



Better Test Design:

A Journey to Succeed with UI Automation



TestComplete



Scale your UI Test Automation
Strategy with TestLeft



LEARN MORE ABOUT TESTCOMPLETE

UI automation often starts on a great note. You buy a test automation tool and start writing automated tests. But then over time, as you begin to scale your testing efforts — from ten tests to hundreds or even thousands — maintenance of UI tests becomes difficult.

The brittleness of UI tests can be caused by many factors, ranging from test design, test data or even environment configurations. Test design in particular can often cause serious issues as your testing progresses. For example, let's assume you write a test consisting of actions that drive directly at the HTML layer. The challenges with such a process are that even the smallest changes to the UI would cause these tests to break. Tests thereby are brittle. And as a result, the testing process becomes less scalable, stable, and reusable.

Another challenge affecting the test design process is that UI tests are often tests dependent on each other. This lack of independence in turn causes challenges primarily because when one test fails, the tests that are dependent on that component can break. We created this eBook to focus upon how as a tester you can create a testing framework that scales as well as is easier to maintain. This eBook is structured into five parts:

1. Where does UI automation typically go wrong?
2. Why UI tests are essential part of your testing strategy?
3. An example of a badly written test
4. Design frameworks to overcome the UI challenges
5. Application of these frameworks with real world examples

So let's start with the first part, where does UI automation go wrong. Let's explore this with an example:

```
private void Login(IWebBrowser browser, string username, string password)
{
    // Find the user name text box
    var UserNameEdit = browser.Find<ITextEdit>(new WebElementPattern()
    {
        ObjectIdentifier = "MainContent_username"
    }, 5);

    // Set the user name text box
    UserNameEdit.SetText(username);

    // Find the password text box
    var PasswordEdit = browser.Find<ITextEdit>(new WebElementPattern()
    {
        ObjectIdentifier = "MainContent_password"
    }, 5);

    // Set the password text box
    PasswordEdit.SetText(password);

    // Find the login button
    var LoginButton = browser.Find<IButton>(new WebElementPattern()
    {
        ObjectIdentifier = "MainContent_login_button"
    }, 5);

    // click the login button
    LoginButton.Click();
}
```

From the first look, this seems to be very well written test. We are performing 6 different actions here.

- Finding the username text box
- Setting the user name text box
- Finding the password text box
- Setting the password text box
- Finding the login button
- Clicking the login button

Even with just six actions being performed, there are designing challenges with tests. And the numbers of tests increase, these challenges will just increase exponentially. Let's explore these challenges:

First and foremost there is a lot of duplication taking place. Take for instance, `browser.Find` is being repeated three times. No if sometime in the future we want to change `browser.find`, we will have to make changes at three places, even with such a small code. This duplication in turn poses maintainability challenges in the future.

Secondary, since this is procedural code readability can be a pain.

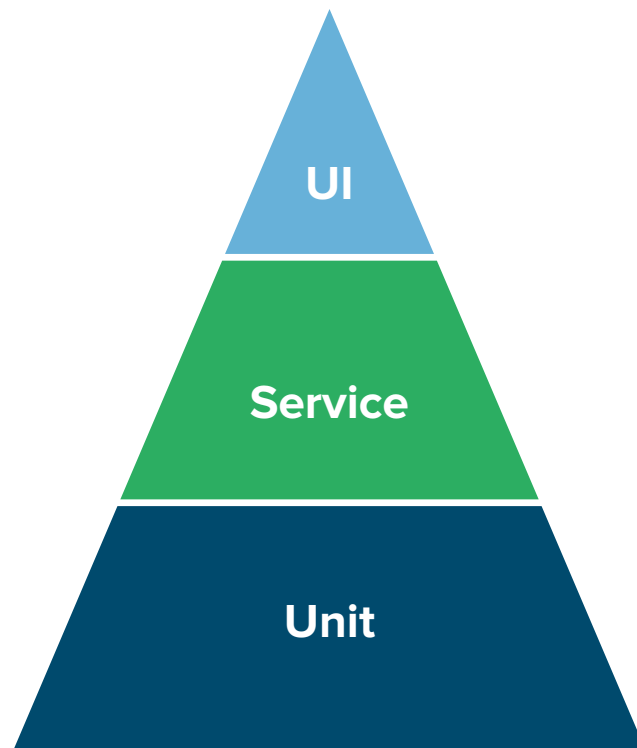
Thirdly, reusability of this code for different test cases can be near to impossible as there are no modularity practices being followed.

Lastly, the use of magic strings within the code makes maintainability a hassle as well. Primarily because, if my identified for login button changes from `"MainContent_login_button"` to `"Content_login_button"`, we will have to make changes at numerous place in the test.



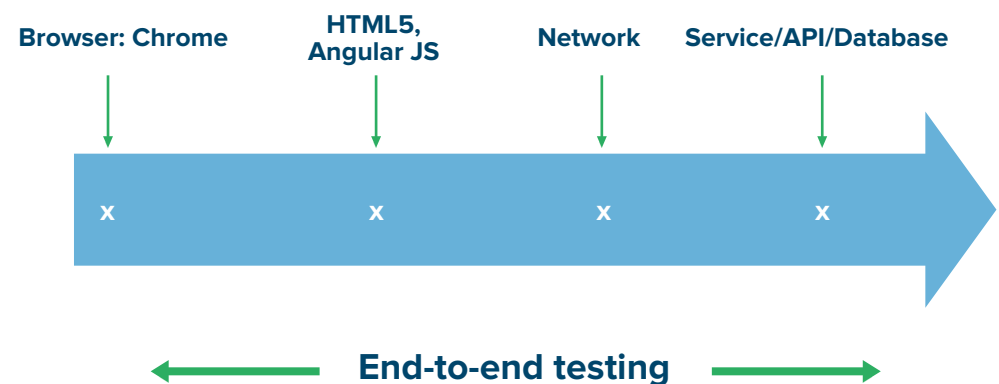
Test pyramid strategy can come in handy, but..

Often when solving brittle GUI tests, test pyramid automation strategy is presented as a solution. The test automation pyramid strategy calls for automating tests at three different levels. Unit testing represents the base of test automation pyramid and the largest part. Next come, service layer or API testing. And finally since GUI tests sit at the top as we want to do least of them as possible. The pyramid looks something like this:



While such test automation could be useful, one thing to keep in mind while following test automation pyramid strategy is that the extensive feedback one receives from GUI tests can't be achieved from Service level or Unit level tests. Take the following as an example.

Here I am driving to a test web application such as Amazon. And when I do that, I get an end-to-end feedback. This is primarily because UI tests touch different parts of the application that's being tested. For instance, the incase of Amazon the UI tests start the browser level (Chrome, Mozilla, Edge, etc.), then goes on to touch framework (AngularJS/HTML5), network, and finally the service or database. This looks something like this:



Over the next part of the presentation, we will discuss three different models in which you can use to start working on an automation framework. All these three models are built upon the concept of a page object. Hence, let's dig into this in more detail.

Introducing a Page Object Model

Building a robust UI test automation strategy begins with following a page object model for your test cases. Within a page object model, you'll be creating a class for every page within your application. Or in other words, each webpage will be represented as class.

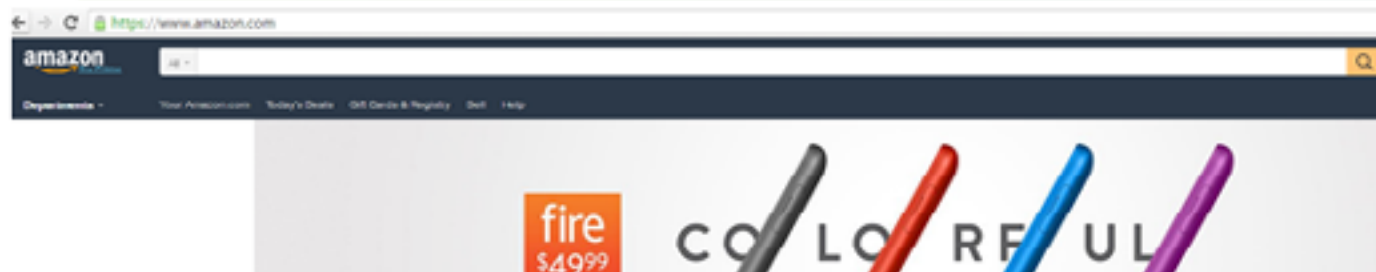
Let's take the following as an example. Here I am ensuring a book is added to cart and Proceeds to checkout button works as expected on the Confirmation Page. This essentially means I will have to navigate through four steps in order to reach the Confirmation page. These are

- **Homepage:** Where the user first lands and begins a search.
- **Search Result Page:** The page that displays the available books, based on the search.
- **Product Details Page:** After a user selects a book and wants to read more about it and see pricing options.

- **Confirmation Page:** The final page a user lands after making a purchase.



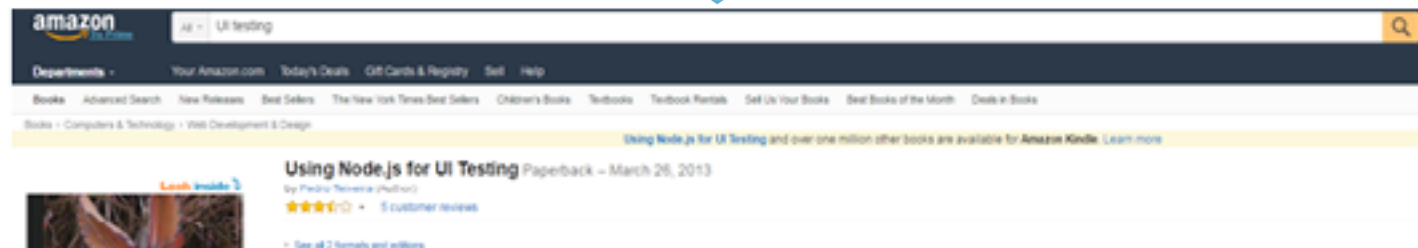
From a pictorial standpoint, the process looks as follows:



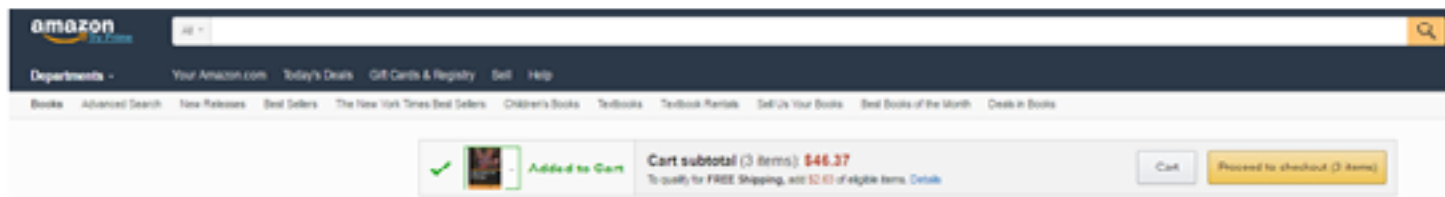
Home Page of Amazon



Result Search Page



Product Details Page



Confirmation Page

In this case, each webpage of Amazon would be represented as one particular class. For example home page of Amazon is one class, followed by result search page, product details page, and finally a confirmation page as another three sets of page classes.

Once page objects for each page have been created, the three models presented below just build upon the page object. And as we go from model 1 to model 3, the abstraction increases. And since abstraction increases, maintainability of test cases becomes much easier. Let's start by looking at abstraction model1.



Abstraction Model #1: Abstracting the Control Identifiers Only

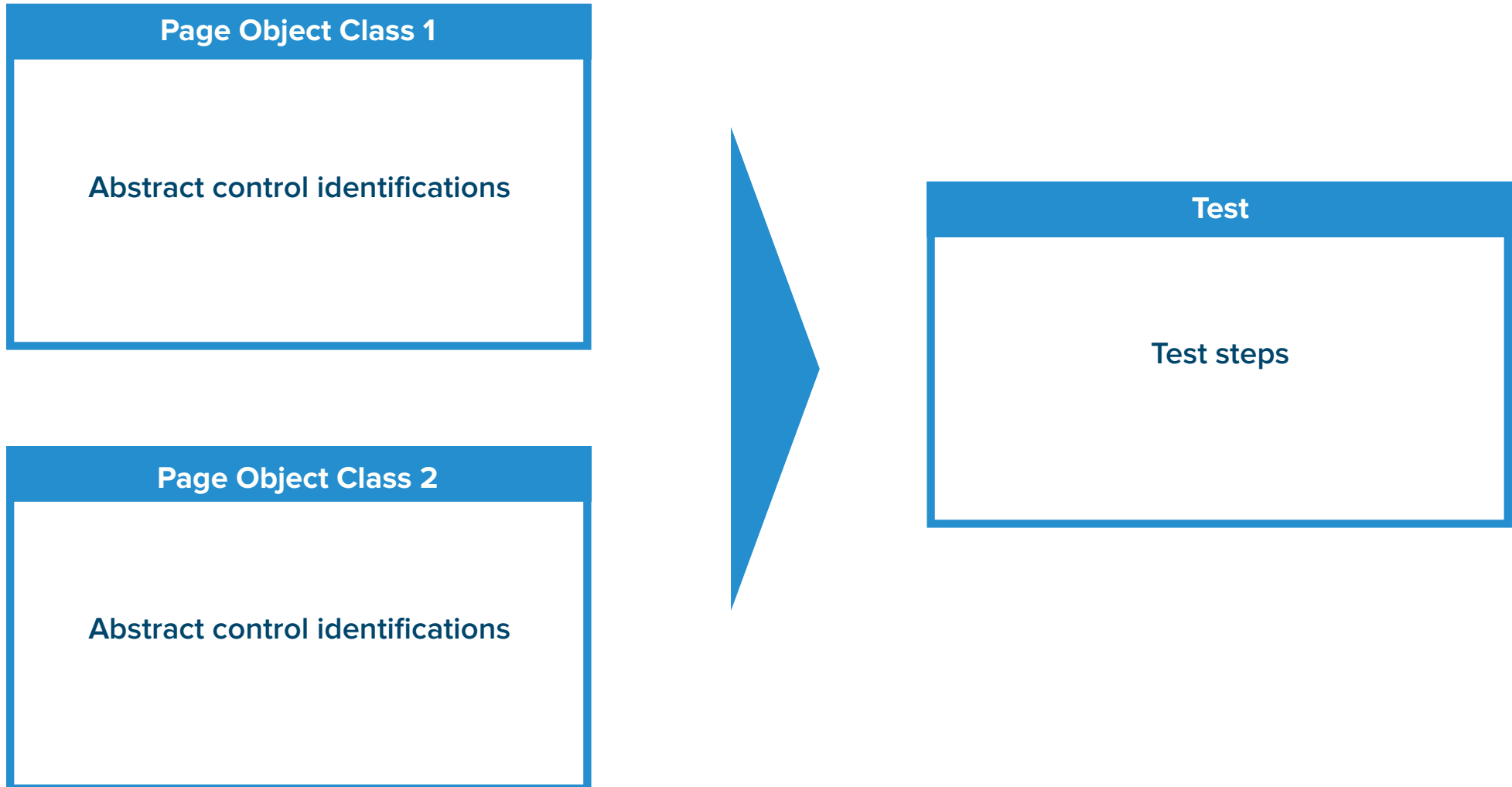
The first model we'll look at will be the easiest to implement but will offer the least amount of abstraction within the test. This method can be a good first step for UI testing teams that are in the early stages of implementing a UI automation strategy.

What does this look like in action?

In model 1, we are abstracting only the control identification for the object on the page. Let's for example you are interacting with login button control in your UI tests. There could be a number of ways to identify this controls including XPath, Name, or Property. The process would work something like this:

- Each webpage in your application would be represented as a class
- Within that class, the identifications of these controls are abstracted within a property or a method

A graphical representation looks something like the following. We have page object class for various pages and within the page object class we have classes for abstracted control identification. And using those page objects and abstracted identifiers, we create tests having different test steps.



Let's now apply the abstraction model 1 to the brittle test discussed in the previous part of this white paper.

```
private void Login(WebOrdersAppModel app, string username, string password)
{
    LoginPageModel login_page = app.LoginPage;

    login_page.UserNameEdit.SetText(username);
    login_page.PasswordEdit.SetText(password);

    login_page.LoginButton.Click();
}
```

← Using the abstracted control identification

```
public ITextEdit UserNameEdit
{
    get
    {
        return _page.Find<ITextEdit>(new WebElementPattern()
        {
            ObjectIdentifier = "MainContent_username"
        }, 3);
    }
}
```

← Abstracting control identification

The benefits:

By abstracting the control identification, you will condense the test into more manageable portions. So for this example, we have abstracted the control identification into the method: `UserNameEdit`.

Once we have abstracted the control identifier “Main-Content_username” in a method, we are simply calling the method in login model class. This then allows us to use the SetText method against it.

One of the bigger benefits of this abstraction is that if my object identification changes, we need to make changes at only place, unlike in the first model. This in turn makes the test more maintainable. But this model still has challenges. Let’s look at them:

The limitations:

To begin with there is some sort of redundancy still involved, which in turn results in maintainability challenges. Take the following as example: setText is being called twice in the Login Method itself. So as our test cases increase, if we want to change the set text to different action, a big repeatability takes place. So in the next model we would abstract actions into a page object.

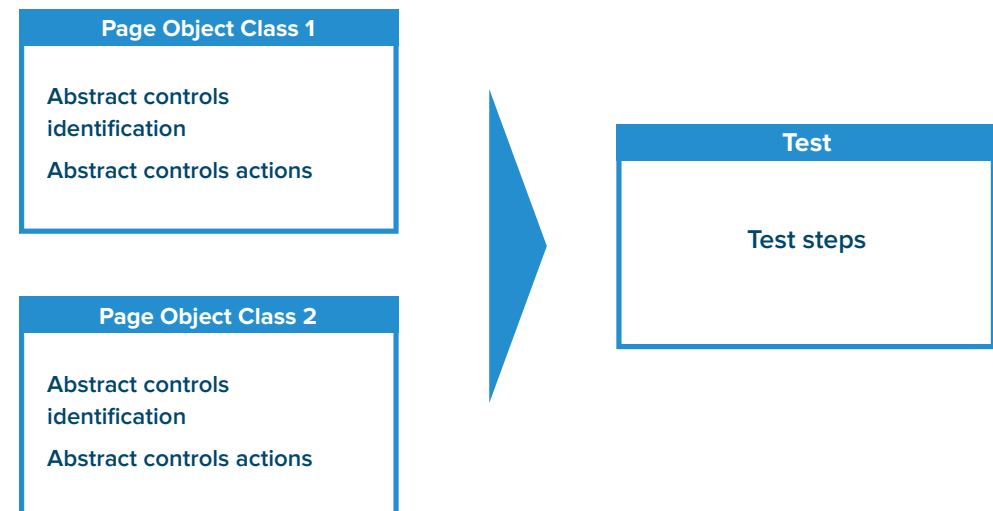
Abstraction Model #2: Abstracting the Control Identification and Action

Model #2 addresses some of the challenges that are still present in model #1.

Similar to the first model, we are abstracting the control identification for the object on the page within a class. So the process remains somewhat similar:

- Each webpage in your application would be represented as a class
- Within that class, the identifications of these controls is abstracted within a property or a method

But then, unlike the previous model, in this model you’re actually abstracting the control actions. Example of these control actions are clicking on a button, setting the text of a control, and getting the text. The graphical representation looks something like this:



When applied to the brittle test example discussed above, following is how the example looks like:

```
private void Login(WebOrdersAppModel app, string username, string password)
{
    LoginPageModel login_page = app.LoginPage;

    login_page.UserNameEdit = username;
    login_page.PasswordEdit = password;

    login_page.LoginButtonClick();
}

public string UserNameEdit
{
    set
    {
        _controls.UserNameEdit.SetText(value);
    }
}

public ITextEdit UserNameEdit
{
    get
    {
        return _page.Find<ITextEdit>(new WebElementPattern()
        {
            ObjectIdentifier = "MainContent_username",
        }, 3);
    }
}
```

← Using the abstracted set text action

← Abstracting the set text action for the control

← Locating the control

The benefits:

In the above example, we have abstracted the control identification in the method `UserNameEdit`. Similarly, in the method `UserNameEdit` we have abstracted the `setText` action. For instance, if later, you decide to send a string of key strokes instead of `setText` action, you can do it in just one method `UserNameEdit`.

The limitations:

Model #2 allows you to take the principles that are introduced in model #1 and more effectively scale it across test cases. For a majority of testing teams, following this framework will be enough to improve the scalability of your UI automation strategy. In fact, model #2 will likely solve a majority of the testing challenges you'll face when testing at the UI layer.

But the final area that is not addressed in this model is Magic Strings. In the method, `UserNameEdit`, we have string literal in a quoted string. Good programming practices state that instead of using literals within a test code, it's better to define a constant for the literal with a semantic name and in turn use that semantic name within the code. The challenge can be seen below and this is what we solve in third model.

```
private void Login(WebOrdersAppModel app, string username, string password)
{
    LoginPageModel login_page = app.LoginPage;

    login_page.UserNameEdit = username;
    login_page.PasswordEdit = password;

    login_page.LoginButtonClick();
}
```

```
public string UserNameEdit
{
    set
    {
        _controls.UserNameEdit.SetText(value);
    }
}
```

```
public ITextEdit UserNameEdit
{
    get
    {
        return _page.Find<ITextEdit>(new WebElementPattern()
        {
            ObjectIdentifier = "MainContent_username"
        }, 3);
    }
}
```

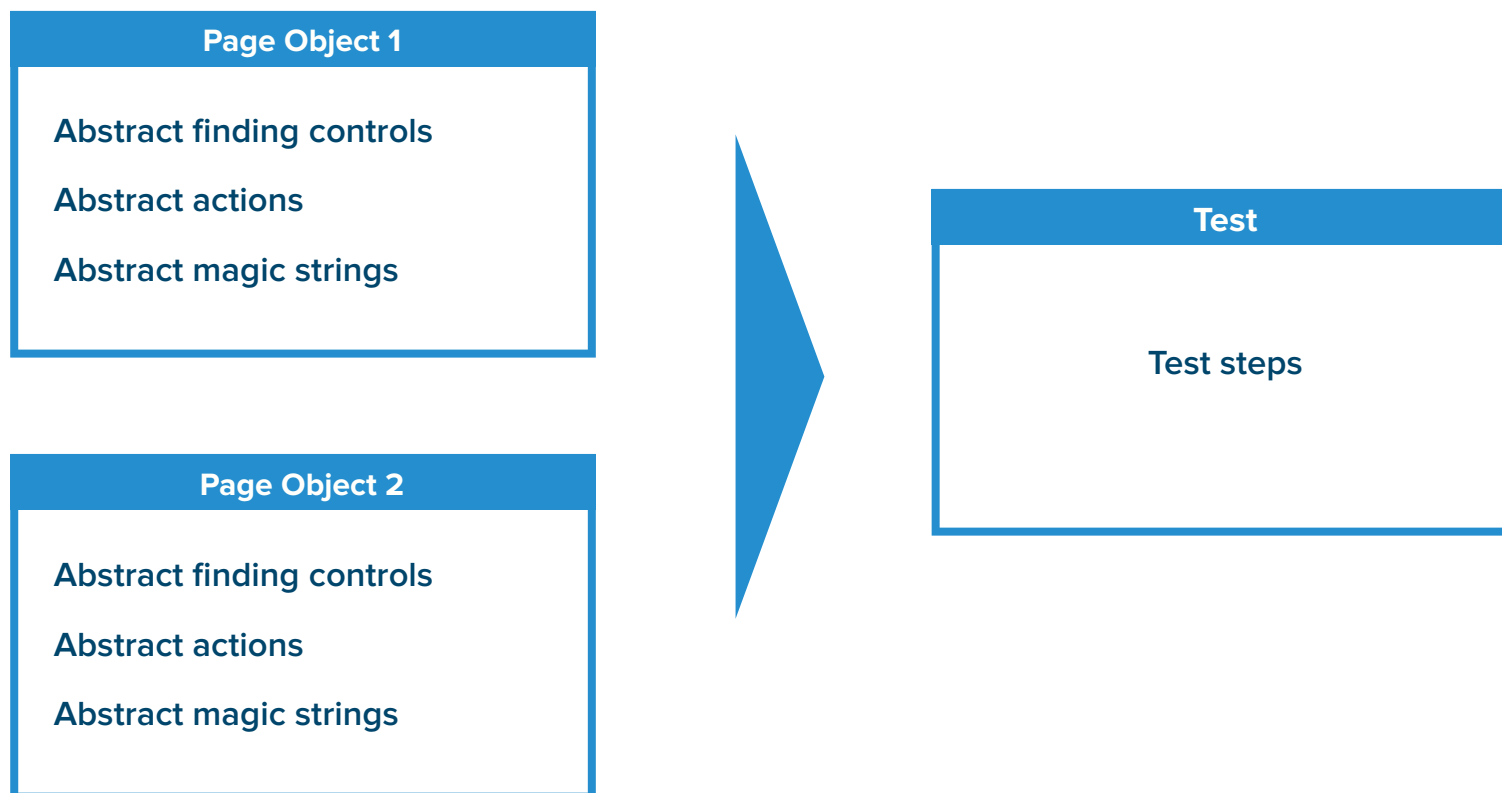
← **Magic Strings**

Abstraction Model #3: Control Identification, Action, and Magic Strings

The most efficient way to scale your UI test automation is to abstract all areas of the test. This includes abstracting control identifiers, abstracting actions, and finally abstracting Magic Strings as well. So to sum up, in this model you have a page object model where you create a class for different pages in the applications. Once you do that, you abstract the controls on that page — for example

login control, logout control, check box, text box, etc. Next, you'll abstract the actions which you perform on those controls. This code actions could be a `SetText`, `GetText`, or even a click of a button.

And then finally, you'll abstract magic strings as well. Similarly to what you saw in the previous models, you'll be able to apply this model to separate pages to create Test Steps. The graphical representation could look something like this:



To recap:

- While it's important to implement a test pyramid approach to limit the number of tests that are performed at the UI layer, you still need a plan for how you're going to scale your UI test automation. UI tests provide a level of feedback that is unable on the service or unit level, and failing to test at the UI level will lead to unforeseen defects that can cause issues in production.
- Scaling your UI test automation starts with following a page object model. In this model, you need to look at each page in your application as a separate class.
- Once you've created an object model, you can use abstraction to limit the complexity of your tests, reduce technical debt, and improve the maintainability of your test cases. The best approach is to abstract the control identification, the action, and the magic strings.



Scale your UI Test Automation Strategy with TestComplete



TestComplete

TRY IT FOR **FREE** TODAY



SMARTBEAR

Over 4 million software professionals and
25,000 organizations across 194 countries
use SmartBear tool

4M+
users

25K+
organizations

194
countries

[See Some Successful Customers >>](#)

API READINESS



Functional testing through
performance monitoring

[SEE API READINESS
PRODUCTS](#)

TESTING



Functional testing,
performance testing and test
management

[SEE TESTING
PRODUCTS](#)

PERFORMANCE MONITORING



Synthetic monitoring for API,
web, mobile, SaaS, and
Infrastructure

[SEE MONITORING
PRODUCTS](#)

CODE COLLABORATION



Peer code and documentation
review

[SEE COLLABORATION
PRODUCTS](#)



TestComplete