AI-Enhanced Unit Testing with xUnit: Optimizing Test Creation through GitHub Copilot

AzraJabeen Mohamed Ali

Independent researcher, California, USA
Azra.jbn@gmail.com

Abstract

The development of robust software systems relies heavily on effective unit testing to ensure code reliability and correctness. With the increasing complexity of modern applications, writing and maintaining unit tests can become a time-consuming task. This research explores the integration of GitHub Copilot, an AI-powered code completion tool, with the xUnit framework to optimize the unit test creation process. We investigate how GitHub Copilot can assist developers in generating, refining, and maintaining unit tests more efficiently, focusing on its impact on both productivity and code quality. Through a series of case studies and experiments, we assess Copilot's ability to suggest test cases, handle edge conditions, and generate mock setups for testing dependencies. The results indicate that AI assistance can significantly reduce the time spent writing tests, enhance test coverage, and support developers in addressing complex scenarios. Additionally, we discuss the limitations and challenges of using AI for test generation, particularly in ensuring test correctness and preventing over-reliance on AI-generated suggestions. This study contributes to the growing body of knowledge on AI-assisted software development, offering insights into the potential of combining machine learning tools with traditional testing frameworks like xUnit to create more efficient and scalable testing practices.

Keywords: xUnit, Github Copilot, AI, productivity, code quality, test generation, mock

1. Introduction

Unit testing is a cornerstone of modern software development, ensuring that individual components of a system function correctly in isolation. As applications grow in complexity, writing comprehensive unit tests becomes increasingly time-consuming and error-prone, often requiring developers to create a wide range of test cases, mock dependencies, and handle edge scenarios. Traditional methods for generating tests are tedious, and the constant need for test maintenance can strain development teams.

With the advent of AI-powered tools, developers are now exploring ways to enhance and streamline their workflows, including the process of unit testing. GitHub Copilot, an AI-based code completion tool powered by OpenAI's Codex model, has emerged as a potential solution to assist developers by suggesting code snippets, including unit tests, directly in the Integrated Development Environment (IDE). Copilot's ability to analyze the context of the code being written and propose relevant test code holds promise for reducing the manual effort required to write unit tests, thus improving both productivity and code quality.

This research focuses on integrating GitHub Copilot with the widely used xUnit testing framework to evaluate how AI can optimize the creation of unit tests. xUnit is a popular open-source framework for .NET

applications, known for its simplicity and extensibility. It is a prime candidate for investigating AI-enhanced testing because of its widespread adoption and rich ecosystem.

We explore the benefits and challenges of using AI to assist in test creation, including how Copilot can help generate common test structures, handle mock dependencies, and suggest edge case scenarios. Additionally, we examine how Copilot's suggestions compare with human-written tests in terms of correctness, coverage, and efficiency.

Through a combination of case studies, experiments, and developer feedback, this research aims to provide insights into the practical use of AI in unit testing and its potential to improve the software development lifecycle. Specifically, we seek to understand whether AI can enhance the testing process within the xUnit framework, offering a more efficient, reliable, and scalable approach to unit test creation.

Test Automation:

Test automation is the process of using specialized software tools or scripts to automatically execute tests on software applications to ensure their functionality, performance, security, and reliability. It is an essential practice in modern software development, especially in agile and continuous integration (CI) environments, where applications are frequently updated, and manual testing can be time-consuming and error-prone.

Types of tests in Test Automation:

There are various types of automated tests that can be performed:

- **Unit Tests:** Test individual components (such as functions or methods) to ensure they work as expected in isolation. Tools like xUnit, NUnit, or JUnit are typically used.
- **Integration Tests:** Test the interaction between multiple components or systems to ensure they work together correctly.
- **Functional Tests:** Ensure that the software functions as expected from the user's perspective. Tools like Selenium, Cypress, or TestComplete are often used.
- **Regression Tests:** Ensure that new code changes do not negatively impact existing functionality.
- **Performance Tests:** Assess the application’s performance under load (e.g., load testing, stress testing).
- **Acceptance Tests:** Ensure that the software meets the business requirements and is ready for deployment.
- **End-to-End (E2E) Tests:** Test the entire application flow, from start to finish, often simulating real-world usage.

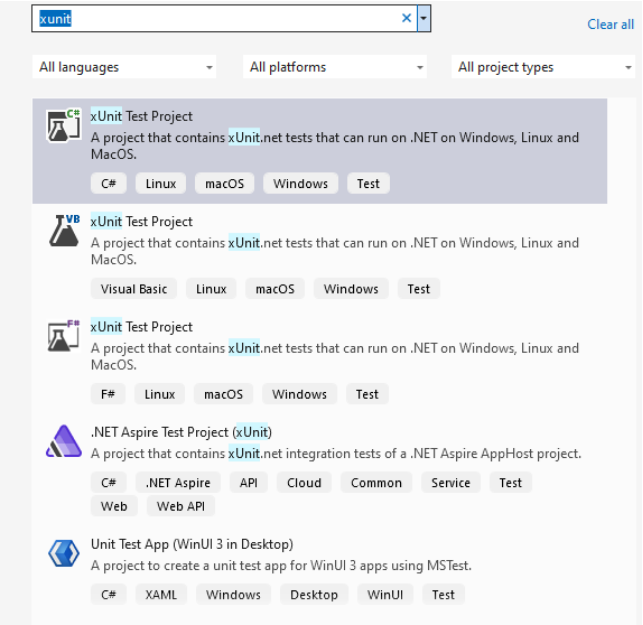
Setting up xUnit and Github Copilot for unit testing:

Before using GitHub Copilot to assist with writing tests, you need to set up xUnit in your project.

Installation and setting up xUnit test Project:

Ensure that you have Visual Studio installed. If you don't have it, you can download it from the official site. After launching Visual Studio, Click on Create a new project.In the Create a new project dialog, search for xUnit in the search box at the top.Choose the xUnit Test Project template from the list of project templates Fig-1.Click Next.In the Project name field, give your test project a name (e.g., MyApp.Tests).Choose the location where you want to save the project.Click Create.

Fig-1:



If there is a need to test an existing application or class library, then a reference is to be added to the project which is to be tested.Right-click on the Dependencies node in the Solution Explorer.Select Add Project Reference.Check the box next to the project you want to reference and click OK.

By default, Visual Studio adds the necessary xUnit NuGet packages to the project. If they are not added, follow these steps to install them manually:Right-click on the Dependencies node in the Solution Explorer.Select Manage NuGet Packages.In the NuGet Package Manager, click on the Browse tab and search for xunit.Install the xunit package.Additionally, install xunit.runner.visualstudio to enable running tests within Visual Studio.

Steps to Install Github Copilot:

- GitHub Copilot requires Visual Studio 2022 or newer as a prerequisite.GitHub Copilot subscription is needed. If we don’t have it yet, we can sign up for a trial or a paid plan at GitHub Copilot's official page.
- Open Visual Studio on your computer.Make sure to use Visual Studio 2022 or later for compatibility with GitHub Copilot.
- In the top menu, click Extensions > Manage Extensions.
- In the Manage Extensions window, click on the Online tab on the left side.
- In the search box, type GitHub Copilot. Find GitHub Copilot in the search results.
- Click the Download button next to the extension.

- After the extension is installed, you will be prompted to restart Visual Studio for the changes to take effect. Click Restart Now.
- After restarting, GitHub Copilot should be ready to use. However, it is necessary to sign in to GitHub to authenticate.
- If GitHub is not signed in yet, Visual Studio will prompt to log in to GitHub account.
- Click Sign in and follow the steps to authenticate using GitHub account. It is to ensure GitHub Copilot subscription is active, as it needs a valid subscription to use the service.

Writing Unit Tests Using xUnit and Github Copilot:

First, we have a simple class “TestArray.cs”in which contains the method “FindLargestElement” to find the largest element in the given array. In traditional method we use to code the method. Now using Github Copilot , just by adding comments in plain English to describe what we need. Fig-2 For instance, if we give the comment saying “write method to find the largest element in an array”, GitHub Copilot uses these comments as a natural language prompt to generate relevant code.GitHub Copilot will analyze this comment in the context of your code and try to generate code that fits what you are asking for.

Fig-2:

```
//write method to find the largest element in an array
```

Copilot uses Natural language processing(NLP) techniques to understand comment. It uses machine learning models (based on OpenAI's Codex) to generate code completions from comments. It's trained to understand the intent behind a comment. Fig-3Once it processes comment and analyzes the context, GitHub Copilot generates code that it thinks is the best match for the comment.

Fig-3:

```
public int FindLargestElement(int[] arr)
{
    int max = arr[0];
    for (int i = 1; i < arr.Length; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    return max;
}
```

Copilot will provide these suggestions as **auto-completions**. We can press Tab to accept the suggestion or modify it if necessary. If the initial suggestion isn’t exactly what we want, can keep typing, and Copilot will adjust the code based on the updated context.
GitHub Copilot continues to learn and improve based on the feedback it gets from developers. If a suggestion is accepted or rejected, or if you modify a suggestion, it will adapt and improve over time.

Using GitHub Copilot to Write Tests:

GitHub Copilot can significantly speed up the process of writing unit tests. When writing a test, GitHub Copilot can provide auto-completion suggestions for code structures like method names, arguments, and

assertions. Here's how to use Copilot in the context of writing unit tests.By typing test method signature, GitHub Copilot will suggest code snippets. When we start typing as per below Fig-4

Fig-4:

```
[Fact]
public void TestLargeNumber()
```

Copilot will suggest the rest of the test code, including the Arrange, Act, and Assert sections. It will automatically infer that you want to test a method that adds two numbers.
We can either accept the suggestion by pressing Tab or modify it as needed. For instance, Copilot might suggest the following as per Fig-5:

Fig-5:

```
//Arrange
TestArray array = new TestArray();
int[] arr = { 10, 20, 30, 40, 50 };
//Act
int result = array.FindLargestElement(arr);
//Assert
Assert.Equal(50, result);
```

If we want to add more tests for edge cases or different input values, start typing a similar test name, and Copilot will continue to help.If code involves dependencies, GitHub Copilot can help writing mock setups.Adding comments can help Copilot understand the intent of tests and suggest more relevant code.Copilot can automatically insert the necessary xUnit attributes ([Fact], [Theory], etc.) based on the type of test we need.Copilot can help writing tests for edge cases like negative numbers, large inputs, or null values.

Attributes used inxUnit:

1. **[Fact]:** It marks a method as a test method that doesn't take parameters. It is used to define simple unit tests that refer Fig-6.
   Fig-6:

```
[Fact]
public void TestLargeNumber()
{
    //Arrange
    TestArray array = new TestArray();
    int[] arr = { 10, 20, 30, 40, 50 };
    //Act
    int result = array.FindLargestElement(arr);
    //Assert
    Assert.Equal(50, result);
}
```

2. **[Theory]:** It Marks a method as a test method that takes parameters. A theory can be used with data sources like inline data or data attributes. It allows parameterized tests, where the same test method is run with different inputs that refer Fig-7.Attributes used with [Theory] are:
   - **[InlineData]:** It is used to pass parameters directly to the test method.
   - **[ClassData]:** Provides test data from a class.
   - **[MemberData]:** Provides test data from a method or property.

Fig-7:

```
[Theory]
[InlineData(20, 60, 30, 80, 50, 80)]
public void TestLargeNumber_2(int a, int b, int c,
    int d, int e, int expected)
{
    TestArray array = new TestArray();
    int[] arr = { a, b, c, d, e };
    int result = array.FindLargestElement(arr);
    Assert.Equal(expected, result);
}
```

3. **[BeforeAfterTest]:**These attributes are used to run code before or after a test method runs.In xUnit, setup and teardown behavior can be implemented using constructor/destructor or by implementing an interface. However, the [BeforeAfterTest] allows more fine-grained control.
4. **[Fact(Skip = "reason")]:** It is used to skip a test method and provides a reason. It is used to temporarily disable a test without deleting it.
5. **[Trait]:**It is used to tag tests with metadata, such as categories or custom tags. It is to categorize tests or mark them with specific attributes, which can be helpful when filtering tests in the test runner.
6. **[Collection]:**Itmarks a test class as part of a test collection.This allows us to share setup and teardown logic across multiple test classes.
7. **[AssemblyInitialize] / [AssemblyCleanup]:** It is to run code before or after all tests in an assembly. These are usually implemented as methods with a special signature.Used for setting up and cleaning up things that are expensive and only need to run once per assembly (such as opening/closing a database connection).

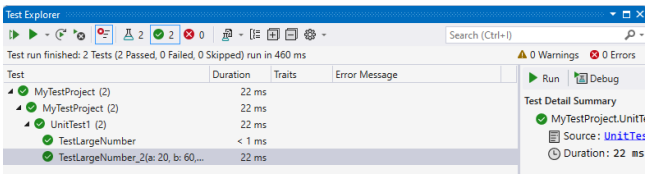
Running Tests:

It is necessary to build the solution ensuring that all tests are compiled by pressing Ctrl + Shift + B. ThenGo to Test > Windows > Test Explorer to open the Test Explorer panel.In the Test Explorer, we can see all the available tests. We can run:

- **All Tests**: Click on the Run All button.
- **Individual Tests**: Right-click on specific tests and select Run Selected Tests.
- **Group of Tests**: We can also select a group or class and run them together.

After running the tests, Visual Studio will display the results in the Test Explorer window as per Fig-8.The Test Explorer will show the status of each test (passed, failed, skipped, etc.).If a test fails, the error details are displayed in the Output window or directly in the Test Explorer.

Fig-8:



Benefits of AI Incorporated xUnit:

1. **Increased Development Speed:**GitHub Copilot's AI-powered suggestions can instantly generate test cases based on the code being written. This dramatically reduces the time developers spend manually crafting tests, allowing them to focus more on the core development process.With AI

assistance, developers can quickly write tests for new code or modify existing tests without having to spend excessive time learning the intricacies of the framework or coming up with test cases from scratch.

2. **Improved Code Quality:**Copilot can automatically suggest well-structured unit tests, ensuring that developers cover all important use cases, including edge cases, without forgetting essential scenarios.AI helps maintain consistency in test code, reducing the likelihood of human error in test creation and ensuring that tests follow best practices and coding standards.

3. **Enhanced Test Coverage:** Copilot can identify areas of code that may require additional tests, helping ensure better coverage of different code paths, including error handling and edge cases, which might otherwise be overlooked.By suggesting common error-handling scenarios and boundary tests, Copilot helps developers cover a broader spectrum of potential application behaviors.

4. **Reduced Manual Effort:** Writing repetitive test structures like assertions, setup/teardown methods, and mock objects is tedious. Copilot streamlines this process by suggesting boilerplate code and refactoring it as needed.Copilot can help with generating mock objects and stubs, which are often needed when testing classes that rely on external dependencies, reducing manual configuration.

5. **Better Focus on High-Level Logic:**GitHub Copilot acts as a helpful assistant, enabling developers to focus on more complex and creative aspects of the application while leaving the tedious task of writing boilerplate tests to the AI. This enables a more efficient workflow and encourages higher-level problem solving.

6. **Support for Continuous Integration/Continuous Deployment (CI/CD):** Automated unit tests can be integrated into CI/CD pipelines, and with Copilot's help in writing tests, developers can quickly adapt to changing code. This enables faster feedback loops, identifying potential issues earlier in the development process.Copilot ensures that automated regression tests are written and updated swiftly, guaranteeing that new changes don't break existing functionality.

7. **Cost Savings:**By accelerating test creation and reducing the manual effort required for writing tests, AI-enhanced testing leads to faster time-to-market, lower development costs, and reduced labor expenses for testing teams.With better test coverage and faster feedback, Copilot can help identify defects early in the development process, reducing the cost of fixing bugs later in production.

8. **Higher Test Reliability:**Copilot's machine learning capabilities allow it to suggest test cases that could potentially catch bugs or logic errors in the code that developers might have missed manually. By analyzing common patterns, Copilot can often identify flaws in test logic before they become a problem.As code evolves, Copilot can help developers update or modify existing tests in real-time, ensuring that tests stay relevant and in sync with the latest codebase.

Conclusion:

The integration of AI-assisted tools like GitHub Copilot with the xUnit framework presents a transformative approach to unit testing, addressing the challenges faced by developers in creating efficient, comprehensive, and reliable test cases. Through the capabilities of GitHub Copilot, which leverages

machine learning to generate context-aware test code, developers can significantly enhance their productivity by automating repetitive tasks and focusing more on high-level application logic. This research has highlighted several key benefits of this integration, including improved test coverage, faster test creation, reduced manual effort, and enhanced code quality.

By automating the generation of unit tests, GitHub Copilot can suggest relevant test cases for various scenarios, including edge cases and error handling, that developers might overlook. Furthermore, the AI-powered tool assists in maintaining consistency across test code, ensuring that tests follow best practices while also supporting continuous integration and continuous deployment (CI/CD) pipelines, which is critical for modern software development practices.

While AI-assisted unit testing with xUnit offers clear advantages in terms of efficiency and coverage, it also introduces certain challenges. The reliance on AI-generated suggestions may require developers to carefully validate the correctness and relevance of the generated tests. Additionally, over-reliance on AI may lead to a reduction in critical thinking or understanding of the underlying code.

Nevertheless, the potential of AI-enhanced testing with xUnit, through tools like GitHub Copilot, offers significant promise in accelerating the development cycle, improving test accuracy, and ultimately delivering more reliable software. Future research can focus on refining these AI tools, addressing limitations, and exploring their integration with other testing frameworks and platforms to further optimize the software development lifecycle.

In conclusion, AI-enhanced unit testing using GitHub Copilot with the xUnit framework is a powerful step towards more efficient, scalable, and accurate testing practices, ultimately contributing to better-quality software and a more streamlined development process.

References

[1] Microsoft, “Best practices for using GitHub Copilot” https://docs.github.com/en/copilot/using-github-copilot/best-practices-for-using-github-copilot(March 2024)

[2] Agrawal Vishesh, “Introduction to Unit Testing” https://medium.com/@agrawalvishesh9271/mastering-unit-testing-in-net-core-1-aad9516858e0(Aug16, 2023)

[3] Bhrugen Patel “Complete Guide to Unit Testing in .NET Core (NUnit and xUnit): Master unit testing with NUnit, xUnit, and MOQ with a real-world N-Tier web application” Packt Publisher (Oct 18, 2021)

[4] xUnit.Net “Getting Started with xUnit.net” https://xunit.net/docs/getting-started/v2/netfx/visual-studio(Jan 2024)

[5] Andrea Chiarelli, “Testing your C# code with xUnit“https://www.codemotion.com/magazine/languages/testing-your-c-code-with-xunit/(Jul01, 2020)

[6] Andrea Chiarelli, “Using xUnit to Test your C# Code“ https://auth0.com/blog/xunit-to-test-csharp-code/ (Apr 07, 2020)

[7] Microsoft, “Testing with the xUnit Framework - Overview (2 of 12) | Automated Software Testing“ https://learn.microsoft.com/en-us/shows/visual-studio-toolbox/testing-with-the-xunit-framework-overview-2-of-12-automated-software-testing(Nov29, 2022)

[8] Robert Martin “Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C.

Martin Series) 1st Edition" Pearson publisher (Sep 10, 2017)

[9] Kent Beck "Test Driven Development: By Example" Addison-Wesley Professional Publisher (Nov, 2002)

[10] ChrisMinnick"Coding with AI"For dummies Publisher (Mar26, 2024)