> **Note:** Some images in this document are replaced with placeholders as they are only available in the GitHub UI.

Introduction

Workshop 3 serves to introduce concepts of the UTPClient library by exploring common UTP modules such as Buttons and Game Rules. By the end of this workshop, you'll have created a test that uses UTP modules to enter Game Rules and take screenshots of all pages. Throughout the workshop, we will create small tests that work towards this goal, and then bring them all together at the end. Please refer to the following resources from Workshop 3 and follow the 5 Parts of the workshop as documented below.

Workshop 3 Resources:

Workshop 3 PowerPoint

Part 0: Updating Packages

Before getting started with UTP modules, let's go over how to update packages. Updating packages ensures you have the latest features, bug fixes, and improvements. Follow the steps on updating packages before proceeding to Part 1.

Part 1: Get UTP Modules

This series of steps serves to introduce the concept of viewing all available UTP modules available in the game.

1. Just like in Part 1 of Workshop 2, let's start by creating a new class. Name this new file/class **UtpTests.cs**.

2. Add a using statement to include **BugSweeper.Tests**. Then add the **[TestFixture]** attribute before defining the class. Let this class **inherit from EgmTestBase**. Your class should look like this:

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{

}
```

3. Let's add a test to this class called **GetUtpModules()**. Mark it with the **[Test]** attribute above the method. This test will collect all the available UTP modules available in the game and then print them out. We'll make this test asynchronous (async) to allow it to run without making the program wait.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {

    }
}
```

4. The first thing we want to have this test do is get the list of available UTP modules from the current game. Create a new variable called **modules**, then set it equal to the list of available modules. We can use **Utp.GetClientModules**() to do this. You'll see we're passing in a Cancellation Token. This just allows the operation to be cancelled if needed. We also need to await this call

because our test is async.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
    }
}
```

5. Let's add an asset to check that we properly got the list of modules. We want to check (or "assert") that the **list is not empty**.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);
    }
}
```

6. Now that we got the list of available modules, we want to print the whole list out. let's create a loop to go through each module in the list.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);

        foreach(var module in modules)
        {

        }
    }
}
```

7. Inside that loop, print out the name of each module. You can use **Console.Writeline**() to print to your test output.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);

        foreach(var module in modules)
        {
            Console.WriteLine($"Module: {module.Name}");
        }
    }
}
```

8. Next, let's print each command in each of the modules. Inside the loop you just created, create another loop to go through all of the

commands within a module.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);

        foreach(var module in modules)
        {
            Console.WriteLine($"Module: {module.Name}");

            foreach(var command in module.Commands)
            {

            }
        }
    }
}
```

9. Inside this loop, print out the module's commands.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);

        foreach(var module in modules)
        {
            Console.WriteLine($"Module: {module.Name}");

            foreach(var command in module.Commands)
            {
                Console.WriteLine($"  Command: {command.Command}");
            }
        }
    }
}
```

10. Now, let's print any parameters that the commands might require. Create one more loop inside the last loop. This one will go through all of the parameters in the commands, if applicable.

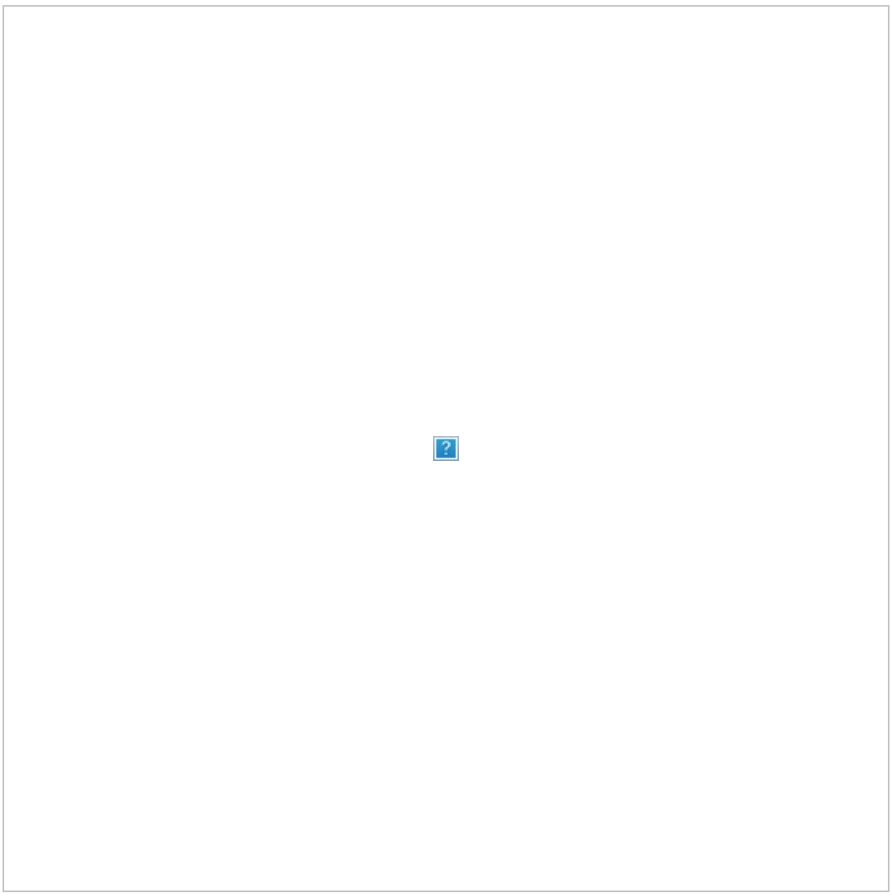
```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);

        foreach(var module in modules)
        {
            Console.WriteLine($"Module: {module.Name}");

            foreach(var command in module.Commands)
            {
                Console.WriteLine($"  Command: {command.Command}");

                foreach(var parameter in command.Parameters)
                {
```

```
                    }
                }
            }
        }
    }
```

11. Inside that loop, print out the name and type for each of the command parameters.

```
using BugSweeper.Tests;

[TestFixture]
internal class UtpTests : EgmTestBase
{
    [Test]
    public async Task GetUtpModules()
    {
        var modules = await Utp.GetClientModules(CancellationToken);
        Assert.That(modules, Is.Not.Null.Or.Empty);

        foreach(var module in modules)
        {
            Console.WriteLine($"Module: {module.Name}");

            foreach(var command in module.Commands)
            {
                Console.WriteLine($"  Command: {command.Command}");

                foreach(var parameter in command.Parameters)
                {
                    Console.WriteLine($"    Parameter:
{parameter.Name}");

                    Console.WriteLine($"      Type:
{parameter.Type}");
                }
            }
        }
    }
}
```

12. Your test is complete. Run this test. You should see an output with all of the available UTP modules for the game, as well as each module command and the parameters that they require:



Part 2: Get All Buttons

This series of steps serve to introduce using UTP Buttons modules.

1. For this next part, we'll need to add another using statement to allow us to use the **UTP framework**.

```
using BugSweeper.Tests;
using IGT.Utp.Framework;
```

2. Create a new test in your UtpTests class. Name this test **GetAllButtons()** and mark it with the **[Test]** attribute. This test will be **async** just like the last one. The purpose of this test is to get a list of all of the buttons available in a game.

```
[Test]
public async Task GetAllButtons()
{

}
```

3. The first thing we want this test to do is to retrieve the list of all buttons by their IDs. Create a new variable called **cmd** and set it equal to a new automation command. We'll pass in the **"Buttons"** module and the **"GetButtonMapping"** command. The "Buttons" module and "GetButtonMapping" command were found from the results we got in Part 1, as that tells us what UTP modules are available for the current loaded game.

*__Note__: If you are using a Poker game, this may need to be written slightly differently. Please refer to the code in the Workshop 3 final content before proceeding or switch to using a recommended non-Poker game for the workshop.

```
[Test]
public async Task GetAllButtons()
{
    var cmd = new AutomationCommand(
        "Buttons",
        "GetButtonMapping");
}
```

4. Next we want to send the command to the UTP server and retrieve the response. Create a new variable called **result** and set it equal to the **SendCommandGetResponse**() method, passing in the cmd we just retrieved and the CancellationToken. Once again, we will need to **await** this because our method is async.

```
[Test]
public async Task GetAllButtons()
{
    var cmd = new AutomationCommand(
        "Buttons",
        "GetButtonMapping");

    var result = await Utp.SendCommandGetResponse(
        cmd,
        CancellationToken);
}
```

5. After we get the result from the command, we want to print each button and their relevant IDs. Create a loop that goes through each button in the result.

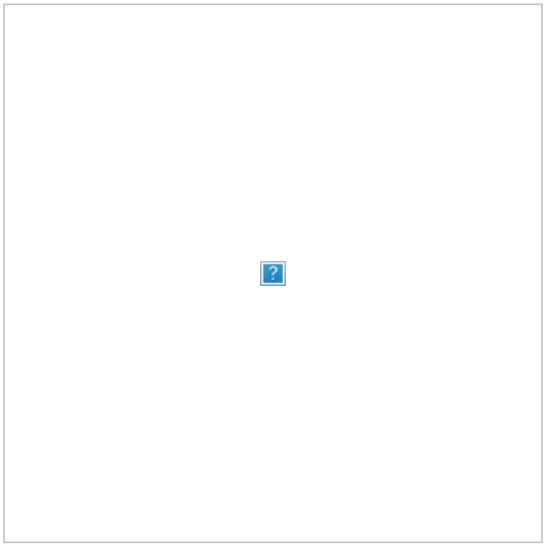
```
[Test]
public async Task GetAllButtons()
{
    var cmd = new AutomationCommand(
        "Buttons",
        "GetButtonMapping");

    var result = await Utp.SendCommandGetResponse(
        cmd,
        CancellationToken);
```

```
        foreach(var button in result.Parameters)
        {

        }
    }
```

6. Inside the loop, print the button.

```
[Test]
public async Task GetAllButtons()
{
    var cmd = new AutomationCommand(
        "Buttons",
        "GetButtonMapping");

    var result = await Utp.SendCommandGetResponse(
        cmd,
        CancellationToken);

    foreach(var button in result.Parameters)
    {
        Console.WriteLine(button);
    }
}
```

7. Your test is complete! Run this test. You should see an output with all of the available buttons for the game.



Part 3: Press A Button

This series of steps will introduce using the UTP Buttons module to press an in-game button.

1. Let's make a test that presses the Speed button in our game. Create a new test Called **PressAButtonTest**(). Mark it with the **[Test]** attribute. This test will also be **async**.

```
[Test]
public async Task PressAButtonTest()
{

}
```

2. The first thing we want this test to do is retrieve the list of all buttons by their IDs. create a new variable named cmd and set it equal to a new Automation Command, passing in the **"Buttons"** UTP module, the command **"EmulateButtonPressByName"**, the **"ButtonName"** parameter, and the name of the button: **"Speed"**. "EmulateButtonPressByName" was another command in the "Buttons" module that we found in the results of Part 1.

```
[Test]
public async Task PressAButtonTest()
{
    var cmd = new AutomationCommand(
```

```
            "Buttons",
            "EmulateButtonPressByName",
            new AutomationParameter(
                "ButtonName",
                "Speed"));
    }
```

3. Next, we want to send the command to the UTP server and retrieve the response. Use **SendCommandGetResponse**() like in Part 1. The test is async, so we need to **await** this.

```
    [Test]
    public async Task PressAButtonTest()
    {
        var cmd = new AutomationCommand(
            "Buttons",
            "EmulateButtonPressByName",
            new AutomationParameter(
                "ButtonName",
                "Speed"));

        await Utp.SendCommandGetResponse(
            cmd,
            CancellationToken);
    }
```

4. This test is complete! Run the test. You should see it press the Speed button in your game.

Part 4: Enter Game Rules

This series of steps will introduce using UTP modules to enter the Game Rules.

Create a Helper Method to Check if Game Rules are Visible

1. Let's make a test that uses UTP modules to enter the Game Rules. Before we create our test, let's make a helper method that we can call from our test. This method will check if the Game Rules are already open. Name this method **IsGameRulesVisibleAsync**(). Make the method **async**. This method will also return a **bool** (true or false) result. Note: This method is not a test, but will be used to help write tests. By creating this method, we avoid having to rewrite the same steps in multiple places.

```
    private async Task<bool> IsGameRulesVisibleAsync()
    {

    }
```

2. We want this method to check if the game rules are visible. Create a new variable named **visibleCmd** and set it equal to a new Automation Command, passing in the **"GameRules"** module and the **"IsVisible"** command. The "GameRules" module and "IsVisible" command were found in the results of Part 1.

```
    private async Task<bool> IsGameRulesVisibleAsync()
    {
        var visibleCmd = new AutomationCommand(
            "GameRules",
            "IsVisible");
    }
```

3. Now we want to send this command to the UTP server and retrieve the response. Create a new variable called **visibleResult** and set it equal to **SendcommandGetResponse**() to do this, passing in our visibleCmd and the CancellationToken.

```
    private async Task<bool> IsGameRulesVisibleAsync()
    {
        var visibleCmd = new AutomationCommand(
            "GameRules",
```

```
            "IsVisible");

        var visibleResult = await Utp.SendCommandGetResponse(
            visibleCmd,
            CancellationToken);
    }
```

4. We want this method to **return the result** of whether or not the Game Rules are visible. Return the bool value of the result we just retrieved.

```
    private async Task<bool> IsGameRulesVisibleAsync()
    {
        var visibleCmd = new AutomationCommand(
            "GameRules",
            "IsVisible");

        var visibleResult = await Utp.SendCommandGetResponse(
            visibleCmd,
            CancellationToken);

        return bool.Parse(visibleResult.Parameters[0].Value);
    }
```

Create a Helper Method to Enter Game Rules

5. We'll create one other helper method for the test we want to write. This method will use the Game Rules UTP module to enter the Game Rules. Name this method **EnterGameRulesAsync**(). Make the method **async**.

```
    private async Task EnterGameRulesAsync()
    {

    }
```

6. First, we want the method to open the game rules. Create a new variable named **cmd** and set it equal to a new Automation Command with the **"GameRules"** module and **"Open"** command. "Open" was another command in the "GameRules" module that we found in the results of Part 1.

```
    private async Task EnterGameRulesAsync()
    {
        var cmd = new AutomationCommand(
            "GameRules",
            "Open");
    }
```

7. Now we want to send the command to the UTP server and retrieve the response.

```
    private async Task EnterGameRulesAsync()
    {
        var cmd = new AutomationCommand(
            "GameRules",
            "Open");

        await Utp.SendCommandGetResponse(
            cmd,
            CancellationToken);
    }
```

Create a Test Using the Helper Methods to Enter the Game Rules

8. Now our two helper methods are complete and we can write our test. Create a new test and name it **EnterGameRulesTest**(). Mark it with the **[Test]** attribute. Make this test **async**. This test will check if the Game Rules are already open, enter the Game Rules, and then verify that we have successfully entered the Game Rules.

```
    [Test]
    public async Task EnterGameRulesTest()
```

```
{

}
```

9. First, we want this test to call the **IsGameRulesVisibleAsync**() and check (assert) **that the result is False**, meaning that the Game Rules are not yet open. Remember we need to **await** this because we are working with async methods.

```
[Test]
public async Task EnterGameRulesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");
}
```

10. Then we'll have this test call the **EnterGameRulesAsync**() method to open the Game Rules. Remember to **await** this call.

```
[Test]
public async Task EnterGameRulesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");

    await EnterGameRulesAsync();
}
```

11. Next, we want to check (assert) **that the game rules are visible** after opening them. We can use **IsGameRulesVisibleAsync**() again in this Assert.

```
[Test]
public async Task EnterGameRulesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");

    await EnterGameRulesAsync();

    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.True,
        "Game Rules is not visible after entering.");
}
```

12. This test is complete. Run the test. You should see it open the Game Rules in your game.

Part 5: Screenshot Game Rules

This series of steps will apply the content from previous steps and use them together in a test.

Create a Helper Method to Take Screenshots of Game Rules

1. Now we want to make a test which uses these UTP modules to enter Game Rules, take screenshots of each page, and exit Game Rules. First, we'll want to make a couple more helper methods. Create a new method named **ScreenshotGameRulesAsync**(). This method will cycle through all of the Game Rules pages and take a screenshot of each page. Make this method **async**.

```
private async Task ScreenshotGameRulesAsync()
{
```

```
}
```

2. In this method, we only want to take screenshots while more pages exist, and we want one screenshot per page. So, we'll create a couple variables before going through the pages. Create a variable named **morePagesExist** and set it equal to true. We will only take screenshots while this variable stays equal to true. Create another variable, name it **pageIndex**, and set it equal to zero. We will use this variable to keep track of which page we are screenshotting.

```
private async Task ScreenshotGameRulesAsync()
{
    var pageIndex = 0;
    var morePagesExist = true;
}
```

3. As we mentioned, we only want to take screenshots **while morePagesExist is true**. Create a while loop using this variable.

```
private async Task ScreenshotGameRulesAsync()
{
    var pageIndex = 0;
    var morePagesExist = true;

    while(morePagesExist)
    {

    }
}
```

4. We'll take our screenshots **inside** this while loop. First we'll have the method wait 1 second between pages using **Task.Delay**(). Remember to **await** this because the method is async.

```
private async Task ScreenshotGameRulesAsync()
{
    var pageIndex = 0;
    var morePagesExist = true;

    while(morePagesExist)
    {
        await Task.Delay(1000);
    }
}
```

5. Now we'll use the Game Rules module to activate each page. Create a variable called **cmd** and set it equal to a new Automation Command with the **"GameRules"** module, the **"ActivatePage"** command, the **"Index"** parameter, and the **pageIndex** variable you created. "ActivatePage" was another command in the "GameRules" module that we found in the results of Part 1.

```
private async Task ScreenshotGameRulesAsync()
{
    var pageIndex = 0;
    var morePagesExist = true;

    while(morePagesExist)
    {
        await Task.Delay(1000);

        var cmd = new AutomationCommand(
            "GameRules",
            "ActivatePage",
            new AutomationParameter("Index", pageIndex));
    }
}
```

6. Next we'll send the command to the UTP server and retrieve the response. Create a variable named **result** and set it equal to **SendCommandGetResponse**(), passing in the cmd variable and the CancellationToken.

```
private async Task ScreenshotGameRulesAsync()
{
    var pageIndex = 0;
```

```
            var morePagesExist = true;

            while(morePagesExist)
            {
                await Task.Delay(1000);

                var cmd = new AutomationCommand(
                    "GameRules",
                    "ActivatePage",
                    new AutomationParameter("Index", pageIndex));

                var result = await Utp.SendCommandGetResponse(
                    cmd,
                    CancellationToken);
            }
        }
```

7. The result we just retrieved gives us a True or False for whether or not we can activate the page at the index we passed in. Let's update the ableToActivatePage variable with this result. Set **ableToActivatePage** equal to the **bool value of this result**.

```
        private async Task ScreenshotGameRulesAsync()
        {
            var pageIndex = 0;
            var morePagesExist = true;

            while(morePagesExist)
            {
                await Task.Delay(1000);

                var cmd = new AutomationCommand(
                    "GameRules",
                    "ActivatePage",
                    new AutomationParameter("Index", pageIndex));

                var result = await Utp.SendCommandGetResponse(
                    cmd,
                    CancellationToken);

                morePagesExist =
bool.Parse(response.Parameters[0].Value);
            }
        }
```

8. Now that **morePagesExist** is updated, let's check to make sure it's still true. **If morePagesExist is still true**, we will continue on to take the screenshot of the page.

```
        private async Task ScreenshotGameRulesAsync()
        {
            var pageIndex = 0;
            var morePagesExist = true;

            while(morePagesExist)
            {
                await Task.Delay(1000);

                var cmd = new AutomationCommand(
                    "GameRules",
                    "ActivatePage",
                    new AutomationParameter("Index", pageIndex));

                var response = await Utp.SendCommandGetResponse(
                    cmd,
                    CancellationToken);

                morePagesExist =
bool.Parse(response.Parameters[0].Value);

                if(morePagesExist)
                {

                }
            }
```

```
}
```

9. Create a new variable called **screenshotPath** and set it equal to the file path and file name where you want to save the screenshot. or this example, we'll make this path *C:/Screenshots/**GameName**/**PaytableName**/GameRules_**PageIndex**.png*, where GameName is the name of your current game, PaytableName is the name of your active paytable, and PageIndex is the index of the current Game Rules page. (e.g. *C:/Screenshots/MagicFortune/AVV061492/GameRules_1.png*). We can use GameUtilities.TakeScreenshot() just like in Part 3 of Workshop 2. We need to pass in the Egm parameter and the screenshotPath.

```
        private async Task ScreenshotGameRulesAsync()
        {
            var pageIndex = 0;
            var morePagesExist = true;

            while(morePagesExist)
            {
                await Task.Delay(1000);

                var cmd = new AutomationCommand(
                    "GameRules",
                    "ActivatePage",
                    new AutomationParameter("Index", pageIndex));

                var response = await Utp.SendCommandGetResponse(
                    cmd,
                    CancellationToken);

                morePagesExist =
bool.Parse(response.Parameters[0].Value);

                if(morePagesExist)
                {
                    var screenshotPath = "C://Screenshots//" +
                                         Egm.ActiveGame.Name + "_" +

Egm.ActiveGame.PaytableInfo.Paytable + "_" +
                                         "GameRules_" + (pageIndex +
1) + ".png";

                    GameUtilities.TakeScreenshot(Egm, screenshotPath);
                }
            }
        }
```

10. After we take the screenshot, we'll want to increase the **pageIndex**

```
        private async Task ScreenshotGameRulesAsync()
        {
            var pageIndex = 0;
            var morePagesExist = true;

            while(morePagesExist)
            {
                await Task.Delay(1000);

                var cmd = new AutomationCommand(
                    "GameRules",
                    "ActivatePage",
                    new AutomationParameter("Index", pageIndex));

                var response = await Utp.SendCommandGetResponse(
                    cmd,
                    CancellationToken);

                morePagesExist =
bool.Parse(response.Parameters[0].Value);

                if(morePagesExist)
                {
```

```
                    var screenshotPath = "C://Screenshots//" +
                                         Egm.ActiveGame.Name + "_" +
Egm.ActiveGame.PaytableInfo.Paytable + "_" +
                                         "GameRules_" + (pageIndex +
1) + ".png";

                    GameUtilities.TakeScreenshot(Egm, screenshotPath);

                    pageIndex++;
                }
            }
        }
```

Create a Helper Method to Exit Game Rules

11. There's one more helper method we want to make before creating our test. We'll make a helper method to exit the Game Rules, much like the one we made to enter Game Rules. Create an **async** method and name it **ExitGameRulesAsync**().

```
private async Task ExitGameRulesAsync()
{

}
```

12. Create a new variable named **cmd**. and set it equal to a new Automation Command, passing in the **"GameRules"** module and the **"Close"** command. "Close" was another command in the "GameRules" module that we found in the results of Part 1.

```
private async Task ExitGameRulesAsync()
{
    var cmd = new AutomationCommand(
        "GameRules",
        "Close");
}
```

13. Next, send the command to the UTP server and retrieve the response. Use **SendCommandGetResponse**() and pass in the cmd variable and the CancellationToken. **Await** this call because we are working with async methods.

```
private async Task ExitGameRulesAsync()
{
    var cmd = new AutomationCommand(
        "GameRules",
        "Close");

    await Utp.SendCommandGetResponse(
        cmd,
        CancellationToken);
}
```

Create a Test Using the Helper Methods to Enter Game Rules, Take Screenshots, and Exit Game Rules

14. Now we're ready to put these helper methods together to write a test which enters the Game Rules, takes a screenshot of each page, then exits the Game Rules. Create a new **async** test named **EnterGameRulesAndScreenshotAllPagesTest**() and mark it with the **[Test]** attribute.

```
[Test]
public async Task EnterGameRulesAndScreenshotAllPagesTest()
{

}
```

15. We want this test to check (assert) that the Game Rules **are not already visible** before trying to open them. We can do this with the **IsGameRulesVisibleAsync**() helper method. **Await** the call because we are working with async methods.

```
private async Task ScreenshotGameRulesAsync()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");
}
```

16. Then we can enter the Game Rules using the **EnterGameRulesAsync**() method, just like we did in Part 4. **Await** this call because we are working with async methods.

```
[Test]
public async Task EnterGameRulesAndScreenshotAllPagesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");

    await EnterGameRulesAsync();
}
```

17. Next, we want to check (assert) that the game rules are visible after opening them. We can use **IsGameRulesVisibleAsync**() again in this Assert. **Await** the call because we are working with async methods.

```
[Test]
public async Task EnterGameRulesAndScreenshotAllPagesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");

    await EnterGameRulesAsync();

    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.True,
        "Game Rules is not visible after entering.");
}
```

18. Once we are in the Game Rules, we'll use the **ScreenshotGameRulesAsync**() method to take our screenshots. **Await** this call because we are working with async methods.

```
[Test]
public async Task EnterGameRulesAndScreenshotAllPagesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.False,
        "Game Rules is already visible.");

    await EnterGameRulesAsync();

    Assert.That(
        await IsGameRulesVisibleAsync(),
        Is.True,
        "Game Rules is not visible after entering.");

    await ScreenshotGameRulesAsync();
}
```

19. After taking our screenshots, we'll have the test exit the Game Rules using **ExitGameRulesAsync**(). **Await** this call because we are working with async methods.

```
[Test]
public async Task EnterGameRulesAndScreenshotAllPagesTest()
{
    Assert.That(
        await IsGameRulesVisibleAsync(),
```

```
            Is.False,
            "Game Rules is already visible.");

        await EnterGameRulesAsync();

        Assert.That(
            await IsGameRulesVisibleAsync(),
            Is.True,
            "Game Rules is not visible after entering.");

        await ScreenshotGameRulesAsync();

        await ExitGameRulesAsync();
    }
```

20. The last thing we want this test to do is check that we successfully exited the Game Rules. Use **IsGameRulesVisibleAsync**() to check (assert) that the game rules are **not visible**.

```
    [Test]
    public async Task EnterGameRulesAndScreenshotAllPagesTest()
    {
        Assert.That(
            await IsGameRulesVisibleAsync(),
            Is.False,
            "Game Rules is already visible.");

        await EnterGameRulesAsync();

        Assert.That(
            await IsGameRulesVisibleAsync(),
            Is.True,
            "Game Rules is not visible after entering.");

        await ScreenshotGameRulesAsync();

        await ExitGameRulesAsync();

        Assert.That(
            await IsGameRulesVisibleAsync(),
            Is.False,
            "Game Rules is still visible after exiting.");
    }
```

21. This test is complete! Run the test. You should see it open the Game Rules, take a screenshot of each page, and close the Game Rules. Check your *C:/Screenshots* folder for the screenshots. Congratulations! You've completed all of the steps for Workshop 3.