persistence layer: the place where the data necessary for our application is stored.

However, in the context of an RPA project, our layers are:

- the domain layer aka the "**Logic Layer**"
- the **GUI Interaction Layer** (this basically replaces the presentation layer since the main objective of an automation is to interact with other application instead of with an actual user)
- the persistence layer: in an RPA project development, this is not a mandatory layer due to the fact that there are use cases in which we don't have permanent data storage. Given the scope of this document, we'll leave out this layer.

Another principle we need to take into consideration when designing and RPA project is **the single responsibility principle**. This means that a xaml file should do only one thing and one only. This principle is associated with **low coupling and high cohesion principle**. This means that the code should be modularized, even though it is layered. So, for a certain layer we can have multiple modules.

- High cohesion means that actions associated with the same application should be kept in the same module.

- Low coupling means that a module should depend on another module as little as possible. In a UiPath project we obtain modularization by using libraries (you can find more about this in the Method section).

Leveraging Object Repository

An RPA process can be simplified to a simple definition: a combination of Data, Actions and UI. Actions are UiPath activities, that manipulate data while interacting with different user interfaces.



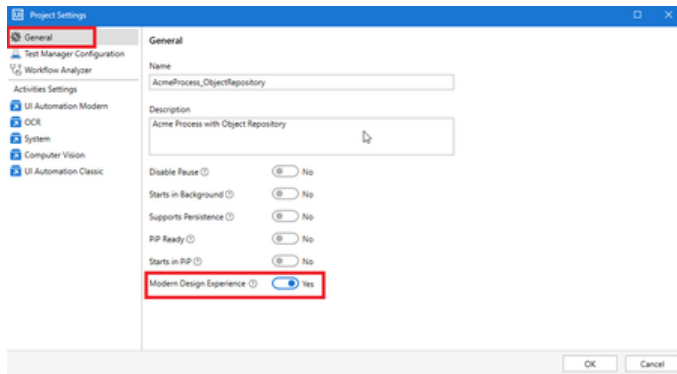
- **activities:** out-of-the-box activities, grouped by packages. UiPath also has the ability to build your own custom activities for something that does not exist OOB, by offering a framework on which you can build your custom activities:
<https://docs.uipath.com/activities/docs/creating-a-custom-activity>
- **data:** pre-made types when defining variables or arguments. UiPath also offers the ability to create your own types, which you can import and use in Studio. In 20.8, **Data Service** was introduced, so now you can have a centralized location where you can create data types and reuse them across automation.
- **UI:** you can reuse UI interaction by either using classic **Libraries** or by using the new **Object Repository**.

Object Repository is a new feature introduced in UiPath Studio starting with 20.4 Community edition and 20.10 Enterprise edition. By using Object Repository, RPA projects will have a better reusability in terms of UI interactions, it will be easier to capture not only classic selectors, but also descriptors and the UI structure of the application and to upgrade processes to a new version of the same application. Object Repository ensures the management, reusability and reliability of UI elements by capturing them in a DOM-like repository, sharable across projects. Thus it allows for creating and reusing UI taxonomies inside and across automation projects.

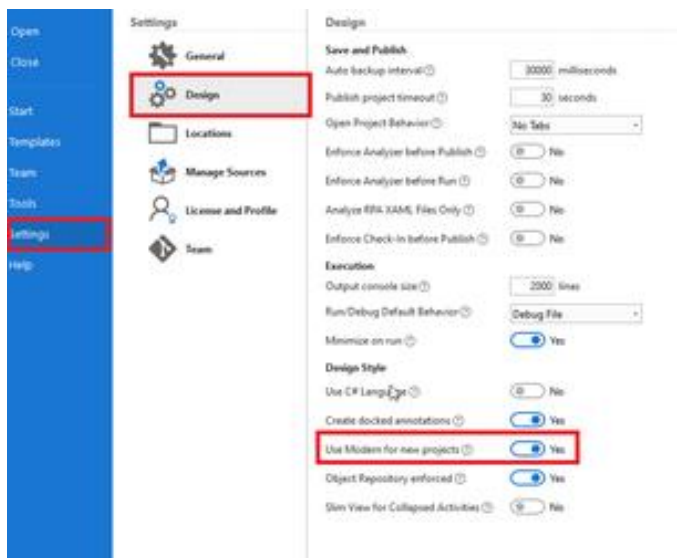
In order to use the Object Repository, you need to have projects that use the **Modern Design Experience** which work with **UiPath.UIAutomation.Activities** package version **2020.10** or higher.

Modern Design Experience can be enabled:

- on a project level, in the **Project** panel by going to project **Settings** → **General** → toggle **Modern Design Experience**



- on a Studio level, to make Modern Experience the default experience for all new projects, in **Home** → **Settings** → **Design** → toggle **Modern Design Experience**. When the **Object Repository enforced** is set to **Yes**, activities part of the UiAutomation pack need to reference elements from the Object Repository.

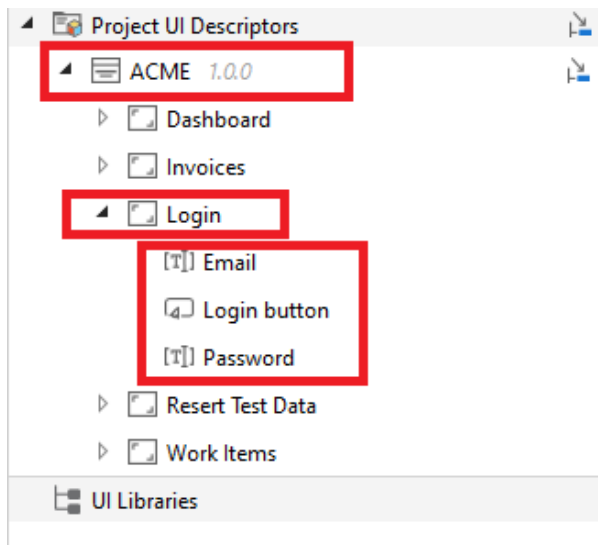


The Object Repository has a hierarchical structure where each node represents screens or elements, all under the application level. The structure is as follows:

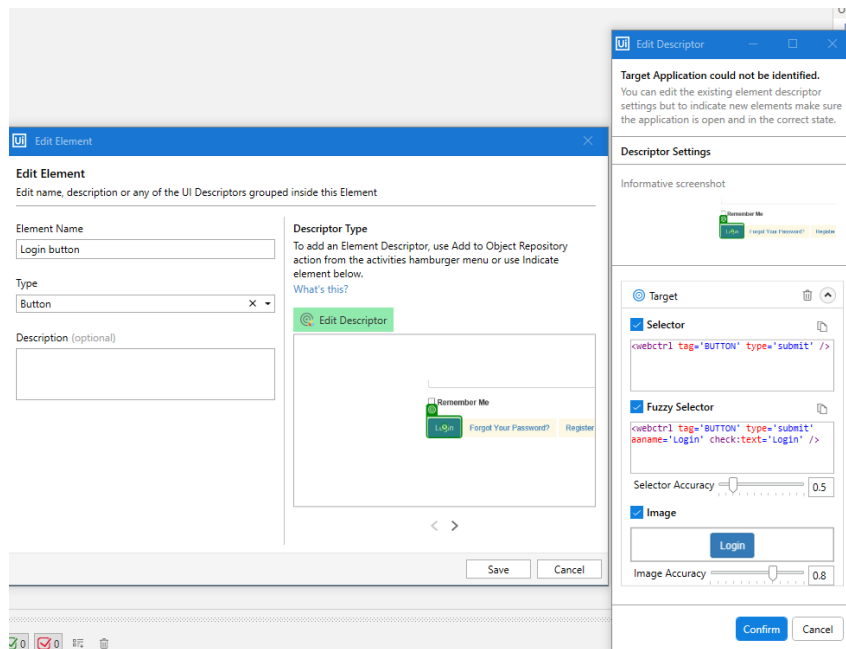
- **UI Application:** is the targeted application that can have multiple versions and each version can have multiple screens.
- **Screen:** is a UI Scope that groups together multiple elements belonging to the same screen. The UI Scope is either extracted from activities inside the workflow or are generated at element capture time.
- **UI Element:** contain full or partial selectors, anchor selectors, screen and element image capture context, and other metadata that describe the element on the screen.

- **UI Descriptors** is a superset of selectors which holds information for uniquely identifying elements on the screen. They are extracted from activities in the workflow and added to the structured schema which groups them by Application, Application Version, Screens and UI Elements. Out of the taxonomy structure, only Screens and Elements hold descriptor information → whereas the rest are used for grouping and their role is to ensure upgrades between versions of an application.

For example, in the below image we can identify:



- UI Application: ACME
- UI Application version: 1.0.0
- Screens: Dashboard, Invoices, Login...
- UI Element: Email, Login button...
- UI Description: double click *Login button*
 - UI Descriptors incorporate classic selectors, Fuzzy selectors or Image based automation. We do recommend using Image based automation only as the last resort, when everything else does not work.



In terms of reusability, **Object Repository** enables you to reuse your UI elements across projects

- at **project level** by using all locally stored elements
- for testing purposes, by using **Snippets** panel
- by extracting elements into **UI libraries** and installing them as a dependency into your projects when you want to reuse at a **global level**, You can also take the approach by creating UI Libraries with the elements you need across all of the automation projects.

Scaling up RPA by combining Workflow Libraries and Object Repository

About Libraries

By itself, the Object repository can significantly impact the ease and efficiency of building Ui automations. However, there still is a way in which we can further expand the potential of this feature and build workflows that are extremely easy to use and to maintain. Specifically, we're referring to incorporating our Object Repository Elements into a separate library project and further including in that library all of the Ui Interactions that need to be built, thus **keeping only the logic of the automation and no Ui interactions in the main UiPath Projects**.

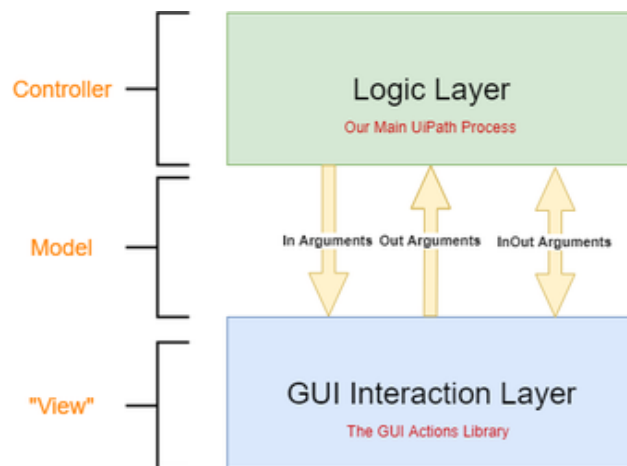
The recommended way of achieving this is to build such a library for each Application we're planning to automate and have that library contain all UI interactions and Object Repository Elements that belong to it. So a library (which is basically a way of packaging reusable components) will be created that will contain all the Object repository elements, descriptors and the workflows containing the UI interactions with the GUI layer. The library's

activity can be called from a xaml which is contained by the logic layer (our main project). One of the advantages with using this approach is that even though the UI interaction will change, this will not affect the logic of the application.

One could see this separation as a view-controller separation. Of course there should exist a model layer, which, in our case, is primarily concerned with the data passed from the controller to the view.

Another advantage is that once a library is published and a package is created, it can be referenced in different projects without the need to be rewritten every time or to copy past code between multiple projects. If a change occurs in the UI interaction, that library can be updated in a new version and all project updated to use the latest version. Also, if a change occurs in the logic layer, this will not affect the way the UI interaction is done. If an earlier version needs to be used again, it will be easily retrievable, because Orchestrator can store multiple versions of the same library. This way, different versions of the same libraries can be used in different projects and a single automation project can instantly switch between multiple versions of the same library.

Using this methodology, our main automation project (the controller) will contain all the process dependent logic and it will interact with the UI by leveraging the workflows inside our Library (which can be thought of as a pseudo-view). Finally, our model is represented by the arguments which help us pass data between these 2 components:

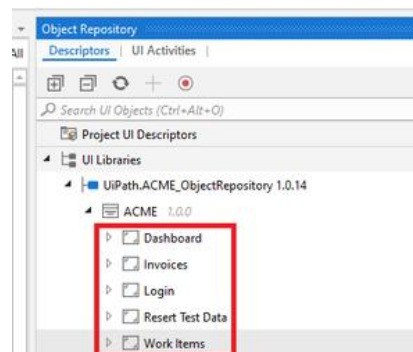


Step by Step Method

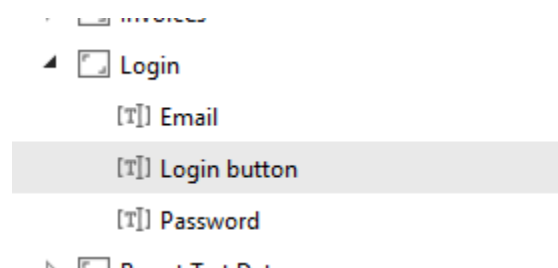
The following steps should be followed in order to create a Ui Activities Library as described above:

1. Analyze the current project and understand its steps: designing a flow diagram is recommended.
2. Assign each one of the steps to a different layer. For example, we could have a GUI layer category and a logic layer category.

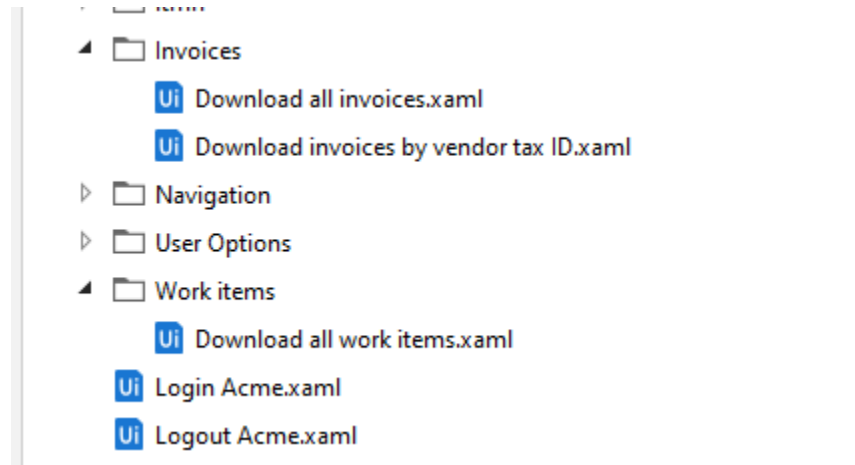
1. for example: click, type into, get text - could be considered as part of the GUI interaction layer. However, **we don't consider placing plain activities into the library - but placing more complex actions** (for example: click until something appears on the GUI or go through all pages of a table and extract all the rows). Of course, you can have complex actions, like logging in - which basically is formed by smaller GUI interaction parts. But the main point to take away is that **each component in a library should be an action** (e.g. a more complex piece of automation using multiple activities) **rather than a plain UI activity**.
2. **In the logic layer we should have actions which do not require UI interaction** like file processing, data validation and filtering, business exceptions handling, etc. Keep in mind that in order to get to the desired location where the logic layer can perform its steps, the GUI layer would also need to be called.
3. after the necessary interactions with the GUI have been identified, we should create a **separate library for each Application we're using in the automation**
4. for each Ui Library created, do the following (**the screenshots below are retrieved from the Library build as part of the example at the end of this file**):
 1. create the application object and the necessary screens in the Object Repository window



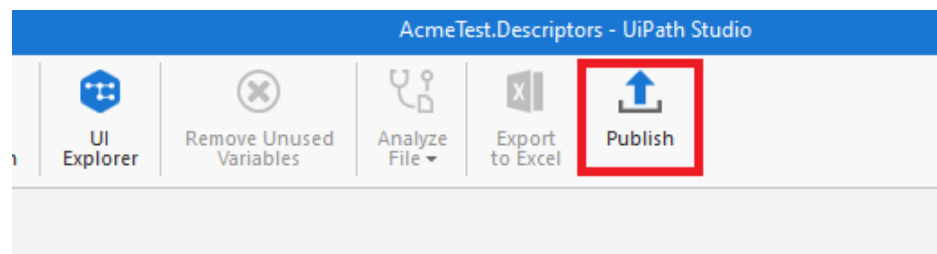
2. for each screen in the Object Repository, create the necessary Elements and build their descriptors



3. develop a xaml file for each necessary action (**this step can be done interchangeably with 4.a. and 4.b**, the Elements and Screens can added as new workflow files are being build)
 - For example, this are some of the components built to automate the Acme Test Processes (detailed explanation can be found at the end of this article)













4. publish the Library to the Orchestrator (or a local nuget feed)



5. Import the Library(ies) into your automation project (from the Manage Packages window) and use them in your automation project
 - after installing the libraries, they will appear as dependencies:

▶ `UiPath.ACME_ObjectRepository = 1.0.15`

6. the workflows inside the Libraries are visible in the Actions panel. From here, we can drag-n-drop them into our workflow and use them the same way we use activities:

- ▲ UiPath
 - ▲ ACME ObjectRepository
 - ▲ Invoices
 -  Download all invoices
 -  Download invoices by vendor tax ID
 - ▲ Navigation
 -  Navigate to home page
 -  Navigate to reset test data
 -  Navigate to search for invoice
 -  Navigate to work items
 - ▲ User Options
 -  Reset test data
 - ▲ Work items
 -  Download all work items
 -  Login Acme
 -  Logout Acme

Observation:

In case using Object Repository is not possible, then the process above can be followed exactly as described with the only exception being that for step 4, where only point **4.c** needs to be performed.

General Practices

Arguments

- Arguments should be named using the **PascalCase** standard.
 - this is the standard used for the parameters of all the activities inside the UiPath Studio so it makes sense to also use it for Library arguments in order to achieve a better level of standardization.
- the name of the arguments should not contain in_/out_/io_ prefix because those arguments will appear as properties when the library is being created.

Example:

This is how your arguments should look like:

The screenshot shows a workflow editor with two steps: 'Log Message' and 'Invoke Navigate to search for invoice workflow'. The 'Log Message' step has a 'Log Level' of 'Info' and a 'Message' of 'Start downloading all invoices by vendor t'. The 'Invoke Navigate to search for invoice workflow' step has a 'Workflow file name' of 'Navigation\Navigate to search for invoice.xaml' and buttons for 'Import Arguments' (with a count of 2) and 'Open Workflow'.

| Name | Direction | Argument type | Default value |
|-------------------|-----------|---------------|-----------------------------|
| Browser | In/Out | UiElement | Default value not supported |
| TimeoutDesired | In | Int32 | Enter a VB expression |
| VendorTaxID | In | String | Enter a VB expression |
| ExtractedInvoices | Out | DataTable | Default value not supported |

Create Argument

If you look at your published component, you can see that your arguments are already separated in different categories based on their type, so adding the prefix would be redundant.

The screenshot shows the 'Properties' window for a component named 'UiPath_ACME_ObjectRepository.Invoices.Download_invoices_by_vendor_tax_ID'. The properties are categorized into 'Common', 'Input', 'Input/Output', 'Misc', and 'Output'.

| Category | Property Name | Value |
|--------------|-------------------|--------------------------------|
| Common | DisplayName | Download invoices by vendor ta |
| Input | TimeoutDesired | in_TimeoutDesired |
| | VendorTaxID | in_VendorTaxID |
| Input/Output | Browser | io_Browser |
| Misc | Private | <input type="checkbox"/> |
| Output | ExtractedInvoices | ExtractedInvoices |

Check the Example section for the full version of this demo process.

- all workflows inside a GUI library which open/launch the application we automate (for example an activity that opens and performs the login into the application) should **return the resulting application Window** using either an out argument of type **UiPath.Core.Window** or **UiPath.Core.Browser** for **Classic Experience** projects (or **UiPath.Core.UIElement** if having **Modern Design Experience** enabled). Additionally, all other activities in the package should have an in argument of type **UiPath.Core.Window** or **UiPath.Core.Browser** which will indicate the correct window in which the current action should be performed in. This is extremely helpful if we have multiple instances of an application open during the runtime of our processes and it's a practice that needs to be followed for both the classic design experience and the modern one (for the classic design experience, **we should not send selectors as arguments!**)

Differences between classic and modern experience

If developing a library of automation components for the ACME site (<https://acme-test.uipath.com/>), I will obviously need a component which opens the browser and performs the Login Operation. This activity should take as arguments the username, password, site URL and return the **Browser** element (optionally, the Browser can be sent as an in/out argument if we want to support the situation in which the login should happen in an already existing window).

Classic Design Experience Example

The browser element is of type **UiPath.Core.Browser**, note that if we would be using a Desktop application instead of using a web application, then we would use an argument of type **UiPath.Core.Window**:

The screenshot shows the 'UiPath_ACME.Login' activity in the Classic Design Experience. The arguments are organized into three sections: Common, Input, and Input/Output. The 'Common' section has a 'DisplayName' argument with the value 'Login'. The 'Input' section has two arguments: 'Credential' with the value 'ACMECredits' and 'URL' with the value '"https://acme-test.u'. The 'Input/Output' section has one argument: 'Browser' with the value 'Browser'. Each argument has a dropdown menu to its right, indicated by three dots.

| UiPath_ACME.Login | |
|---------------------|----------------------|
| Common | |
| DisplayName | Login |
| Input | |
| Credential | ACMECredits |
| URL | "https://acme-test.u |
| Input/Output | |
| Browser | Browser |

This is how the arguments would look like:

| Name | Direction | Argument type |
|------------|-----------|---------------|
| Credential | In | Credentials |
| URL | In | String |
| Browser | In/Out | Browser |

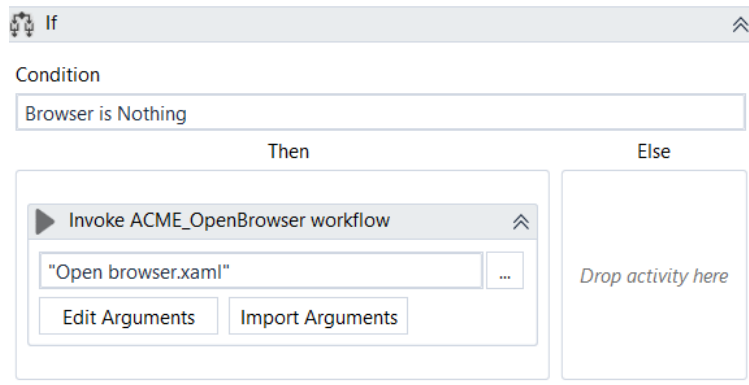
Modern Design Experience Example

The Browser element is of type **UiPath.Core.UiElement**:

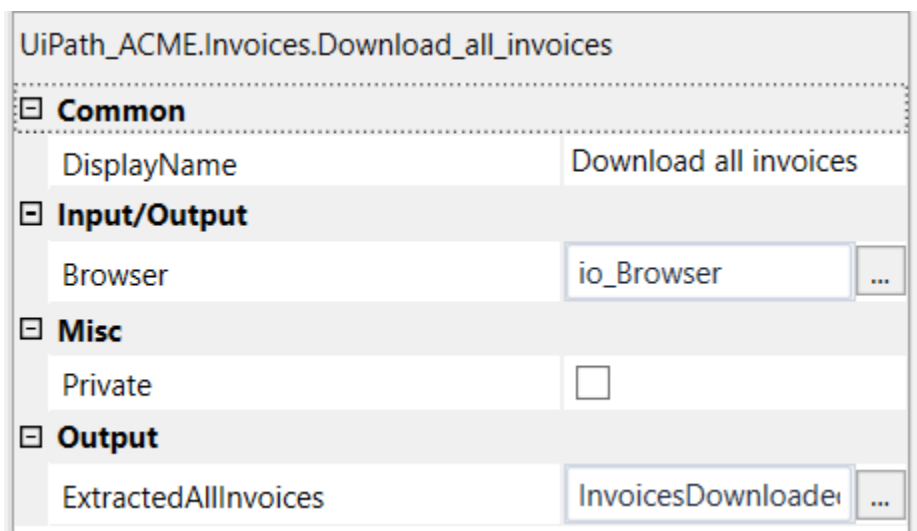
This is how the arguments would look like:

| Name | Direction | Argument type |
|----------|-----------|---------------|
| URL | In | String |
| Browser | In/Out | UiElement |
| Username | In | String |
| Password | In | SecureString |

When building the workflow that opens the Browser (e.g. the **Login** workflow), both in the case of Modern or Classic design, the first thing that needs to be done is check if the Browser argument is null and, in case this is true, to open a new browser at the <https://acme-test.uipath.com/> (the open browser action is encapsulated inside its own separate workflow):



After the login is performed and the **Browser** element is passed back to the process, using any other activities from this library will require this parameter. For example, the activity that downloads all invoices from the site will have the following parameters:



(without passing a valid browser to this activity, the execution would result in an error)

Check the Example section for the full version of this demo library.

Error handling

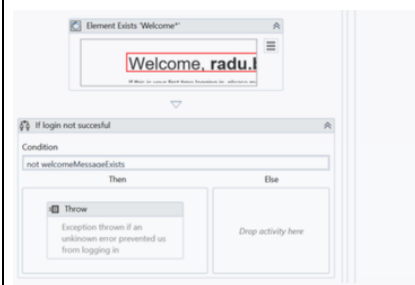
- Inside a library we should **throw errors when they appear**, not signal them through output arguments.
- At the end of a library component, it is recommended to validate its outcome. We should check if the desired action took place and throw an exception if it didn't.

| Classic Design Experience Example | Modern Design Experience Example |
|-----------------------------------|----------------------------------|
|-----------------------------------|----------------------------------|

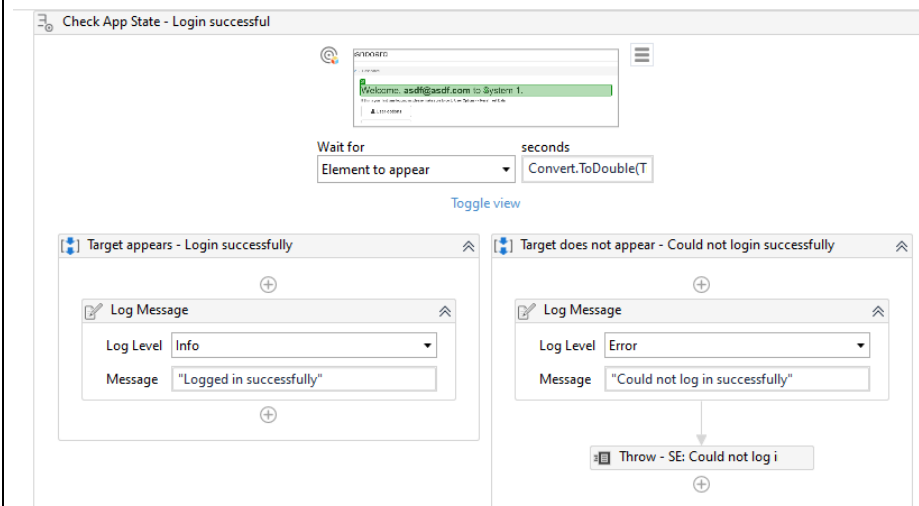
If I'm creating a Module that logs me into an application, I might want to throw an exception if the login does not happen correctly.

So, after I inserted the Credentials and clicked the Login button, I will add an Element Exist with a timeout of 10 seconds which checks if the "**Welcome...**" label exists. If this element exists returns false, it means that the authentication process was not successful and an exception is thrown:

For classic design, I might use an Element Exists activity to achieve this:



For modern design, a check App State Activity works best:



Structure and naming

Creating a component for each activity means that your library will get very large, very quickly, in order to compensate for the large number of components inside every library, it is essential to define a standard for naming your components, so that your developers are able to better search for them inside the activities panel. The standard we recommend for naming a component is the following: “**{Action} {Entity Used by Activity}**”. This standard has the following benefits:

- allows the developer to find a component by search for it either by its name, the action performed by this activity or by the entity used during this action
- makes it very easy to understand what the component can be used for and with what pages of the application it interacts with

In some cases we can just use: “**{Action}**” if the action is not performed on an entity (for example: **Login**) or, if there is a need to give more information through the name, add more attributes to the name, each separated by a space.

Additionally, it is recommended to create a folder structure inside the library project and grouping similar components in the same folder. A good rule of thumb would be to have one folder for each main entity that you interact with inside the GUI Layer. If there is a large number of entities you can interact with in an application, a multi-layered folder structure can be used.

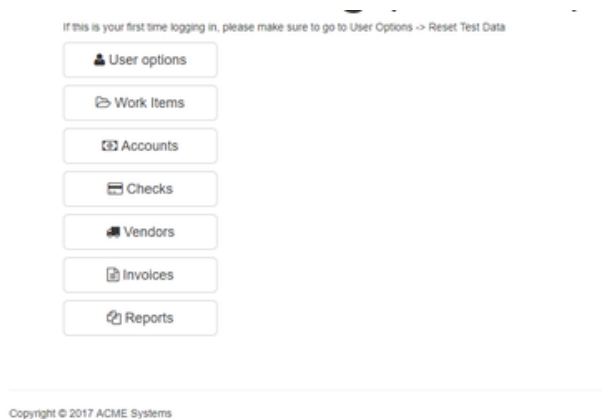
This greatly improves the cohesiveness of your library and makes it easier to see the possible actions that can be leveraged for each entity.

A good standard for naming folders is simply by using the name of the entity they act upon: “**{Entity Used by Activity}**” or the general activity they perform “**{Action}**” (for example, if we would want to create a folder where we store all components that navigate through the pages of a web application, we would call it **Navigation**) .

A naming convention needs to be applied to Object Repository entities like Screens or Elements too, preferably the same as in the case of Workflows and folders, so in our case it would be Pascal Case.

Example

The ACME site (<https://acme-test.uipath.com/>) contains several pages which allows its users to interact with the following entities: Work Items, Checks, Accounts, Vendors, Invoices, etc:



If I am currently trying to automate the ACME site and create a library with components that interact with the entities inside that site, I can choose to create the following folder structure:

- General
- ▲ Invoices
 - Ui Download all invoices.xml
 - Ui Download invoices by vendor tax ID.xml
- Navigation
- Tests
- User Options
- Work items
 - Ui Login Acme.xml
 - Ui Logout Acme.xml
 - project.json

Above you can also see the way the Library Workflows are named.

Observation:

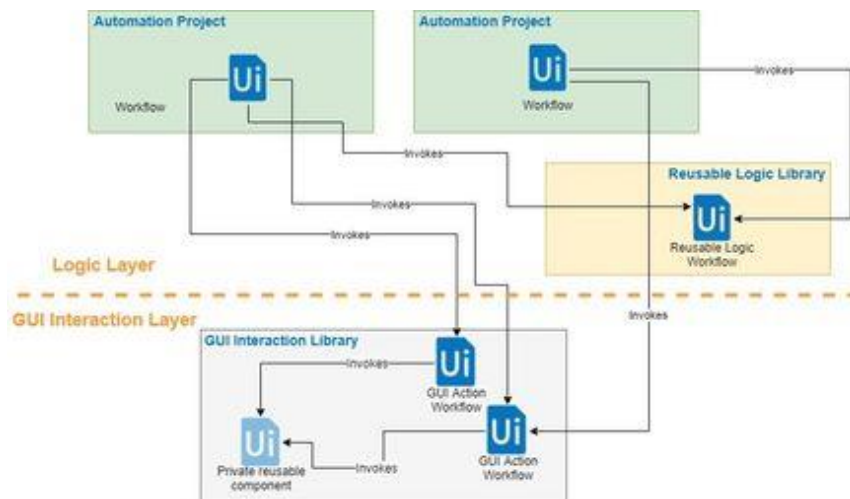
The name parts for each component/folder (**Application Name, Action, Entity Used by Activity**) should follow the **PascalCase** standard.

Approach for Larger Solutions

In larger automation projects it is possible that several components inside a Library have a very similar or even identical piece of code. In this case, we should isolate this component in a separate workflow. In the case we do not want this workflow to be accessible outside the library, we should mark it as private.

Additionally, it is possible that the same logic needs to be implemented between multiple processes. In this case, the reusable piece of logic should be isolated itself inside a Library.

The following diagram gives us an overview of how a solution like this might look like:



Benefits

- one could build only one layer without interfering with the other one.
- reusing already existing code, without the need for it to be rewritten each time when it is used.
- this methodology makes it easier to change the UI Interaction when an update in the application occurs and to push the change to all of the processes using the library without having to republish them.
- sharing the UI Interaction Layer between multiple processes means that the same module is tested more thoroughly and that any fixes can for all processes at once, resulting in very robust and reliable automations.

- having a dedicated GUI Interaction library makes it **very easy to build dedicated Test Automation Projects** for testing the Ui automation for a specific application, thus ensuring that our code is very easy to test and that breaking changes to the application are detected immediately.

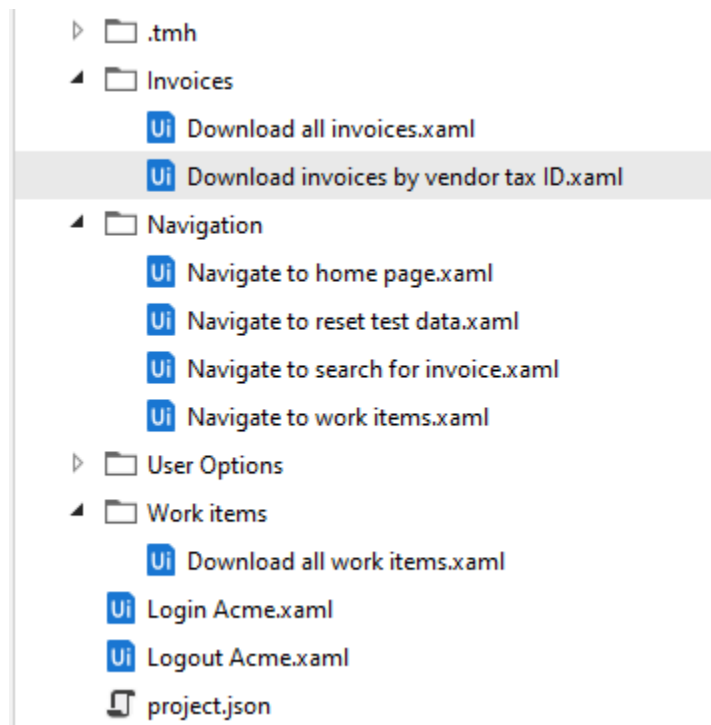
Examples

To exemplify the creation and usage of a library using Object Repository, we created a use case with the ACME web application.

a) The library containing all the Acme automation activities is called **Acme_ObjectRepository**. You can find the latest version below:



The library, even though it is a module focused on the interaction with the ACME web application, is also modularized and organized into multiple folders, based on their purpose:



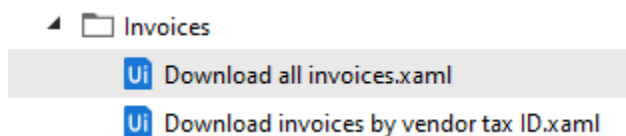
Here's a quick breakdown of our library components:

- **Login Acme.xaml**

- this file's main purpose of this file is to log in inside the application and it has 4 arguments, as seen below

| Name | Direction | Argument type | Default value |
|-----------------|-----------|---------------|-----------------------------|
| URL | In | String | Enter a VB expression |
| Browser | In/Out | UiElement | Default value not supported |
| Username | In | String | Enter a VB expression |
| Password | In | SecureString | Enter a VB expression |
| TimeoutDesired | In | Int32 | Enter a VB expression |
| Create Argument | | | |

- The "Browser" argument is of type **UiPath.Core.UiElement** and it's an In/Out argument. This component will attach to any Acme windows that are already opened or if none exists, open a new browser window itself and perform the login. The browser parameter will return the browser window and this variable will be passed on to all other components from this library.
- **Logout Acme.xaml**
 - it invokes **Navigate to home page** workflow from the Navigation module and subsequently it terminates the current session.
- The **UserOptions** module consists of the options that can change the current account's configurations inside the ACME web app.
 - Reset test data.xaml
 - Reset the test data used in the ACME application
- The **Invoices** module is focused on the actions which take place on the invoices found in the test data.



- Download all invoices
 - This workflow extracts all the items in the invoice table inside the application and returns them as a DataTable.
- Download invoices by vendor tax ID
 - This workflow navigates through the entire invoice table until it finds an invoice with a specific tax ID, which it then returns.
- The **Navigation** module consists of workflows that simply navigate through the different pages of the web application

- Navigation
 - Ui Navigate to home page.xaml
 - Ui Navigate to reset test data.xaml
 - Ui Navigate to search for invoice.xaml
 - Ui Navigate to work items.xaml

- The components in this category are mostly used by the other library workflows and normally they should not be invoked directly from our main process, but they are still marked as Publishable because they might be useful in Attended processes where we might want our robot to navigate to a specific page and then allow the user to interact with the application.
 - The **WorkItems** module is specialized on the actions which take place on the work items found in the application

- Work items
 - Ui Download all work items.xaml

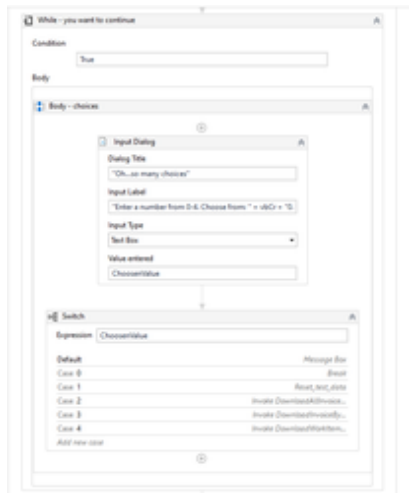
- Download all work items
 - This permits the downloading of all work items in the test data

b) The main project, **AcmeProcess_ObjectRepository** consists of 4 different use-cases. In order for it to work, the user must create a Window credential of Generic type called "**ACMETest**", which saves the username and the password with which the robot will log into the ACME web application.



All of the 4 use-cases use components in this process perform various tasks that make use of the multiple components built in the ACME Ui Components Library.

When the **Main.xaml** workflow is invoked, the robot will login into the ACME application and then through a dialog box, it will ask the user to select one of the use-cases then invoke the workflow corresponding to it:



c) The testing project, **Test_UiPath.AcmeProcess_ObjectRepository** consists of 5 different test-cases, where we test different functionalities that the process has: login/logout, download all invoices / download invoices by vendor tax id, download work items.

