Understanding Page Object Model (POM) in Selenium

This document provides a comprehensive guide to the Page Object Model (POM) design pattern in Selenium. It covers the key concepts, benefits, components, and implementation strategies for POM, along with practical code examples and common pitfalls to avoid. The document is structured to provide a clear understanding of POM, enabling developers and testers to effectively implement and maintain automated tests.

M by Marcus Boykin

Page Object Model (POM): Why It Matters for Reliable Test Automation

Think of your company's website or application as a building. Every page is like a different room with buttons, forms, and menus—like doors, light switches, and furniture. When our team tests the website, we need to make sure everything works perfectly in every room.

The Problem:
If we check each room manually every time something changes, it's slow and error-prone. Even when we automate the checks, if we mix the "how to get around the room" with "what we're actually testing," it quickly becomes messy and hard to fix when the website changes.

The Solution – Page Object Model (POM):
POM is a smarter way to automate testing. We create a "blueprint" for each page—like a floor plan that knows where every button, form, and label is. This blueprint also knows how to use them (e.g., click a button, type into a field).

Why Use the Page Object Model?

Easier Maintenance

POM significantly eases the maintenance of automated tests. When a web page undergoes changes, such as alterations to element locators or the addition of new elements, only the corresponding page object needs to be updated. This localized modification prevents the need to update every test that interacts with that page, saving considerable time and effort.

Cleaner Test Logic

By encapsulating page-specific logic within page objects, tests become more readable and focused on the actual testing steps. This separation of concerns makes it easier to understand the purpose of each test and reduces the likelihood of introducing errors during test development or modification. Cleaner test logic enhances collaboration among team members and facilitates more efficient debugging.

Reduced Code Duplication

POM promotes the principle of "Don't Repeat Yourself" (DRY) by centralizing common page interactions and element locators within page objects. This eliminates the need to duplicate code across multiple tests, reducing the overall codebase size and improving maintainability. Reusing page objects across tests also ensures consistency in how elements are accessed and interacted with, further minimizing the risk of errors.

Overview

In order to automate Google Chrome with Selenium in C#, you need to install the following NuGet packages:

1. **Selenium.WebDriver**
   Core Selenium WebDriver API for controlling browsers.
2. **Selenium.Chrome.WebDriver**
   Supplies the ChromeDriver executable and Chrome-specific automation support.
3. **DotNetSeleniumExtras.WaitHelpers**
   Provides extension methods (e.g., `ExpectedConditions`) for more readable and reliable wait conditions.

Steps to Install via Visual Studio:

1. **Go to** `Project` → `Manage NuGet Packages`.
2. **Search for** and install the three packages above.
3. **Alternatively**, use the Package Manager Console (PMC) commands

```
Install-Package Selenium.WebDriver
Install-Package Selenium.Chrome.WebDriver
Install-Package DotNetSeleniumExtras.WaitHelpers
```

Quick Tip

After adding these packages, you can write C# Selenium tests (including waits using `ExpectedConditions`) and run them seamlessly in Visual Studio—making it straightforward to automate browser interactions in Chrome.

Key Components of POM in Selenium

Component	Responsibility
Base Page	Common functionality for all pages
Page Objects	Individual classes representing different web pages
Utilities	Support classes (e.g., Driver setup, Wait helpers)
Tests	Use page objects to perform automated tests

- **Base Page:** The base page class provides common functionality and utilities that are shared across all page objects. It typically includes the WebDriver instance, WebDriverWait instance, and methods for basic interactions like navigating to a URL or handling alerts.
- **Page Objects:** Each page object represents a specific web page and contains the elements and methods for interacting with that page. Elements are typically defined as private fields, while methods encapsulate the actions that can be performed on the page.
- **Utilities:** Utility classes provide helper functions for common tasks like setting up the WebDriver, waiting for elements to be visible, or reading data from external sources. These classes help to reduce code duplication and improve the overall structure of the test automation framework.
- **Tests:** Test classes use page objects to perform automated tests. Tests should be focused on verifying specific functionality and should not contain any page-specific logic. This separation of concerns makes tests easier to read, understand, and maintain.

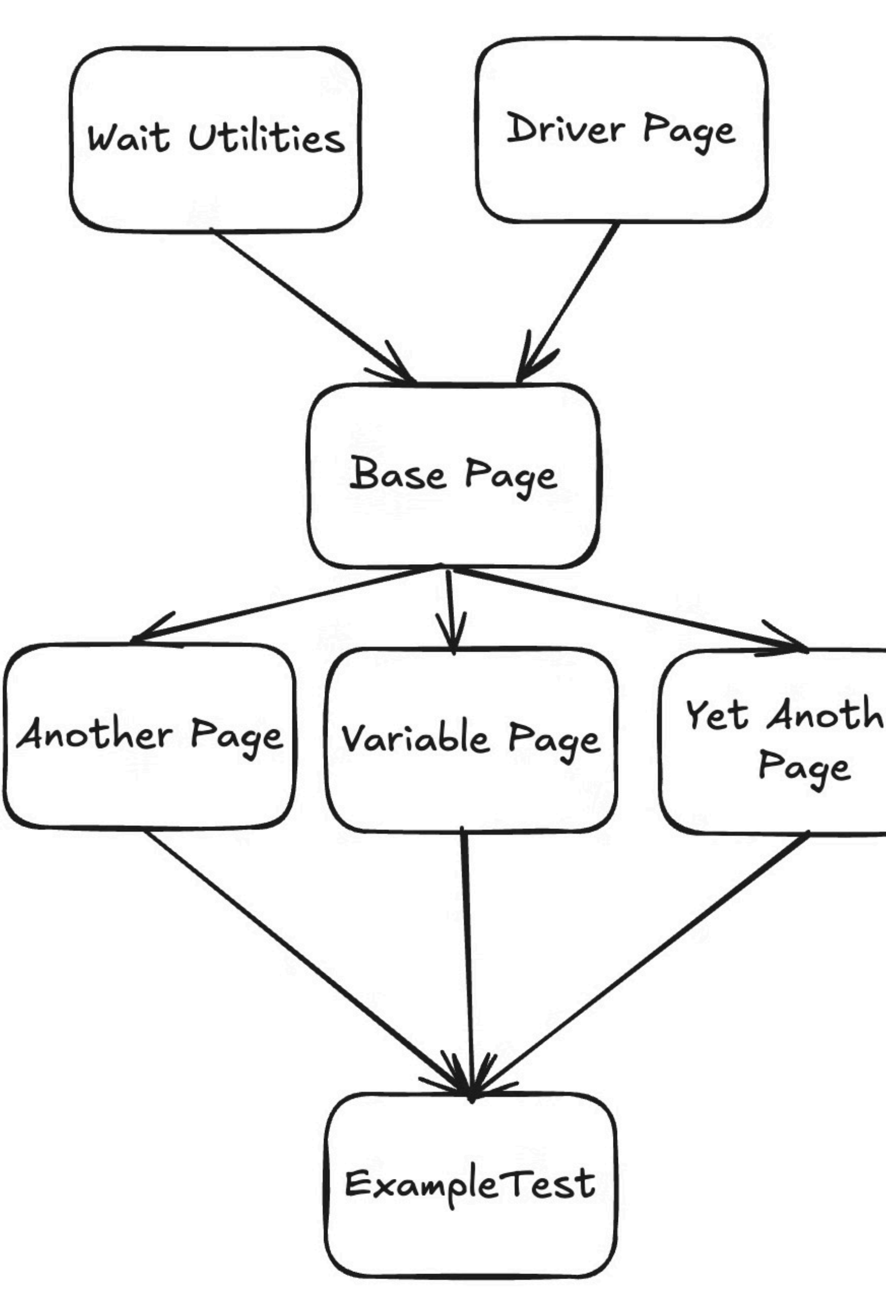
Folder Structure Overview

A well-organized folder structure is essential for maintaining a scalable and maintainable POM-based test automation framework. Below is a common folder structure that can be adapted to suit the specific needs of a project:

- **Pages:** This folder contains the page object classes, with each class representing a specific web page. The classes inherit common functionalities and utilities from the BasePage, centralizing common elements and simplifying the structure.
- **Tests:** This folder contains the test classes. Each test class should correspond to a specific feature or scenario and use the page objects to interact with the application. This separation ensures tests are focused on verification and not page-specific logic.
- **Utilities:** This folder houses the utility classes and helper functions. Common tasks such as WebDriver setup, handling waits, logging, and data input are managed centrally, reducing redundancy and improving maintainability. Classes like DriverPage.cs and WaitUtilities.cs are stored here.

How This Project is Organized

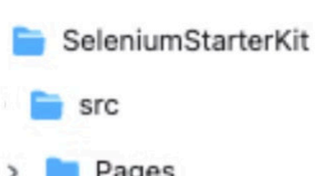
This project is like a **toolbox** that helps us build and test websites automatically. Each folder is like a different section of the toolbox, holding special tools that work together:

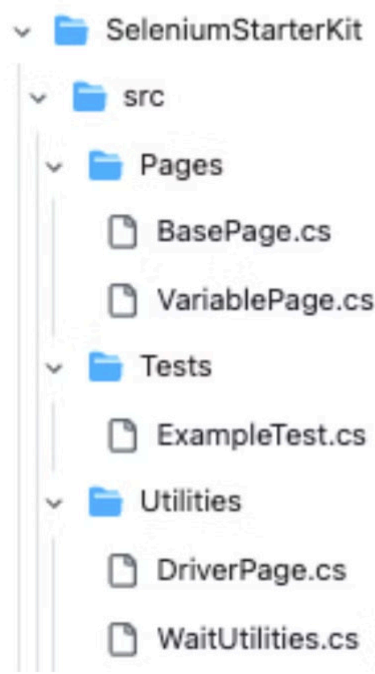


Visual Representation

The following diagram illustrates the relationships between the key components of a POM-based test automation framework:

- **Base Page -> Page Objects (Inheritance):** Page objects inherit common functionality from the base page, promoting code reuse and consistency.
- **Utilities -> Base Page (Support):** Utility classes provide support functions to the base page and page objects, such as WebDriver management and wait helpers.
- **Test -> Page Objects (Uses):** Test classes use page objects to interact with the application under test, keeping the test logic clean and focused.





Setup Structure

Hardest part made easy.

src

Pages

Tests

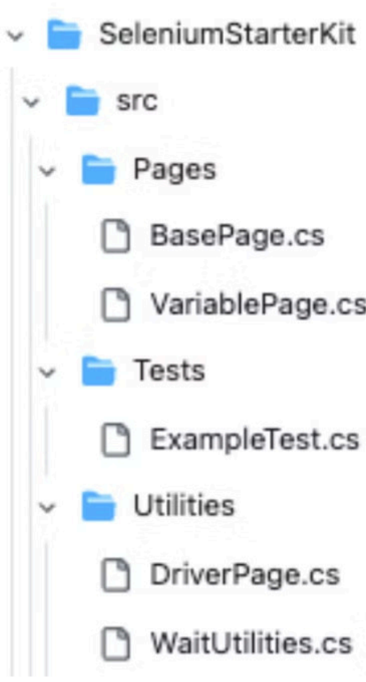
Utilities

🧑‍💻 How It All Works Together

1. **src** → Contains the robot.
2. **Pages** → Show the robot how to work with website pages.
3. **Tests** → Give the robot tasks.
4. **Utilities** → Help the robot with browsers and waiting.

Together, they make our **automation robot smart and ready to work!** 🦾

SeleniumStarterKit
src
Pages
Tests
Utilities

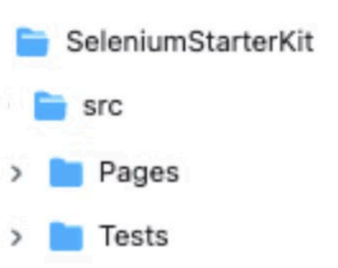


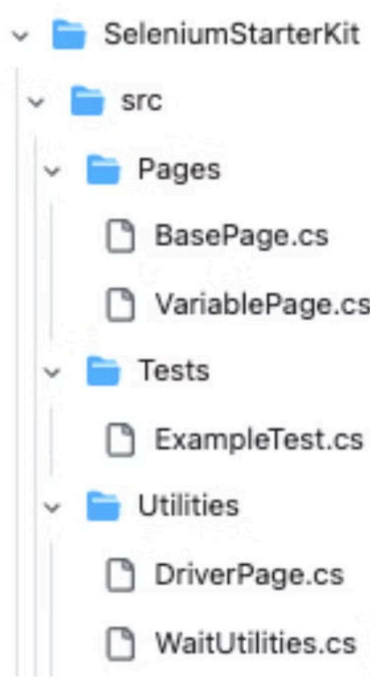
🗂️ Pages Folder

This folder holds the “pages” of the website we are testing. Think of it like making a robot that knows how to click buttons and type in text on different website pages.

- **BasePage.cs:** This is like the robot’s “brain.” It teaches all other pages how to do basic things like clicking, typing, and waiting.

VariablePage.cs: This is like a robot that knows how to work with a specific page on a website. It uses the BasePage brain to help it.

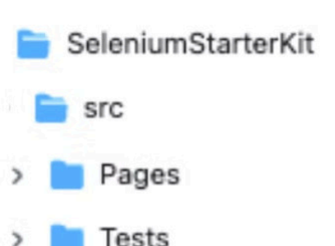


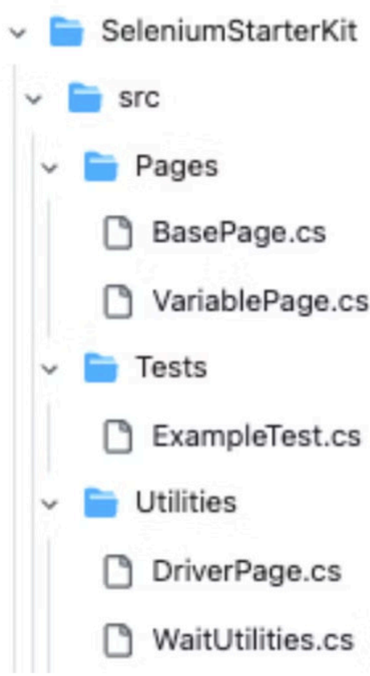


🗂️ Tests Folder

This folder holds the **tests**. A test is like giving our robot instructions: **"Go to the website, type in something, click this button, and check if it works!"**

ExampleTest.cs: This is an example test. It tells the robot exactly what to do on the website to make sure everything works.





🗂️ Utilities Folder

This folder holds **helpers**. These are like little tools that make the robot's job easier.

- **DriverPage.cs:** This helps the robot **open and close the web browser** (like Chrome or Edge).

WaitUtilities.cs: This teaches the robot **how to be patient**. Sometimes the robot needs to wait for things to load on a website.

🧑‍💻 Reading Your First Class (Don’t Worry, I’ll Make It Easy)

Before we jump into writing code, I want to show you something important:
This is what a basic **class** looks like in C#. You’ll see this a lot:

```
public class BasePage {
    protected IWebDriver driver;
    protected WebDriverWait wait;

    public BasePage(IWebDriver driver) {
        this.driver = driver;
        wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));
    }
}
```

If this looks like **an alien language**—that’s okay.
If you know what this is—stick with me. You’ll still pick up some clarity.

Let’s **break it down**, even if you’ve never seen code before.

1️⃣ Classes Are Blueprints

Think of a **class** like a blueprint for a robot.
This blueprint **tells the robot what it knows** and **what it can do**.

- It might **know how to open a door**.
- It might **know how to wait for a website to load**.

```
public class BasePage {
```

- **public** means **anyone can use this blueprint**.
- **class** means **we're making a blueprint**.
- **BasePage** is the **name of our blueprint**.

Pro Tip for Experienced Readers:
Think of `public class BasePage` as saying:
"This is a reusable base class other pages will extend

2️⃣ Curly Braces `{ }` Hold Steps

Whenever you see **{ }**, it means:

“Everything inside here is part of this thing.”

```
public class BasePage {
    // Stuff inside this is part of BasePage
}
```

- **{** opens the blueprint.
- **}** closes it.

If you know code, **this is your scope**.
If you’re new, just remember: **Open brace = start steps. Close brace = done.**

3️⃣ Semicolons `;` End Each Step

In code, **;** is like a period in a sentence.

This:

```
protected IWebDriver driver;
```

… is just saying:

- **“Hey robot, remember something called driver.”**

We put **;** because the robot needs **clear stops**.

For Experts:
Yes, semicolons are end-of-statements in C-like languages. Beginners just need to know **“stop here”**.

4️⃣ Methods Are Actions

A method is just **an action the robot can do**.

Example:

```
public void DoSomething() {
    // steps the robot will do
}
```

This is saying:

- **public** → Anyone can tell the robot to do this.
- **void** → This action doesn't give anything back; it just "does."
- **DoSomething()** → This is the **name** of the action.
- **{ }** → Inside here are the **steps**.

Advanced Readers:
Yes, this is a method with no return type (void). We'll touch on return types later.

5️⃣ Let It Click Together

Let's **read our first class again**:

```
public class BasePage {
    protected IWebDriver driver;
    protected WebDriverWait wait;

    public BasePage(IWebDriver driver) {
        this.driver = driver;
        wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10));
    }
}
```

This is now what you can see:

Code	What It Really Says
`public class BasePage`	**"Open blueprint for everyone called BasePage."**
`{`	**"Start listing what the robot knows and can do."**
`protected IWebDriver driver;`	**"The robot knows about a browser (driver)."**
`protected WebDriverWait wait;`	**"The robot knows how to wait (wait)."**
`public BasePage(IWebDriver driver)`	**"When we make this robot, we give it a browser."**
`this.driver = driver;`	**"Remember this browser for later."**
`wait = new WebDriverWait(driver, 10 sec);`	**"Also remember how to wait 10 sec."**
`}`	**"Done with this part."**

Common Mistakes to Avoid (and How to Make Your Life Easier)

1. **Mixing Test Logic with Page Actions**
    - **What this looks like:** Your page objects contain both "how to click a button" (actions) and "does clicking the button do what we expect?" (test logic).
    - **Why it matters:** When actions and tests blur together, it's harder to spot bugs and maintain your code. Keeping them separate allows you to quickly see what's being tested versus how the page is manipulated.
    - **How to avoid it:** Let your page objects handle the actual interactions (e.g., clicking buttons, typing text), while your test scripts focus on verifying the expected outcomes.
2. **Using Static Methods in Page Objects**
    - **What this looks like:** You rely heavily on static methods, which makes it tough to manage individual WebDriver instances and page-specific states.
    - **Why it matters:** Static methods reduce flexibility and hinder the reusability of your page objects. They can also complicate scaling your tests or introducing variations.
    - **How to avoid it:** Use object instances instead of static methods. This way, you can encapsulate page details and handle the driver more efficiently.
3. **Not Using a 'Base Page' for Shared Behaviors**
    - **What this looks like:** Each page object implements the same utilities—like waiting for elements or handling timeouts—over and over again.
    - **Why it matters:** A base page fosters code reusability and adheres to the **DRY (Don't Repeat Yourself)** principle. By centralizing common methods (e.g., logging, waiting for elements), you only write and maintain the code once. All page objects that inherit from the base page automatically gain these shared capabilities—keeping your tests consistent, clean, and easier to update.
    - **How to avoid it:** Identify recurring actions and place them in a base page class. Let each individual page object inherit these utilities, streamlining your framework.

Bottom Line
By avoiding these mistakes, you'll build a test automation framework that's easy to **maintain**, **scale**, and **extend**. Keeping test logic separate from page actions, steering clear of static method overuse, and leveraging a base page for shared functionality sets you up for success—so you can focus on writing tests that truly matter.

DriverPage

🚗 Driver Page – What Does It Do?

The **Driver Page** is like **the driver of our robot's car (the web browser)**.

Every time we run a test, we need to **open a browser** (like Chrome), **drive it around the website**, and **close it when we're done**.
DriverPage.cs is the class that **handles all of that** for us.

```
using OpenQA.Selenium.Chrome;
using OpenQA.Selenium;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SeleniumStarterKit.src.Utilities
{
    public class DriverPage
    {

        private static IWebDriver Driver;

        public static IWebDriver GetDriver()
        {

            if (Driver == null)
            {
                InitializeDriver();
            }

            return Driver;

        }


        public static void InitializeDriver()
        {
            ChromeOptions options = new ChromeOptions();
            options.AddArgument("--incognito");

            Driver = new ChromeDriver(options);

        }

        public static void CloseBrowser()
        {

            if (Driver != null)
            {

                Driver.Quit();
                Driver = null;
            }
        }
    }
}
```

}

🧑‍💻 Breaking Down DriverPage.cs – What's Happening Inside?

Now that you know **DriverPage.cs is like the car engine** that starts, drives, and stops the browser, let's **walk through what this code is actually doing** – step by step.

🔍 Line-by-Line Breakdown

Line in the Code	What It Means (Plain English)
`using OpenQA.Selenium.Chrome;` and others	"These are **toolboxes** we're grabbing to work with Selenium and the browser."
`namespace SeleniumStarterKit.src.Utilities`	"This is like a **folder label**, so we know this class lives in Utilities."
`public class DriverPage {`	"We're **creating a blueprint called DriverPage** that everyone can use."
`private static IWebDriver Driver;`	"We're **keeping track of the car (browser)** so we don't lose it."
`public static IWebDriver GetDriver()`	"This is an **action to start the car and give it to other classes**."
`{`	"We're **starting the steps** inside this action."
Inside `GetDriver()` (Your actual logic here)	"**We start the car (open Chrome)** and **maximize the window**. Then, we **hand the car (driver) back** so other classes can use it."
`}`	"**End of this action's steps.**"

🔄 How It Connects to Everything Else

Let's **zoom out** and see how **DriverPage.cs** plays with the rest of your Selenium framework:

🧩 Part	🛠️ What It Does	🔗 How It Needs DriverPage
DriverPage.cs	**Starts the browser** and **hands the driver (car) to other classes.**	**This is the source – everything starts here.**
BasePage.cs	**Knows how to click buttons, type, and wait.**	**Needs the driver from DriverPage to know which car it's driving.**
VariablePage.cs	**Works on a specific page of a website.**	**Inherits from BasePage, which already has the driver from DriverPage.**
ExampleTest.cs	**Tells the robot what test to run (go here, click this).**	**Starts by calling DriverPage to open the browser first.**

🚗 Simple Visual Flow

Think of the whole thing like this:

1. **DriverPage.cs → Starts the Car (Browser)**
2. **BasePage.cs → Learns How to Drive (Click, Type, Wait)**
3. **VariablePage.cs → Drives to a Specific Place (A Page on a Website)**
4. **ExampleTest.cs → Gives the Robot the Plan (Test Steps to Follow)**

✅ Key Takeaway

Without **DriverPage.cs**, **the car never starts**.
Every test you write **depends on it to open and close the browser**.

Next time you open DriverPage.cs, just remember:
This is the engine. Everything else is the journey.

⏳ WaitUtilities – What Does It Do?

When our robot is driving through a website, it sometimes needs to **wait** for things to appear.
Pages don't always load instantly—buttons, forms, or text might take **a few seconds** to show up.

WaitUtilities.cs is the class that **teaches our robot patience**.

```
using OpenQA.Selenium.Support.UI;
using OpenQA.Selenium;
using SeleniumExtras.WaitHelpers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SeleniumStarterKit.src.Utilities
{
    public class WaitUtilities
    {


        public IWebDriver Driver;
        private WebDriverWait Wait;

        public WaitUtilities(IWebDriver driver)
        {
            this.Driver = driver;
            this.Wait = new WebDriverWait(driver, TimeSpan.FromSeconds(40));
        }


        public void UntilVisible(By locator)
        {
            Wait.Until(ExpectedConditions.ElementIsVisible(locator));
        }


    }
}
```

🧑‍💻 Breaking Down WaitUtilities.cs – What's Happening Inside?

Let's say the robot goes to a website and needs to **click a button**, but the button **takes 3 seconds to show up**. Without waiting, the robot would **try to click it immediately and crash**.

WaitUtilities.cs solves this problem by helping the robot:

- **Wait for buttons or text to appear.**
- **Wait for things to become clickable.**
- **Wait for the page to finish loading.**

🔍 Line-by-Line Breakdown (Example Flow)

Line in the Code	What It Means (Plain English)
`using OpenQA.Selenium;` and others	"These are **toolboxes** we grab to control the browser and set waits."
`namespace SeleniumStarterKit.src.Utilities`	"This is the **folder label** so we know this class lives in Utilities."
`public class WaitUtilities {`	"We're **creating a helper class called WaitUtilities** to help with waiting."
`public static void WaitForElement(IWebDriver driver, By by, int timeoutInSeconds)`	"We're **making an action** that tells the robot: **Wait for this thing to show up.**"
`{`	"We're **starting the steps** inside this action."
Inside the method	"We **wait up to X seconds for the button, text, or element to appear.**"
`}`	"**End of this action's steps.**"

⏳ Simple Visual Flow Example

Imagine the robot doing this during a test:

1. **DriverPage.cs → Starts the car (browser).**
2. **BasePage.cs → Drives the car (click, type, navigate).**
3. **WaitUtilities.cs → Teaches the robot patience (wait if the road is blocked).**
4. **VariablePage.cs → Does page-specific actions.**
5. **ExampleTest.cs → Runs the full test journey.**

✅ Key Takeaway

Without **WaitUtilities.cs**, the robot is **impatient and crashes**.
With it, the robot is **patient and smart**, waiting for things to load before acting.

Next time you open **WaitUtilities.cs**, just remember:
This is the robot’s patience. It waits so your tests don’t fail.

🧠 Base Page – What Does It Do?

Every robot needs **basic driving skills** before it can get fancy.
Our **BasePage.cs** is like the **driving school** for our test robot.

It **teaches every page** how to:

- **Click buttons**.
- **Type in text**.
- **Wait for things to load** (with help from **WaitUtilities.cs**).
- **Use the web browser (driver)** from **DriverPage.cs**.

Any page we create (like **VariablePage.cs**) can **inherit from BasePage.cs** to **automatically know all of these skills**.

```
using OpenQA.Selenium.Interactions;
using OpenQA.Selenium;
using SeleniumStarterKit.src.Utilities;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SeleniumStarterKit.src.Pages
{
    public class BasePage
    {

        public WaitUtilities Wait;
        public IWebDriver Driver;
        public Actions actions;

        public BasePage(IWebDriver driver)
        {
            this.Driver = driver;
            this.Wait = new WaitUtilities(Driver);
            this.actions = new Actions(Driver);

        }

    }
}
```

🧑‍💻 Breaking Down BasePage.cs – What's Happening Inside?

Line in the Code	What It Means (Plain English)
`using OpenQA.Selenium;` and others	"We're **getting our toolboxes** so we can control the browser and wait."
`namespace SeleniumStarterKit.src.Pages`	"This is the **folder label**—this class lives in **Pages**."
`public class BasePage {`	"We're **creating a blueprint called BasePage**—every page can use this."
`protected IWebDriver driver;`	"**The robot's car (browser)**—we need to keep track of it."
`protected WebDriverWait wait;`	"**A built-in wait tool**—helps the robot wait for things to load."
`public BasePage(IWebDriver driver)`	"This is the **constructor**—when we create a BasePage, we **give it a car** (driver)."
`{ this.driver = driver; ... }`	"We're **saving the car** and **creating a wait tool** (e.g., 10 seconds)."
Other methods (e.g., Click, Type, etc.)	"**Robot's driving skills**—Click this, Type that, Wait here."
`}`	"**End of the BasePage blueprint.**"

🔄 How It Connects to Everything Else

🧩 Part	🛠️ What It Does	🔗 How It Needs BasePage
DriverPage.cs	Starts the browser.	BasePage needs it to get the browser (driver).
BasePage.cs	Teaches robots driving skills (click, type, wait).	Uses DriverPage's driver & WaitUtilities to do its work.
VariablePage.cs	Extends BasePage—Focuses on a specific page.	Inherits BasePage's skills like click, type, and wait.
ExampleTest.cs	Runs the test steps.	Uses VariablePage, which relies on BasePage.

🧠 Simple Visual Flow Example

Think of it like this:

1. **DriverPage.cs → Starts the Car (Browser).**
2. **BasePage.cs → Teaches the Robot Driving Skills (Click, Type, Wait).**
3. **VariablePage.cs → Drives to a Specific Place (A Page on a Website).**
4. **WaitUtilities.cs → Helps the Robot Wait if the Road is Blocked.**
5. **ExampleTest.cs → Gives the Robot the Test Plan to Follow.**

📄 Variable Page – What Does It Do?

When you visit a website, **every page is different**.
Each page has **its own buttons, text fields, and things you can click**.

In our automation project, we **represent each page of a website as a C# class**.
That's what **VariablePage.cs** is—it's a **C# class for one specific page**.

If we were testing **eBay's homepage**, we'd have an **eBayHomePage.cs** file.
If we were testing **a login page**, we might call it **LoginPage.cs**.

🧑‍💻 How It Works:

Real-Life Website Page	Our C# Class (Variable Page)
Login Page	`LoginPage.cs`
Product Page	`ProductPage.cs`
Checkout Page	`CheckoutPage.cs`
Search Results Page	`SearchResultsPage.cs`

Each **Variable Page** class **describes the page's buttons, fields, and actions**.

🔄 What's Inside a Variable Page Class?

Think of it like this:

- **Buttons and fields on the website** → We **describe them as variables**.
- **Clicking and typing actions** → We **write methods (actions) to do these things**.

🖥️ Example Full VariablePage.cs Class

```
using OpenQA.Selenium;

namespace SeleniumStarterKit.src.Pages
{
    public class VariablePage : BasePage
    {
        public VariablePage(IWebDriver driver) : base(driver) { }

        // Describe a button on the page
        private By ExampleButton = By.Id("exampleButton");

        // Action: Click the button
        public void ClickExampleButton()
        {
            driver.FindElement(ExampleButton).Click();
        }
    }
}
```

Breaking Down VariablePage.cs – What's Happening Inside?

Line in the Code	What It Means (Plain English)
`using OpenQA.Selenium;` and others	"We're **getting our toolboxes** to control the browser."
`namespace SeleniumStarterKit.src.Pages`	"This is the **folder label**—this class lives in **Pages**."
`public class VariablePage : BasePage {`	"We're **creating a class called VariablePage**, and **it inherits from BasePage** (driving skills)."
`public VariablePage(IWebDriver driver) : base(driver) {` `}`	"**Constructor**—when we create VariablePage, we **pass the car (driver)** to **BasePage** so it knows how to drive."
`private By ExampleButton = By.Id("exampleButton");`	"This is **describing a button on the web page**. We're saying **'Look for the button with the ID exampleButton'**."
`public void ClickExampleButton() {` `driver.FindElement(ExampleButton).Click(); }`	"**Action:** We're **telling the robot how to click the button** we just described."
`}`	"**End of this page's blueprint.**"

🔄 How It Connects to Everything Else

🧩 Part	🛠️ What It Does	🔗 How It Needs VariablePage
DriverPage.cs	Starts the browser.	VariablePage needs it indirectly through BasePage.
BasePage.cs	Teaches robots driving skills (click, type, wait).	VariablePage inherits from BasePage to get these skills.
VariablePage.cs	Handles actions on a specific page (e.g., clicking a button).	Uses BasePage’s driving skills to click, type, and wait.
ExampleTest.cs	Runs the test steps.	Uses VariablePage to tell the robot what to do on the page.

🧠 Simple Visual Flow Example

Let's imagine we're testing a **Login Page**:

1. **DriverPage.cs → Starts the Car (Browser).**
2. **BasePage.cs → Teaches the Robot Driving Skills (Click, Type, Wait).**
3. **VariablePage.cs → Describes the Login Page (Username Field, Login Button) and Knows How to Fill and Submit.**
4. **WaitUtilities.cs → Helps the Robot Wait if the Road is Blocked.**
5. **ExampleTest.cs → Gives the Robot the Full Test Plan.**

✅ Key Takeaway

Each **page on a website** is represented as a **C# class** (like **LoginPage.cs**).
Inside each page class, we:

- **Describe the buttons and fields** on that page.
- **Write actions to click buttons, type text, and do things** on that page.

Without **VariablePage.cs**, the robot would **know how to drive (BasePage), but wouldn’t know what’s on the page**.
With it, the robot **knows what’s on the page and can interact with it**.

Next time you open **VariablePage.cs**, just remember:
This is the map. It describes what’s on a specific page and how to interact with it.

🧪 Tests – What Do They Do?

Everything we've built so far—**DriverPage.cs, BasePage.cs, VariablePage.cs, and WaitUtilities.cs**—is like **preparing the robot**.

But now, it's **time to tell the robot what to do!**

Tests are like **giving the robot step-by-step instructions**:
👉 **"Go to this page, click this button, type in this text, and check if it worked."**

Each test **simulates a user** visiting the website.
The robot **pretends to be a person**—clicking, typing, and checking if everything works.

📝 The Test Name Describes the Action

Naming your tests is **super important** because the **name should tell anyone reading it exactly what the test is doing**.

When you **read the test name**, you should **know the goal without reading the code**.

Bad Name ❌	Good Name ✅
`Test1`	`Login_WhenCredentialsAreValid`
`ClickStuffTest`	`Checkout_WhenItemIsAddedToCart`
`RandomTest`	`SearchResults_AppearWhenSearchingForProduct`

A **Good Test Name** usually answers:

- **What are we doing?**
- **What do we expect to happen?**

Breaking Down a Test Class – What's Happening Inside?

Line in the Code	What It Means (Plain English)
`using NUnit.Framework;` and others	"We're **getting testing tools (like NUnit)** so we can run tests."
`namespace SeleniumStarterKit.src.Tests`	"This is the **folder label**—this class lives in **Tests**."
`public class ExampleTest {`	"We're **creating a test class** called ExampleTest."
`[Test]`	"This is a **test method marker—**it tells the robot **'Run this as a test'**."
`public void ExampleTestMethod()`	"This is the **test name—**it should describe what this test is doing."
`{ /* steps here */ }`	"We **write the steps inside** (e.g., open page, click button)."
`}`	"**End of the class.**"

🖥️ Example Full Test Class

```
using NUnit.Framework;
using OpenQA.Selenium;
using SeleniumStarterKit.src.Utilities;
using SeleniumStarterKit.src.Pages;

namespace SeleniumStarterKit.src.Tests
{
    public class ExampleTest
    {
        private IWebDriver driver;
        private VariablePage variablePage;

        [SetUp]
        public void SetUp()
        {
            driver = DriverPage.GetDriver();
            variablePage = new VariablePage(driver);
        }

        [Test]
        public void Button_ShouldBeClickable_WhenPageLoads()
        {
            variablePage.ClickExampleButton();
            // Add more steps or assertions as needed
        }

        [TearDown]
        public void TearDown()
        {
            driver.Quit();
        }
    }
}
```

🔄 How It Connects to Everything Else

🧩 Part	🛠️ What It Does	🔗 How It Needs Tests
DriverPage.cs	Starts the browser.	Tests need this to open the browser.
BasePage.cs	Teaches robots driving skills (click, type, wait).	Tests rely on pages that inherit BasePage.
VariablePage.cs	Describes a specific page's buttons and actions.	Tests use VariablePage to tell the robot what to click.
WaitUtilities.cs	Helps the robot wait.	Tests benefit when pages wait correctly.
Tests (This)	Tells the robot what to do.	Uses everything above to perform a test scenario.

Simple Visual Flow Example

1. **DriverPage.cs → Starts the Car (Browser).**
2. **BasePage.cs → Teaches the Robot Driving Skills (Click, Type, Wait).**
3. **VariablePage.cs → Maps Out a Specific Page (e.g., buttons, fields).**
4. **WaitUtilities.cs → Helps the Robot Wait if the Road is Blocked.**
5. **Tests → Give the Robot the Plan (Step-by-Step Instructions).**

✅ Key Takeaway

Without **Tests**, the robot would **know how to drive, know the page, know the buttons—but it wouldn't know what journey to take**.
With **Tests**, we **give the robot a plan** and **name it well so others understand what's being tested**.

A Good Test Name = Clear Plan
Every test should **tell a story** with its name:

- **What are we testing?**
- **What do we expect to happen?**

Next time you open **ExampleTest.cs**, just remember:
This is the plan. You're writing the robot's journey, and the test name is its title.

Troubleshooting: 3 Common Selenium Test Problems

Once you have everything set up and your test is running, you might still encounter situations where your test doesn't work as expected. Most of the time, the problem falls into **one of these three categories:**

1. Wrong Selector

Your test is likely trying to interact with the wrong element on the page. This usually happens when:

- The **XPath or CSS Selector is incorrect**.
- The **element's ID or class is dynamic** and changes every time the page loads.
- You're **selecting a similar element** that is not the intended one.

How to fix it:

- Double-check your selector using the **browser's Developer Tools (Inspect Element)**.
- Use unique attributes like `data-testid`, `name`, or static classes.
- Rely less on auto-generated IDs if they are changing.

2. Wait Issue

Your script might be moving too fast, and the element you need hasn't loaded yet. This is a common cause of failures like **ElementNotInteractableException** or **NoSuchElementException**.

Why this happens:

- The **page or element hasn't finished loading** before your test tries to interact with it.
- Some elements **become visible after an animation or AJAX call**.

How to fix it:

- Use **Explicit Waits** with `WebDriverWait` instead of Thread.Sleep().
- Example in C#:`WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(10)); IWebElement element = wait.Until(ExpectedConditions.ElementToBeClickable(By.Id("example")));`
- Learn about **Implicit, Explicit, Fluent, and Static waits**.

3. Wrong Frame (iframe Issue)

Sometimes, your test might not be able to find an element **because it is inside an iframe**.

An **iframe** is like a separate webpage embedded inside another webpage. It looks like this in HTML:

```
<iframe src="https://example.com"></iframe>
```

When you see an iframe, your WebDriver is **only looking at the main page by default**. You need to **switch into the iframe** before interacting with its elements.

How to fix it:

Switch into the iframe before performing actions inside it:

```
driver.SwitchTo().Frame("nameOrIdOfIframe");
```

After you are done with the iframe, **switch back to the main page:**

```
driver.SwitchTo().DefaultContent();
```

Tip: Sometimes iframes don't have names or IDs. You can switch by index or WebElement:

```
driver.SwitchTo().Frame(0); // First iframe on the page
```

Or:

```
IWebElement iframeElement = driver.FindElement(By.TagName("iframe"));
driver.SwitchTo().Frame(iframeElement);
```

Summary

Problem	Cause	Solution
Wrong Selector	Incorrect or dynamic selector	Verify in DevTools, use stable attributes
Wait Issue	Page or element not fully loaded	Use Explicit Waits
Wrong Frame (iframe)	Element inside an iframe	SwitchTo().Frame() before interacting

Understanding these common issues will save you a lot of time and frustration when debugging your Selenium tests!