ERNI

Beyond the Page Object Model: Unlocking the Screenplay Pattern



Content

03	Introduction
06	Key concepts
08	State of the art
	Linear script
	Page Object Model
	Screenplay pattern
38	Conclusion

Introduction

When it comes to automating tests against a human interface, it is common to start by asking about the tools that will be used. It is easy to get lost among the multiple frameworks and tools available in the market: Selenium, Playwright, Cypress, Appium, WinAppDriver, RobotFramework, TestComplete, AppliTools, Tosca and many others.

Teams begin to question the technical knowledge they possess and what technological challenges are covered by each tool, such as element manipulation, integration with other systems, or reporting.

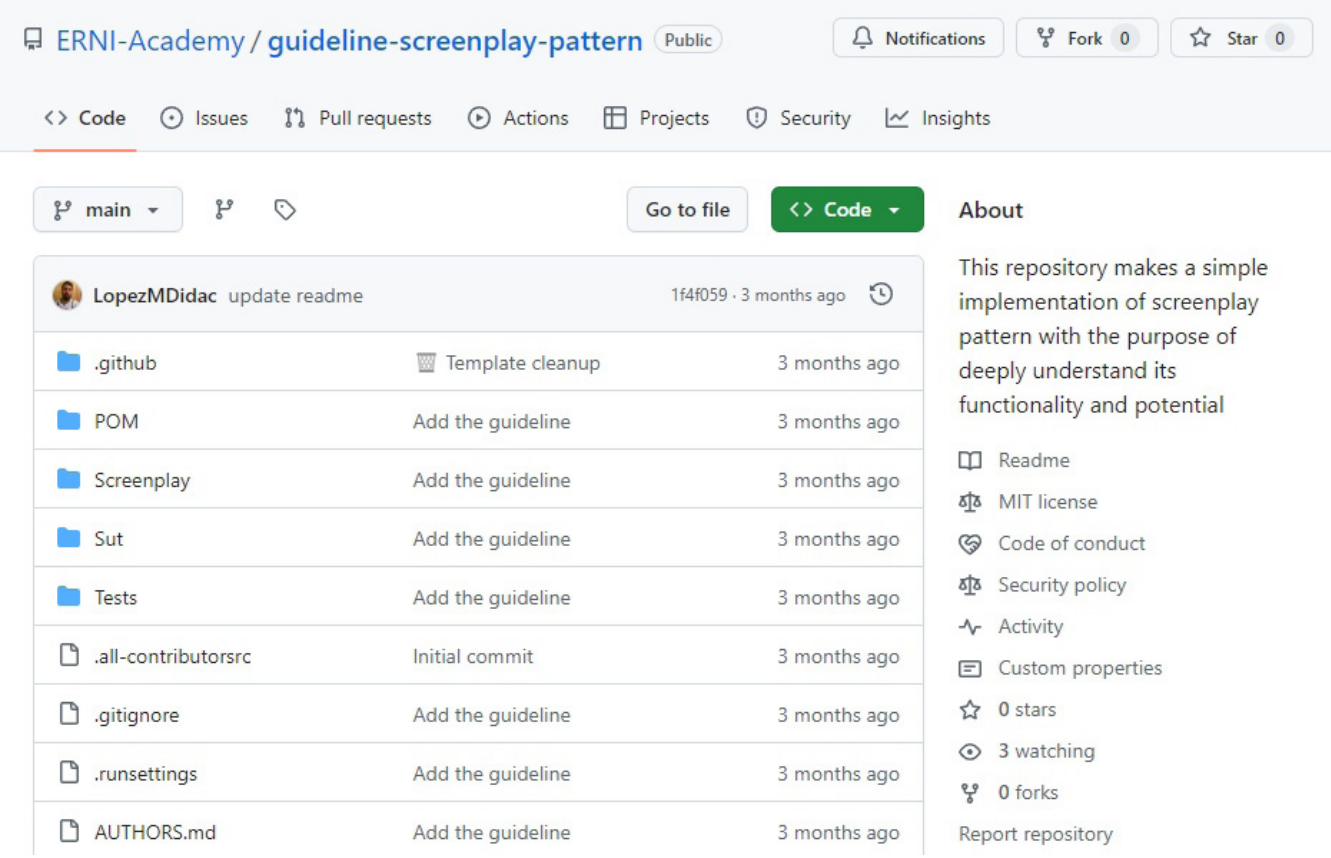
This guides decision making towards a balanced calculation between reducing the learning curve, minimising the cost of licenses (if any), and finding sufficient technological power to meet testing needs.

In this article, however, I will talk about other fundamental aspects in developing a test solution for functional testing. Aspects that are independent of the programming language, tools or frameworks chosen and that answer questions such as: From what point of view are the tests focused? What is the architecture of the automation framework? How are we going to organise our code? What software design patterns are we going to incorporate?

All the code examples shown in this article are written with .NET 8, Playwright and MSTest.

From a functional perspective, this article illustrates the differences between various design patterns, using as an example a test whose objective is to verify that there are job opportunities on the ERNI Spain site, starting from the **homepage https://www.betterask.erni/**

The code used in this article can be found at: **https://github.com/ERNI-Academy/guideline-screenplay-pattern**





Key concepts

It is necessary to start by distinguishing some key concepts that will help in making better decisions when building the test solution.

- **Behaviour:**
  These are the user's objectives. What can or can't the user do with our application?

- **Procedure:**
  Details how the user achieves their objectives through the elements of the application. What are the specific steps to follow to achieve the objective through the application?

- **Tools or frameworks:**
  These are what allow us to interact with the application.

- **Test data:**
  What data the user will work with. This includes both the data the user needs while performing an action and the data the user needs to verify if the system's response is adequat.

- **Structure:**
  Answers how the interface, pages, elements, components, etc. are organised.

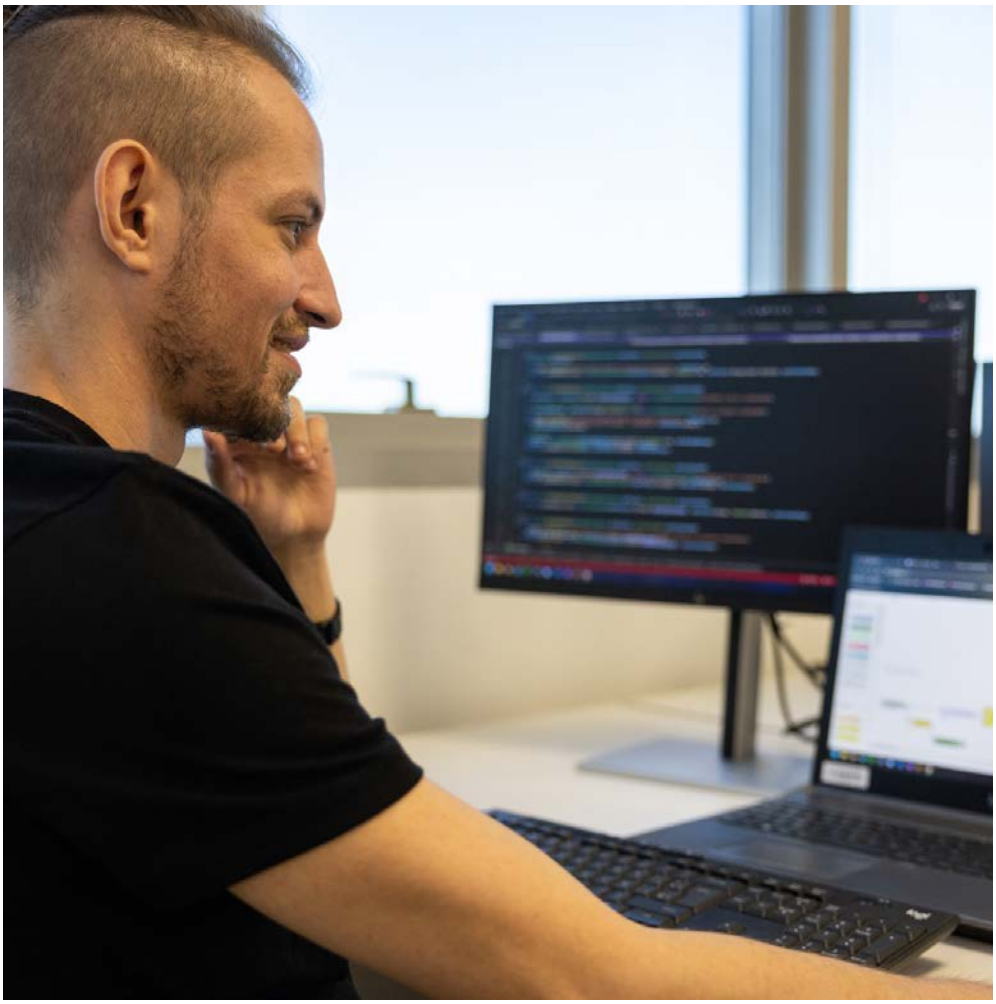
- **SetUp:**
  Everything the infrastructure must do so that the user can execute their behaviours.

- **TearDown:**
  Everything that needs to be done to return the system to a known state, in addition to reporting tasks.



State of the art

When entering the test solution design phase, a common temptation in teams with little experience is to mix all the key concepts within each test without properly segregating responsibilities. This is known as a linear script:



Linear script

In this design, the most important and almost sole element is the test. Each test would initialise the automation tool, navigate through the solution's structure, execute actions by indicating the elements it interacts with, and perform low-level tasks such as capturing evidence.

```
using Microsoft.Playwright;

namespace UnderstandingScreenplay.Tests;

[TestClass]
0 references
public class ERNIWebTest
{
    /// <summary>
    /// Linear Script Test that ensure ERNI Spain site has job opportunities
    /// </summary>
    [TestMethod]
    ● | 0 references
    public async Task  LinearScript()
    {
        var url = "http://betterask.erni";

        var playwright = await Playwright.CreateAsync();
        var browser = await playwright.Chromium.LaunchAsync(new()
        {
            Headless = false,
            SlowMo = 50,
        });
        var page = await browser.NewPageAsync();

        await page.GotoAsync(url);
        await page.GetByRole(AriaRole.Link, new() { Name = "Careers" }).ClickAsync();
        await page.Locator(".elementor-element", new ()
            {
                Has = page.GetByRole(AriaRole.Heading, new() { Name = "ERNI Spain" }),
                HasText = "View job positions"
            }).Last
            .GetByRole(AriaRole.Link).First.ClickAsync();

        await page.GetByRole(AriaRole.Link).Filter(new() { HasText = "View job opportunities" }).First.ClickAsync();
        await Assertions.Expect(page.Locator(".offer-wrapper")).ToBeVisibleAsync();
        var opportunities = await page.Locator(".offer-wrapper").AllAsync();
        Assert.IsTrue(opportunities.Count > 0);
    }

}
```

ADVANTAGES

- Very intuitive for test engineers with little experience.
- Very quick design of test cases.

DISADVANTAGES

- Difficult to read.
- The test performs non-functional tasks, such as initialising the framework.
- Difficult to maintain.
- Poor code reuse.

While it is true that with time and good programming knowledge, this scheme can be greatly improved (auxiliary classes for duplicated code, use of hooks for initialisation and TearDown, etc.), this design is not recommended at all, mainly due to the high maintenance cost, and QA professionals have not used it for a long time.

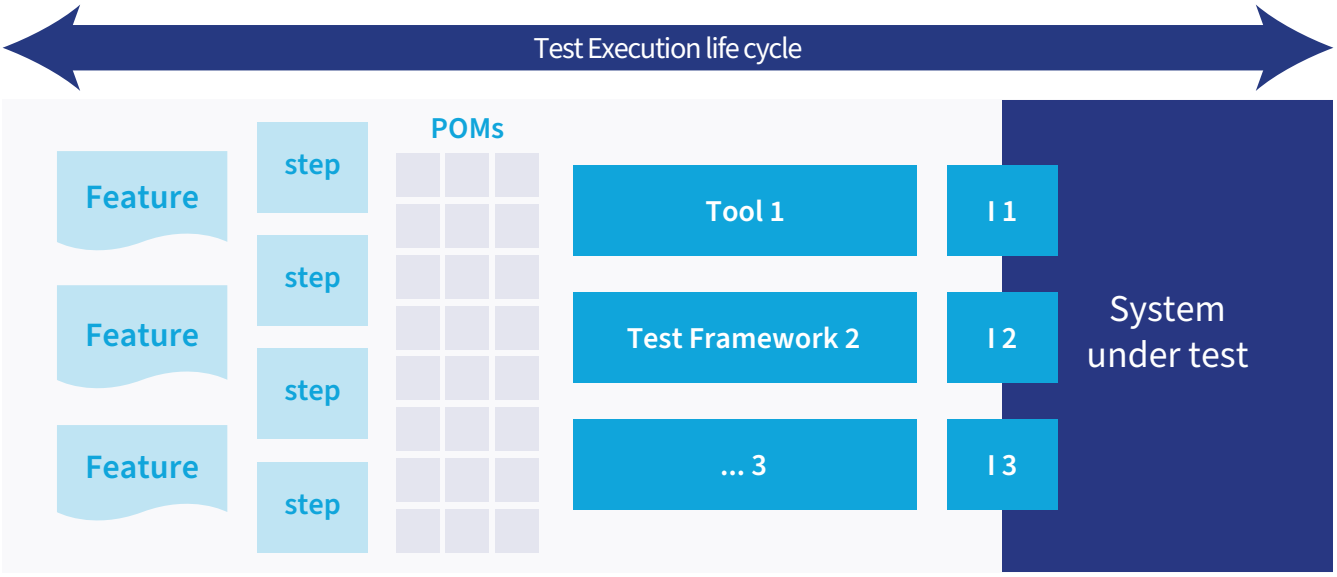
So, what is used then? The most widespread approach within the community is to use the Page Object Model.

Page Object Model (POM)

This is undoubtedly the main design pattern (in all its variants) that is commonly implemented in most test solutions today. It addresses many of the problems that the linear script had, has a large community that knows it, and most test frameworks on the market come with guides on how to implement it.

Let's first look at the theory. In designing a test solution using the Page Object Model, we make a definitive separation between the code that defines our test case and the code that handles interaction with the application.

In this way, the test focuses primarily on the user flow (behaviour or procedure) while the code handles the interaction using specific tools or frameworks.



A class that follows the Page Object Model pattern represents a view of our system under test. Thus, we will have as many POMs as there are views in our application. Each of these classes is responsible for identifying its elements and implementing possible interactions with them.

```
using Microsoft.Playwright;

namespace UnderstandingScreenplay.POM;

9 references
public class BasePage
{
    protected readonly IPage Page;

    4 references
    public BasePage(IPage page)
    {
        Page = page;
    }
}

2 references
public class CareersPage : BasePage
{
    private readonly ILocator _jobPosition;
    private readonly ILocator _siteCards;
    1 reference | ● 0/1 passing
    public CareersPage(IPage page) : base(page)
    {
        _jobPosition = page.GetByRole(AriaRole.Link);
        _siteCards = page.Locator(".elementor-element", new()
        {
            HasText = "View job positions"
        });
    }

    1 reference | ● 0/1 passing
    async public Task GoToJobPositionsFrom(string site)
    {
        await _siteCards
            .Filter(new() { Has = Page.GetByRole(AriaRole.Heading, new() { Name = site }) })
            .Last
            .Locator(_jobPosition).First.ClickAsync();
    }
}
```

The POM is stateless; that is, it does not make internal changes. All calls to this class under the same conditions will produce the same effect. Moreover, it has absolute control over its elements, being the only one that can access them. Lastly, it is important to highlight the existence of a base class that allows the sharing of logic among different pages.

In this way, the logic of interaction with the interface is encapsulated. How would the same test we previously conducted in linear script look now?

```
using Microsoft.Playwright;
using System;
using UnderstandingScreenplay.POM;

namespace UnderstandingScreenplay.Tests;
[TestClass]
0 references
public class ERNIWebTest
{
    /// <summary>
    /// Test using page object model that ensure ERNI Spain site has job opportunities
    /// </summary>
    [TestMethod]
    ● | 0 references
    public async Task PageObjectModel()
    {
        var playwright = await Playwright.CreateAsync();
        var browser = await playwright.Chromium.LaunchAsync(new()
        {
            Headless = false,
            SlowMo = 50,
        });
        var page = await browser.NewPageAsync();

        var homePage = new HomePage(page);
        var careersPage = new CareersPage(page);
        var spainCareersPage = new SpainCareersPage(page);
        var jobOpportunitiesPage = new JobOpportunitiesPage(page);

        await homePage.Navigate();
        await homePage.AccessToCareers();
        await careersPage.GoToJobPositionsFrom("ERNI Spain");
        await spainCareersPage.GoToJobOportunities();
        var opportunities = await jobOpportunitiesPage.GetNumberOfOpportunities();
        Assert.IsTrue(opportunities > 0);

        await browser.DisposeAsync();
        playwright.Dispose();
    }
```

ADVANTAGES

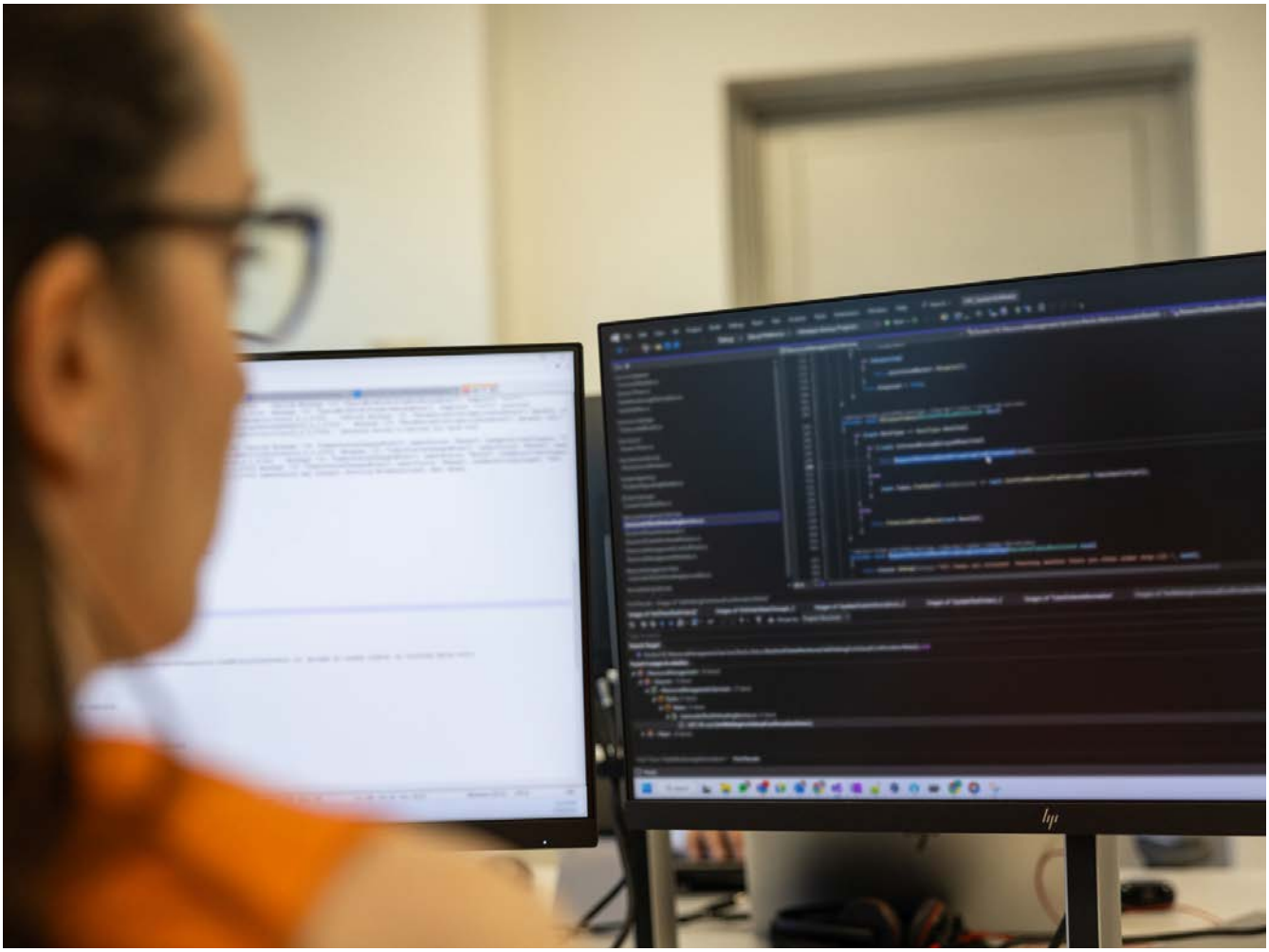
- It is popular in the market and the community.
- Easy to extend for junior/mid-level professionals.
- Good code readability.
- The test is agnostic to the automation tools we use.
- Encourages good practices in code reuse.
- Single point of failure in case of changes in the system under test.

DISADVANTAGES

- Requires advanced knowledge to implement from scratch.
- Tends to become complicated in large applications, especially in modern applications with reusable components and subcomponents.
- Generic methods tend to manage many specific cases.
- Forces us to handle many pages within the test.
- It is an approach closely linked to the structure of the system under test; the same actions are added multiple times for each page.
- POM objects have multiple responsibilities. That makes them difficult to maintain.

As in the case of linear script, the code can be improved: by moving the initialisation and the TearDown of the test towards hooks, making some optimisations so that the navigation method returns the new page, etc. The complexity of our system will lead us to more complex patterns such as the Facade Design Pattern, Lean Page Object Model, Page Component Model and Fluent Design Pattern, which are beyond the scope of this article.

POM is the most used design due to the many advantages it has. We have also seen that there are some disadvantages. But what is beyond that? What could we do differently in designing automation?



Screenplay Pattern

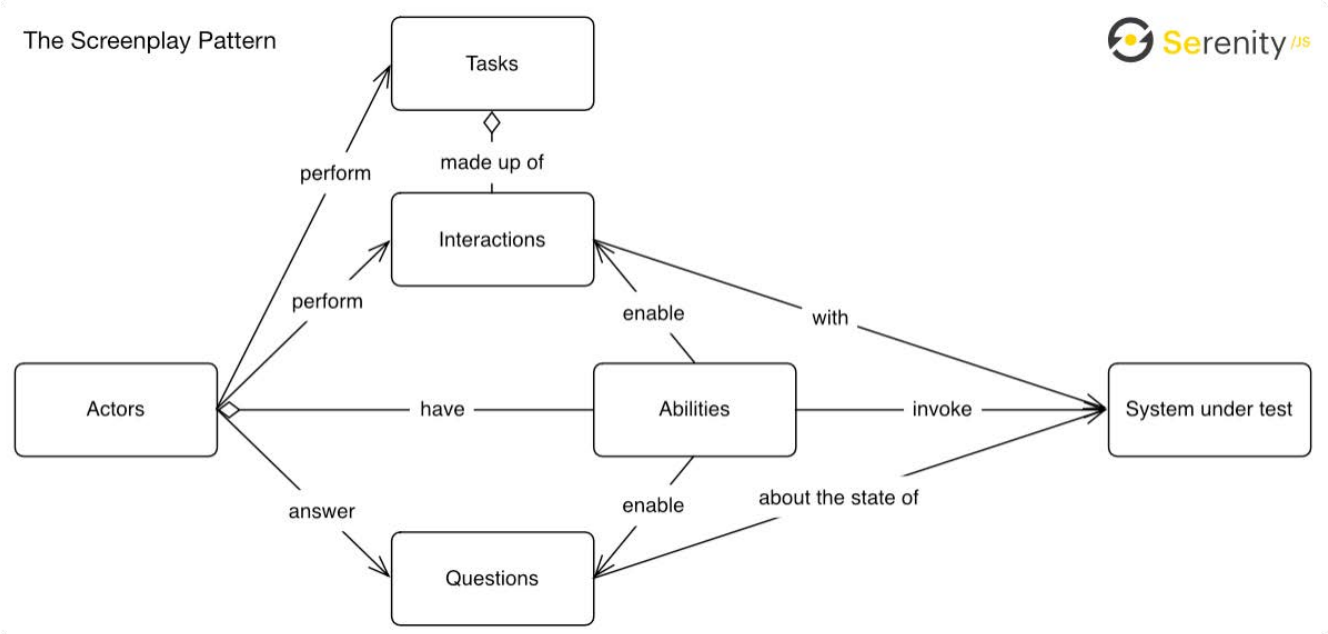
Jan Molak devised the Screenplay Pattern as a way to overcome the limitations of the Page Object Model (POM) in modern applications. Keeping in mind the increasing complexity of these applications and the need for more modular and maintainable tests, Molak sought a solution that offered a clear separation of responsibilities and greater flexibility. Thus, he implemented his pattern in a framework called Serenity BDD. This implementation demonstrates the advantages of the Screenplay Pattern, providing a more robust and adaptable methodology for functional testing of web applications, and stand out for being inspired the structure of a film script.
The key point is that it proposes a paradigm shift. Instead of focusing the design of our test solution from the perspective of the solution being implemented, it places the user at the Centre and focuses entirely on behaviour.

In this way, the test will no longer be about pages doing things but about a user performing tasks, carrying out interactions and posing questions to the system under test, enabled by tools and frameworks.

ADDRESSING THE WEAK POINTS OF POM:

- Screenplay simplifies the management of the system under test for large and complex applications, as it is behaviour-oriented and not particularly affected by design changes.
- It simplifies the number of entities we manage in our test, as it is the actors (usually just one) who interact with the entire application.
- By focusing on the behaviour of the actor as an active subject and not on the pages as a passive subject, it reduces the code derived from invoking the same action from each page.

Let's look at the design of Screenplay as defined by Serenity BDD.



- **Actors:** who represent people and external systems interacting with the system under test
- **Abilities:** that act as thin wrappers around any integration libraries required to interact with the system under test
- **Interections:** which represent the low-level activities an actor can perform using a given interface
- **Tasks:** used to model sequences of activities as meaningful steps of a business workflow in your domain
- **Questions:** used to retrieve information from the system under test and the test execution environment

There are frameworks in the market that implement this design such as Boa-Constrictor for .NET, SerenityBDD for JS/TS, and ScreenPy for Python. The goal of this article is not to explain how to use any of these frameworks, but to understand the pattern itself.

For a productive project it is recommended to use one of these frameworks, which will speed up the development of the test solution in the initial phases.

To delve deeper into the Screenplay Pattern, this article illustrates a lightweight implementation, going into detail on each of the entities presented. This exercise helps to understand the pattern well and lays a good foundation for starting to use it with any of the frameworks.

THE METHOD FOR USING TO THE PATTERN IS DEVIDED OUT INTO THREE PHASES:

- **Define** the abstractions (interfaces) for the different entities.
- **Implement** the entities (base classes + examples).
- **Redefine** the Test Case following Screenplay?

Define the abstractions

ACTOR

The actor represents the user or system that will perform actions against the system under test. A test could potentially require multiple actors with well-defined capabilities. At a low level, it will orchestrate all the action.

Following the pattern, the interface that all actors should follow would be as follows:

```
1  namespace UnderstandingScreenplay.Screenplay.Core.Interfaces;
2
   24 references
3  public interface IActor
4  {
       5 references | 2/2 passing
5      public IActor Performs(IBehaviour tasks);
       21 references | 3/3 passing
6      public IActor Performs<T>(IInteraction<T> interactions);
       7 references | 3/3 passing
7      public IActor HasAbility<T>(IAbility<T> ability);
       6 references | 3/3 passing
8      public IActor Answers<T,R>(IQuestion<T,R> question, out R response);
       5 references | 3/3 passing
9      public R Answers<T,R>(IQuestion<T,R> question);
10 }
```

- Most methods return the actor's own instance, following a fluent design.
- The actor implements a "Performs" method, whose responsibility is to execute the task or interaction received as an argument.
- The actor implements a "HasAbility" method, whose responsibility is to add a new ability to this actor.
- The actor implements an "Answers" method, whose responsibility is to execute the question received as an argument. This method will offer two implementations: one that returns the answer to the question and another that stores this answer in an output method, continuing with the fluent approach.

ABILITY

An ability is a capability that our actor (already introduced) possesses. The same actor can have multiple skills. Skills allow us to interact with the system under test. If our actor accesses the system via a web application, the skill needed is "web browsing". If the actor also needs to interact with a physical device, we could also provide the skill of "robotic arm control".

At a low level, a skill encapsulates a testing tool or framework and will take care of everything necessary to make it available to the actor.

```
namespace UnderstandingScreenplay.Screenplay.Core.Interfaces;

3 references
public interface IAbility<T>
{
    3 references
    public T GetEnabler();
}
```

- The interface is of type T, which is a generic type and refers to the engine of the tool or framework used that will enable interactions and questions.
- Through the “GetEnabler” method, a skill offers its Enabler for use. (An Enabler can be a WebDriver in Selenium, a Page in Playwright, etc.).

INTERACTION

An interaction represents a unitary and generic action on the system under test. In the example of a web page, some cases would be: Click, Fill with text, Select. These interaction-type entities implement the procedural part.

At any time, an actor with a skill such as “Web Browsing” can execute an interaction that uses the Enabler of this skill, such as navigate.

```
namespace UnderstandingScreenplay.Screenplay.Core.Interfaces;

7 references
public interface IInteraction<T>
{
    5 references
    public IInteraction<T> PerformedBy(IActor actor);
    2 references
    public IInteraction<T> EnabledBy(T enabler);
}
```

- This Interaction has a generic type T, which is the Enabler needed to execute this action and will be provided by the skills of the actor attempting to execute it.
- The “PerformedBy” method will execute the interaction’s logic.
- The “EnabledBy” method will serve to inject the Enabler into the interaction at the time of execution.

TASK (BEHAVIOUR)

Tasks serve to group interactions. In this way, a separation between procedure (interaction) and behaviour is achieved. In this implementation, Task has been renamed to Behaviour to avoid namespace collisions in .NET.

At a low level, a task simply makes calls to multiple interactions.

```
1  namespace UnderstandingScreenplay.Screenplay.Core.Interfaces;
2
   5 references
3  public interface IBehaviour
4  {
       4 references
5      public IBehaviour PerformedAs(IActor actor);
6  }
```

- The PerformedAs method will execute the block of calls to the different interactions.

QUESTION

Questions are entities like interactions, with the difference that they allow the actor to retrieve information about the state of the system under test. This is a fundamental activity for performing verifications (the main goal of test automation).

```
1  namespace UnderstandingScreenplay.Screenplay.Core.Interfaces;
2
   9 references
3  public interface IQuestion<T,R>
4  {
       2 references
5      public IQuestion<T, R> EnabledBy(T enabler);
       4 references
6      public IQuestion<T, R> AnsweredTo(IActor actor, out R response);
7  }
```

- This question has a generic type T, R, where T is the Enabler needed to execute this question and R is the type of the response.
- The “AnsweredTo” method will execute the logic of the question.
- The “EnabledBy” method will serve to inject the Enabler into the question at the time of execution.

SYSTEM UNDER TEST

The last part we are missing is the system under test. Here the pattern blurs a bit. Automation needs to know how to find the elements and components that make up the application we are testing.

Locating elements is closely linked to the automation engine being used. This example is using playwright, and it has been decided to use an intermediate class for localisation.

```
1   using Microsoft.Playwright;
2
3   namespace UnderstandingScreenplay.Sut;
4
    41 references
5   public class Locator
6   {
7       private Func<IPage, ILocator> _getter;
        10 references
8       public ILocator this[IPage page]
9       {
10          get => _getter(page);
11      }
12
        13 references
13      public Locator(Func<IPage, ILocator> getter)
14      {
15          _getter = getter;
16      }
```

- The Locator class is constructed using a function that implements the necessary logic to acquire an ILocator from Playwright from an IPage of Playwright.
- Invoking the this [IPage page] property will execute the function and retrieve the element.
- An element is identified as Locator element = page => page.Locator(“Element property”).
- Locator[Enabler] allows access to an element from an interaction or question.

This is a generic class that helps us define and access elements in a simple way, integrated with the Playwright automation engine.

Implement the entities

ACTOR

At this point, we have individually understood what each entity does and have created the necessary interfaces to carry out the pattern, but we need to dig a bit deeper to understand how these entities relate to each other.

This implementation makes use of inheritance; the base class (parent) will implement the general flow of the pattern, and the child classes will only address their specific logic.

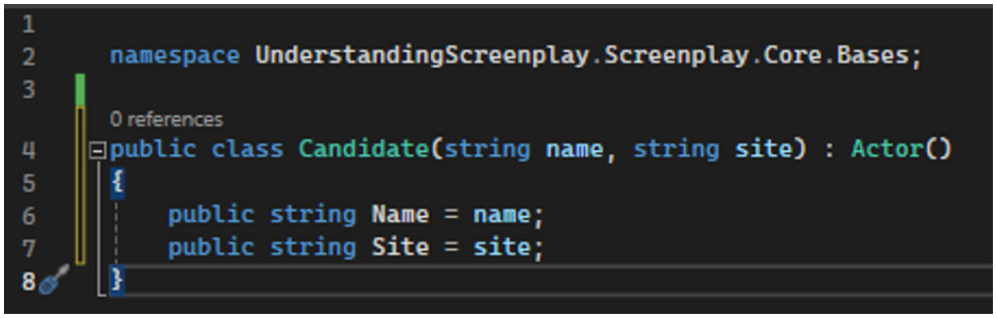
```
using UnderstandingScreenplay.Screenplay.Core.Interfaces;

namespace UnderstandingScreenplay.Screenplay.Core.Bases;
3 references
public abstract class Actor : IActor
{
    protected IEnumerable<dynamic> Abilities;
    1 reference
    public Actor()
    {
        Abilities = [];
    }
    6 references | 3/3 passing
    public virtual IActor Answers<T,R>(IQuestion<T,R> question, out R response)
    {
        response = Answers(question);
        return this;
    }
    5 references | 3/3 passing
    public virtual R Answers<T, R>(IQuestion<T,R> question)
    {
        question.EnabledBy(Abilities.Select(a => a.GetEnabler()).First(e => e is T)).AnsweredTo(this, out R response);
        return response;
    }
    7 references | 3/3 passing
    public virtual IActor HasAbility<T>(IAbility<T> ability)
    {
        Abilities = [.. Abilities, ability];
        return this;
    }
    5 references | 2/2 passing
    public virtual IActor Performs(IBehaviour task)
    {
        task.PerformedAs(this);
        return this;
    }
    21 references | 3/3 passing
    public virtual IActor Performs<T>(IInteraction<T> interaction)
    {
        interaction.EnabledBy(Abilities.Select(a => a.GetEnabler()).First(e => e is T)).PerformedBy(this);
        return this;
    }
}
```

- The Base Actor class implements the IActor interface.
- Actor maintains a dynamic-type array to store its abilities (implemented in this way for simplicity of the example).

- The Answers method is implemented as a call to the AnsweredTo of the IQuestion received as a parameter. Before making the call, it is enabled with the first Enabler available among the actor's abilities that matches the type of the received IQuestion.
- The HasAbility method simply adds the new ability to the abilities array.
- The Performs method that receives an interaction is implemented as a call to the PerformedBy of the IInteraction. Before making the call, it is enabled with the first Enabler available among the actor's abilities that matches the type of the IInteraction.
- The Performs method that receives an IBehaviour simply calls the PerformedAs method of the IBehaviour

Although the Actor class fully implements the interface, we should not use it directly. We could still create specific actors with concrete data that help us understand who this actor is and what they want to do.



A potentially good idea would be to populate these attributes from files with test data to achieve maximum flexibility in testing. Another possibility for expansion is to have actor profiles (implemented via interfaces) to limit what types of tasks they can or cannot perform.

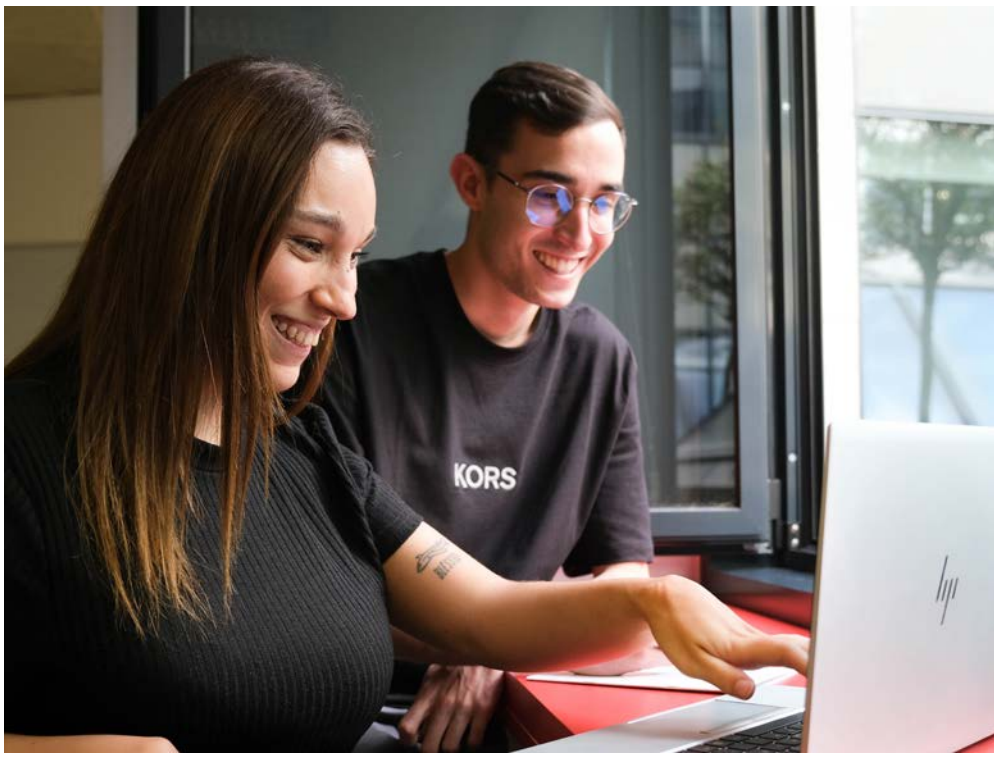
ABILITY

```
using UnderstandingScreenplay.Screenplay.Core.Interfaces;

namespace UnderstandingScreenplay.Screenplay.Core.Bases;
2 references
public abstract class Ability<T> : IAbility<T>
{
    protected T Enabler;
    1 reference
    public virtual T GetEnabler()
    {
        return Enabler;
    }
}
```

- The base class of the ability simply returns the Enabler.

With this base class, each of our abilities can focus on its area of responsibility, which is managing the tool or framework of its type.



```
using Microsoft.Playwright;
using UnderstandingScreenplay.Screenplay.Core.Bases;

namespace UnderstandingScreenplay.Screenplay.Abilities;

9 references
public class BrowseTheWeb : Ability<IPage>
{
    private IPlaywright _playwright;
    private IBrowser _browser;
    2 references
    private BrowseTheWeb(IPage page) { Enabler = page; }
    3 references | 3/3 passing
    public static BrowseTheWeb With(IPage page)
    {
        return new BrowseTheWeb(page);
    }
    0 references
    public static BrowseTheWeb WithNewSession()
    {
        var playwright = Playwright.CreateAsync().Result;
        var browser = playwright.Chromium.LaunchAsync(new()
        {
            Headless = false,
            SlowMo = 50,
        }).Result;

        var page = browser.NewPageAsync().Result;
        var ability =  new BrowseTheWeb(page);
        ability._playwright = playwright;
        ability._browser = browser;
        return ability;
    }
    0 references
    public async Task<BrowseTheWeb> Dispose()
    {
        await _browser.DisposeAsync();
        _playwright.Dispose();
        return this;
    }
}
```

- BrowseTheWeb is an ability of type IPage (Playwright).
- In its private constructor, it initialises the Enabler property, which
- This class accepts two initialisations: With and WithNewSession. In the first case, we accept a page that has been initialised elsewhere, and in the second, it is the ability itself that initialises it.

- Symmetrically, the ability also provides a TearDown method.

This ability has the potential to grow; with just a few lines of code, we have achieved different initialisation approaches. Deciding which fits best with our needs will depend on the architecture of the entire test solution.

Even so, the initialisation process can be improved, for example, by reading properties from configuration files and adding flexibility. This implementation, as a starting point and for the scope of this article, is sufficient.

INTERACTION

```
using UnderstandingScreenplay.Screenplay.Core.Interfaces;

namespace UnderstandingScreenplay.Screenplay.Core.Bases;

4 references
public abstract class Interaction<T> : IInteraction<T>
{
    protected T? Enabler;

    2 references
    public virtual IInteraction<T> EnabledBy(T enabler)
    {
        Enabler = enabler;
        return this;
    }
    5 references
    public abstract IInteraction<T> PerformedBy(IActor actor);
}
```

- The base Interaction class implements the EnabledBy by storing the Enabler.
- The PerformedBy method is declared as abstract, thus forcing interactions to implement their logic.

With this base class, any interaction can focus on the logic it must implement. Let's look at the Enter interaction as an example, which is used to enter text into an element.

```
using Microsoft.Playwright;
using UnderstandingScreenplay.Screenplay.Core.Bases;
using UnderstandingScreenplay.Screenplay.Core.Interfaces;
using UnderstandingScreenplay.Sut;

namespace UnderstandingScreenplay.Screenplay.Interactions;

8 references
public class Enter : Interaction<IPage>
{
    private string? _value;
    private Locator? _locator;

    4 references | 1/1 passing
    public static Enter TheValue(string value)
    {
        return new Enter() { _value = value };
    }

    4 references | 1/1 passing
    public Enter Into(Locator locator)
    {
        _locator = locator;
        return this;
    }

    2 references
    public override Enter PerformedBy(IActor _)
    {
        _locator![Enabler!].FillAsync(_value!).Wait();
        return this;
    }
}
```

- The Enter interaction inherits from interaction with type IPage, with IPage being the Enabler of the ability.
- Following a fluent design, the interaction stores the value and the fields entered through TheValue and Into to have them available during execution.
- The PerformedBy method overrides the abstract method and requires the implementation of the logic.

This is a simple interaction that could be improved by adding field-clearing logic, ensuring all necessary information is available before execution, and offering customised errors, among other enhancements.

TASK (BEHAVIOUR)

```
using UnderstandingScreenplay.Screenplay.Core.Interfaces;

namespace UnderstandingScreenplay.Screenplay.Core.Bases;

3 references
public abstract class Behaviour : IBehaviour
{
    5 references
    public abstract IBehaviour PerformedAs(IActor actor);
}
```

- The base Behaviour class (for tasks) only has the abstract PerformedAs method, which obliges the tasks that implement it to implement the logic there.

The logic of a task will be a call to the necessary interactions to execute a set of actions that make sense for this particular domain. Thus, while interactions are atomic and reusable even across multiple test solutions, Behaviours are more tied to behaviour and linked to a specific system under test.

Following the example test, a possible task would be to go from the Careers section to the job opportunities of a specific site.

```
using UnderstandingScreenplay.Screenplay.Core.Bases;
using UnderstandingScreenplay.Screenplay.Core.Interfaces;
using UnderstandingScreenplay.Screenplay.Interactions;
using UnderstandingScreenplay.Sut.Locators;

namespace UnderstandingScreenplay.Screenplay.Behaviours;

4 references
public class NavigateToSiteCareers : Behaviour
{
    private int _siteIndex;

    1 reference | 1/1 passing
    public static NavigateToSiteCareers WithIndex(int site)
    {
        return new NavigateToSiteCareers() { _siteIndex = site };
    }

    3 references
    public override NavigateToSiteCareers PerformedAs(IActor actor)
    {
        actor
            .Performs(Click.On(ERNIWeb.JobPositionBtn).WithPosition(_siteIndex))
            .Performs(Click.On(ERNIWeb.JobOpportunities));
        return this;
    }
```

- NavigateToSiteCareers includes the Click interaction on the site button of a specific index and the second step in Job Opportunities.
- It uses the actor to execute each of the actions individually.

We can see how a behaviour groups multiple calls to interactions and executes them individually. In my view, the ideal for this task would be for it to be invoked with the name of the site it wants to interact with, and for the task itself to figure out how to find the correct button.

However, the pattern does not allow this, since a task is composed of interactions and not interactions and questions. Nonetheless, I find it interesting that a task could ask things about the system with some limitations, such as not making assertions and not including conditional logic.

In any case, these are some ideas to consider beyond the pattern:

QUESTION

```
1  using UnderstandingScreenplay.Screenplay.Core.Interfaces;
2
3  namespace UnderstandingScreenplay.Screenplay.Core.Bases;
4
   4 references
5  public abstract class Question<T,R> : IQuestion<T,R>
6  {
7      protected T? Enabler;
       5 references
8      public abstract IQuestion<T, R> AnsweredTo(IActor actor, out R response);
       2 references
9      public virtual IQuestion<T, R> EnabledBy(T enabler)
10     {
11         Enabler = enabler;
12         return this;
13     }
```

- The base class question implements the EnabledBy by storing the Enabler.
- The method AnsweredTo is declared as abstract, thus forcing the questions to implement their logic.

With this base class, any question only needs to focus on the logic related to gathering the required information from the system under test.

As an example, let's look at the Question Count, which is responsible for counting the number of elements present in the system under test, given specific conditions.

```
using Microsoft.Playwright;
using UnderstandingScreenplay.Screenplay.Core.Bases;
using UnderstandingScreenplay.Screenplay.Core.Interfaces;
using UnderstandingScreenplay.Sut;

namespace UnderstandingScreenplay.Screenplay.Questions;
6 references
public class Count : Question<IPage, int>
{
    private Locator? _locator;

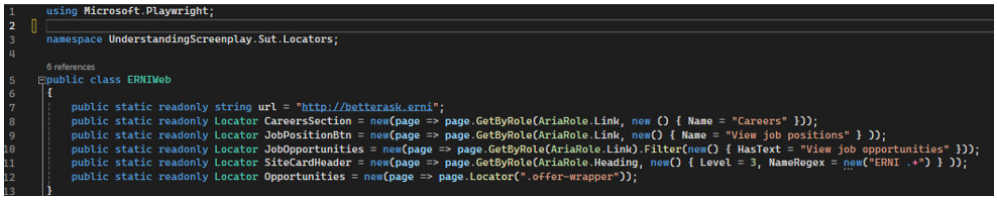
    3 references | 3/3 passing
    public static Count Of(Locator locator)
    {
        return new Count() { _locator = locator };
    }

    2 references
    public override Count AnsweredTo(IActor _, out int response)
    {
        try
        {
            Assertions.Expect(_locator![Enabler!].First).ToBeVisibleAsync().Wait();
            response = _locator[Enabler!].CountAsync().Result;
        }
        catch (Exception e)
        {
            response = 0;
        }
        return this;
    }
```

- This question uses IPage as the Enabler and returns an int.
- With the method Of, the question is instantiated using the Locator to count.
- The method AnsweredTo implements the question, which waits for these elements to exist and counts them; in case of an error, the number of elements is 0.

This question can potentially add more methods with filters or conditions that offer more details about the elements that need to be counted. The objective of this article is to illustrate the use and implementation and not to provide a complete implementation.

SYSTEM UNDER TEST



```
using Microsoft.Playwright;

namespace UnderstandingScreenplay.Sut.Locators;

0 references
public class ERNIWeb
{
    public static readonly string url = "http://betterask.erni";
    public static readonly Locator CareersSection = new(page => page.GetByRole(AriaRole.Link, new () { Name = "Careers" }));
    public static readonly Locator JobPositionBtn = new(page => page.GetByRole(AriaRole.Link, new() { Name = "View job positions" } ));
    public static readonly Locator JobOpportunities = new(page => page.GetByRole(AriaRole.Link).Filter(new() { HasText = "View job opportunities" }));
    public static readonly Locator SiteCardHeader = new(page => page.GetByRole(AriaRole.Heading, new() { Level = 3, NameRegex = new("ERNI .+") } ));
    public static readonly Locator Opportunities = new(page => page.Locator(".offer-wrapper"));
}
```

- The class ERNIWeb models the elements with which we need to interact.
- Each element is static and read-only.
- We use the Locator presented in the previous section to define the search strategies.

This approach is specific to this example and has been implemented this way for simplicity. In this example, all elements are part of the same class, which presents scaling issues. In larger solutions, it would be sensible to separate the elements into units that make sense for the system being tested, which do not necessarily have to be pages.

Redefine the Test Case following Screenplay

Once all the entities have been implemented, let's see how the test would look.

```
using Microsoft.Playwright;
using UnderstandingScreenplay.POM;
using UnderstandingScreenplay.Screenplay.Abilities;
using UnderstandingScreenplay.Screenplay.Behaviours;
using UnderstandingScreenplay.Screenplay.Core.Bases;
using UnderstandingScreenplay.Screenplay.Interactions;
using UnderstandingScreenplay.Screenplay.Questions;
using UnderstandingScreenplay.Sut.Locators;

namespace UnderstandingScreenplay.Tests;
[TestClass]
0 references
public class ERNIWebTest
{

    /// <summary>
    /// Test using Screenplay pattern that ensure ERNI Spain site has job opportunities
    /// </summary>
    [TestMethod]
    0 references
    public async Task ScreenPlay()
    {
        var surfTheWeb = BrowseTheWeb.WithNewSession();

        var candidate = new Candidate("Diego de la Vega", "ERNI Spain");

        candidate.HasAbility(surfTheWeb)
            .Performs(Navigate.To(ERNIWeb.url))
            .Performs(Click.On(ERNIWeb.CareersSection))
            .Answers(Location.Of(ERNIWeb.SiteCardHeader).WithText(candidate.Site), out var position)
            .Performs(NavigateToSiteCareers.WithIndex(position))
            .Answers(Count.Of(ERNIWeb.Opportunities), out var opportunities);

        Assert.IsTrue(opportunities > 0);

        surfTheWeb.Dispose();
    }
```

- The test creates an ability to navigate the web.
- The test creates a candidate actor according to the type of action they will perform with relevant information.
- The actor includes the necessary abilities to carry out their test - in this case, navigating the web.
- The actor performs the tasks and interactions using the elements of the system under test and requests relevant information from it.
- The test performs the TearDown and final actions.

By following the Screenplay Pattern and a fluent design, we achieve a behaviour-focused test that truly describes the behaviour an actor should exhibit to perform the test.





Conclusion

The Screenplay Pattern is an interesting pattern to consider when starting a framework from scratch. We can see how the pattern correctly encapsulates the responsibility of each entity, allowing the ability to manage the tools, letting the interactions and questions focus exclusively on their logic, and enabling the actors to be the central piece that orchestrates the test. Less responsibility means entities that are easier to maintain and quicker to develop.

Modelling the system under test as a collection of static properties separates the test framework from the page structure of the application, making the test framework more resilient to design changes.

Understanding the pattern is complicated; there are many entities at play, and it is challenging to use it properly. Additionally, it is necessary to resolve how to manage variations: is a click with an offset a second click interaction? Or does the same interaction handle all its cases? Poor implementation can lead to duplicated, unused, or overly complex interactions. I do not recommend implementing this pattern for teams with little maturity in automation.

Compared to POM, Screenplay reduces the amount of code. We no longer have to include the “Click” call on every clickable element of all pages; instead, we have a “Click” interaction that we use against clickable elements.

In POM we would say:

- The shelf holds a jar of jam in plain sight.
- The jar of jam is added to the shopping cart.
- The shopping cart checks that it has 1 product inside.

In Screenplay we say:

- The shopper sees a jar of jam on the shelf.
- The shopper adds the jar of jam to the shopping cart.

	Linear script	POM	Screenplay
Complexity	Minimum	Medium	High
Maintainability	High	Medium/Low	Low
Orientation	Procedure	System under Test	Behaviour
Design Speed	High	Medium	Medium/High
Community Popularity	Deprecated	de facto standard	Incipient

References:

Page Object Model: https://martinfowler.com/bliki/PageObject.html

Screenplay Pattern: https://serenity-js.org/handbook/design/screenplay-pattern/

Playwright: https://playwright.dev/dotnet/

Contacto

ERNI Schweiz AG
Löwenstrasse 11
8001 Zürich

Phone: +41 58 268 12 00
Email: info@erni.ch

www.betterask.erni
Germany I Philippines I Romania
Singapore I Slovakia I Spain
Switzerland I USA

better ask ERNI