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

Recording	Slides
Workshop 1 Recording - SharePoint	Series1-Workshop-Intro-Setup.pptx

Overview

1. Introduction to **BugSweeper** tool
2. Creating, cloning a **GitHub** repository
3. Building, running first **NUnit** tests
4. Debugging **NUnit** tests
5. Using **GitHub Copilot** AI to write tests

Prerequisites

Prerequisites in order to run BugSweeper tool are found here . We **need** these completed before moving forward.

Video learning resources

Video resources here cover key topics such as **C#**, **GitHub**, **GitHub Desktop**, **Visual Studio**, **VS Code**, **NUnit**, and **Copilot**. For those new to programming, it is suggested to spend some time getting familiar with **C#**.

Step 1: Let’s introduce BugSweeper!

BugSweeper is a test automation tool used for all business verticals of **IGT** gaming.

BugSweeper uses the libraries **EgmClientsLite** and **UtpClient** to connect to an EGM to facilitate automated configuration and gameplay for our games.

- **EgmClientsLite** communicates with the **Test FI** to perform Foundation/AI-level actions (i.e. games configuration, machine-wide settings, pressing physical buttons, other **Test FI** features)
- **UtpClients** communicates with the game-side **UTP** server to perform game-specific actions (pressing screen buttons, seeding snippets, etc.).

BugSweeper overview

Feedback-driven development

This tool is tailored to the user needs, meaning your feedback is crucial . Throughout **BugSweeper**'s development the team relies on your feedback to make fixes and improvements.

Reach out to the BugSweeper Team for any feedback, thank you!

Step 2: Let's clone our first GitHub repository!

Now that we have **GitHub** access, let's jump to the BugSweeper Template repository and create our first repository.

This page should guide users on how to create and clone their repository to their local computer.

☝ From here on out we will use **VS Code** as our IDE, but users with **VS Pro 2022** can still follow along.

Step 3: Let's build and run our first tests!

At this point, for our **VS Code** users, after cloning the IDE should be look something similar to this:

image

If so, we are ready to keep moving ✔

Documentation on using VS Code for Testing exists in the **BugSweeper** wiki, but this workshop will provide steps as well.

VS Code Extensions

For our **VS Code** users (**NOT VS Pro** users), we will need to install additional extensions to run our tests:


image

On the far left-side, select the 'Building Blocks' icon labeled 'Extensions':

> 
> image

Here we can search for and install the two needed extensions:

- `.NET Core Tools`
- `C# Dev Kit`

These extensions set up our **VS Code C#** development environment to start working in.

Building the BugSweeper project

In **VS Code**, there are a couple ways to build the **BugSweeper** project. Navigate to the 'Explorer' tab on the far left-side at the top, where we previously found the 'Extensions' tab:

> 
> image

To build, right-click either the `Tests.csproj` or `BugSweeper.sln` file and select 'Build .NET Core project':

via .csproj file	via .sln file
	

If everything in your environment is set up, you should see a green `succeeded` result in the Terminal:


image

If not... refer to our wiki section: Building **BugSweeper** fails! If still encountering issues, please reach out to the **BugSweeper** Team.

Running the example tests

After a successful build, let's run the example **NUnit** tests! This workshop's version of the **BugSweeper** template comes with a few example, game-agnostic tests that should apply to virtually any **Ascent/GGA** game.

These example tests reside in `Workshop1\ExampleTests.cs`, an existing file within our **BugSweeper** Workshops repository.

So, to run these example tests we would need a game enabled and running. This is when you should have your EGM/AoW up with the installed Test FI:


image

In **VS Code**, jump to the 'Testing' tab on the far-left:

> 
> image


image

Hey! The example tests from `Workshop1\ExampleTests.cs` are now there! 😎

Not seeing them? Refer to our wiki section: My tests aren't being discovered in VS!

Ensure the `appsettings.json` in the BS project is targeting the IP address of the EGM.

- Using AoW? Assign `Address` to `127.0.0.1`.

Let's finally run our first test, `AddCredits`. This test will simply add 1000 credits to the EGM. To run the test, either right-click on the test and select the option or select the 'Run Test' icon to its right when hovering over it:


image

WOO! Running the test **SHOULD** result in 1000 credits being added to our game:


image

Not seeing this? **EgmClientsLite** not behaving? Refer to our wiki section: **EgmClientsLite** is failing to connect!

Step 4: Let's debug these tests!

Debugging is an essential skill for any programmer, as it allows you to identify and fix issues in your code. In this step, we will guide you through the process of debugging **NUnit** tests using **Visual Studio Code**.

Debugging involves setting breakpoints, running your tests in debug mode, inspecting variables, and controlling the flow of execution.

Here are the official **VS Code** docs on debugging, but below is a quick introduction:

What is a breakpoint?

A **breakpoint** is a marker that you can set in your code to tell the debugger to pause execution at that specific line. This allows you to inspect the state of your application at a particular point in time, which is useful for identifying and fixing bugs.

How to Debug a test/hit a breakpoint in Debug mode

1. **Open Your Test File**: In Visual Studio Code, open the file containing the NUnit tests you want to debug.
2. **Set a Breakpoint**: Click in the left margin next to the line number where you want to set a breakpoint. A red dot will appear, indicating that a breakpoint is set.

   
   image
3. **Debug Your Test**: In the **Testing** tab, right-click on the test you want to debug and select Debug Test.

   
   image

Analyzing variables when hitting breakpoint

When the debugger hits a breakpoint, the execution of your code will pause, and you can inspect the variables:

1. **Run and Debug view**: Users can inspect variables and expressions in the VARIABLES section of the Run and Debug view:

image

2. **Hover Over Variables**: Simply hover your mouse over a variable in the code to see its current value.

   image

Continue from breakpoint

Once you have inspected the variables and the state of your application, you can continue the execution:

1. **Continue Execution**: Press F5 or click the **Continue** button in the debug toolbar to resume running the code until the next breakpoint or the end of the test.
2. **Step Over**: Press F10 to step over the current line of code, which means the debugger will execute the line and pause at the next line.
3. **Step Into**: Press F11 to step into a function call, allowing you to debug inside the function.
4. **Step Out**: Press Shift+F11 to step out of the current function and return to the calling function.

Step 5: Let's play with Copilot AI!

Please refer to the official GitHub Copilot Chat docs for an in-depth explanation of all the features available for Copilot Chat.

For a quick crash course in getting up-and-running with **Copilot**, keep following!

Installing GitHub Copilot

For **VS Pro**: **Copilot** should be integrated with the latest version of the IDE. If you don't already have it, update your **VS Pro**.

For **VS Code**: There are two additional extensions needed to be installed in order to integrate **Copilot** into our IDE:

image

Once installed, open the **Copilot** chat window here:

image

You may need to sign in to your **GitHub** account to use **Copilot**, go ahead and do that.

Having issues with Copilot? Refer to our wiki section: I can't connect to **GitHub Copilot**!

With **Copilot** up, we can start leveraging its AI powers to help us write tests!

Using Copilot

For our users who are new to programming, there are high-level programming concepts that may be hard to grasp initially. Fortunately, we can use **Copilot** to clarify these concepts!

For example, in `ExampleTests.cs`, the test `PlayGameUntilWin()` executes a method `PlayOneGame()` until a win is achieved. To understand its logic, one could simply prompt **Copilot** with the question:

> Can you explain what the method `PlayOneGame()` does, line-by-line?

This should describe `PlayOneGame()` from one individual step to the next. At the end **Copilot** summarizes with:

> Summary: The PlayOneGame method simulates playing a single game on the machine. It: 1. Prepares the machine by adding credits. 2. Sets the bet amount. 3. Starts the game by pressing the “Spin” button. 4. Waits for the game to start and end. 5. Ensures the machine returns to an idle state before finishing. 6. Cleans up event subscriptions to avoid resource leaks.

Great job, **Copilot**!

What else can we do with this AI? We can certainly have **Copilot** generate us code as well!

We can access the Chat to prompt for code generation in several ways. Another way to submit prompts is with the **Inline Chat**!

Here we can select a block of code and right-click on it to contextualize the prompt we are about to send:


image

In this example we will highlight the test `TestUtpModulesExist` and ask it to update the logic to print out all modules and their respective commands:

> Can you modify this test to print out all the modules, as well as the command for each of those modules? The command string is found in the command.Command property.

This prompt given this test should generate the following code suggestion:


image

Accepting the changes should update `TestUtpModulesExists` to:

```
[Test]
public async Task TestUtpModulesExists()
{
    var connected = await Utp.Connect(CancellationToken);
    Assert.That(
        connected,
        Is.True);
    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}");
        }
    }
}
```

With this change, running `TestUtpModulesExists` should result in output within ‘Test Results’ that should look similar to:

```
Module: UtpController
  Command: InitializeModules
  Command: ReinitializeModules
  Command: GetHelp
  Command: GameShutdown
  Command: PauseStatusUpdated
Module: MPT
  Command: GetGamePlay
  Command: QueueNextGamePlay
  Command: SetGamePlayAutomated
  Command: GetGamePlayAutomated
```

```
  Command: GetCurrentPersistence
  Command: SetCurrentPersistence
  Command: SetPlayerDecisions
.........
```

Viola! Copilot helped us extend this test to do more!

Conclusion

That wraps up **Workshop 1**, congratulations!

More documentation on **BugSweeper** functionality are found in the **BugSweeper** Template wiki, which covers other concepts we will be exploring in the later workshops.