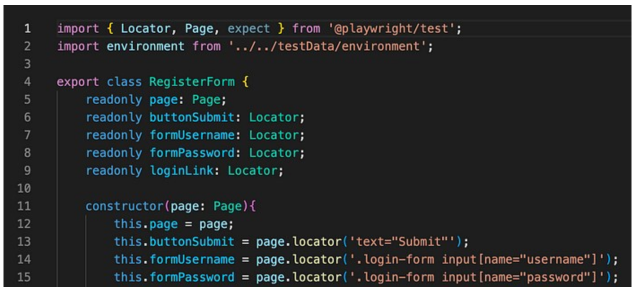
Member-only story

Going Deeper into the Page Object Model

Twelve design considerations when implementing page objects

Blake Norrish · Follow
20 min read · Jul 13, 2022

252 3


```
import { Locator, Page, expect } from '@playwright/test';
import environment from '../../testData/environment';

export class RegisterForm {
    readonly page: Page;
    readonly buttonSubmit: Locator;
    readonly formUsername: Locator;
    readonly formPassword: Locator;
    readonly loginLink: Locator;

    constructor(page: Page){
        this.page = page;
        this.buttonSubmit = page.locator('text="Submit"');
        this.formUsername = page.locator('.login-form input[name="username"]');
        this.formPassword = page.locator('.login-form input[name="password"]');
```
Just a random page object. You can probably find issues.

Google "page object model" and you will get over a million hits. Unfortunately, the vast majority of these resources provide only a high-level overview or a few simple examples. They provide a nice introduction but are woefully inadequate for real world page object challenges.

Below you will find twelve deeper topics on page objects, things that go beyond what you find in those million google hits. Should you use declarative or imperative interfaces? How and when to leverage an aggregator/actor layer? Is there ever a good time to use inheritance in page object class design? This is the stuff you'll have to figure out in real automation implementations.

Here's the list to whet your appetite:

Page Objects as a Design Pattern
Imperative vs Declarative Page Object Interfaces
Actor / Orchestrator Layer
Static vs Instance Methods — Chaining Page Object Calls
The Debate on Assertion Location
Page Objects vs Page Components
Page Objects and Inheritance
Interface Reusability — Separation What vs How
Centralized Locator Class vs Page Object Locators
DRY vs DAMP — BeforeEach and BeforeAll
Coupling in Page Object Interfaces
ScreenPlay and other Page Object Alternatives

While a full treatment of everything related to page objects would take a book, hopefully the introduction of these topics gets you thinking deeper about this valuable design pattern and points you in a direction toward healthier, more robust UI automation.

Page objects as a Design Pattern .

Page objects are just a design pattern and people seem to forget this. They are not magic and they are not rocket science; they are a specific organization of code that creates specific benefits.

All design patterns attempt to accomplish the same thing: they attempt to isolate the code that changes often, together, from code that changes differently, or not at all.

That is it. That is the purpose of every design pattern you have ever read about.

This is not how many people think about design patterns. When they think of design patterns, they think reusability, readability, coupling, cohesion, clear interfaces, encapsulation, etc.

100% yes, but these are *outcomes* of good design, effects of doing it right. The immediate goal of the design is still: put all the stuff that will change together in one place. Put stuff that changes for a different reason at a different time in another place. Put stuff that doesn't change at all in yet another place.

Do that and your design will inevitably be reusable, readable, have low coupling, high cohesion, etc. Do this over and over and you will see the same set of *patterns* emerge, and we call these design patterns.

The Page Object Model pattern isolates several types of changes, the most significant and obvious is the interface between the test code and the application DOM. UI tests must contain, somewhere, the knowledge of how to find elements on pages. This knowledge tends to change, and it tends to change together, per page.

Thus, page objects collect and encapsulate this knowledge in one place. They say: "hold on everyone, don't worry about HOW to find element x on page y, I WILL OWN all the knowledge about finding stuff on page Y. I am your 'find in Y' expert!"

When page Y changes because of a redesign, functional change, or any other reason, anyone needing to update this knowledge knows exactly where to go, as all logic to find stuff should exist in the page object for page Y. This knowledge should not exist anywhere else.

Another thing that changes often in UI tests is the steps necessary to perform the test, IE: "first click on this button, go to that page, input this text, click on that button, then look for this data". This knowledge is explicitly different and changes differently than the knowledge of how to find elements on a specific page. Thus, it is NOT part of page objects, and has no business existing there. This is knowledge the *test* should own, and you can very quickly identify a confused page object implementation by looking for test knowledge in page objects.

This understanding of design patterns does not dictate every detail of your page objects. There is a huge amount of subjectivity in "isolate what changes together", and a significant amount of personal preference and discretion in crafting your design based on the language, system design, complexity of the automation, etc. This is why test automation requires the critical and creative thinking of software development, and is not *scripting* (yuk).

Imperative vs Declarative Page Object Interfaces

With declarative page object interfaces you tell the object what to do. With imperative interfaces you tell the object how to do it. Here is a super simple example:

```
Declarative:
LoginPage.logInAs("test user", "test-password");
Imperative:
LoginPage.setUserName("test user");
LoginPage.setPassword("test-password")
LoginPage.clickSubmit();
```

This example is *almost* too simple, but it still shows the concept: in the declarative interface you tell the page what you want (to be logged in), in the imperative example you tell the page how to do it (enter these fields, click this button).

Readers who didn't skip the first section might yell: "But the declarative interface puts knowledge not related to finding stuff *inside* the Login page object! It knows *how* to login! Violation of good design!".

This is a valid criticism, and I have heard convincing arguments on both sides. Does the knowledge of *how* to login by performing three separate actions constitute something *outside* of the single responsibility of a page object? I say no, but even if you disagree at least we are framing the conversation correctly. Too many automators simply throw methods into files or classes wherever they are convenient.

One thing to notice is that declarative style almost always creates a *higher level of abstraction*, which segues nicely into our next topic.

Actor / Orchestrator Layer

One of the problems with page objects, especially for large user-journey style tests that require many steps, is that they can get quite long. The interface between the page object and the test is actually more granular than what the test really wants, and this creates tests overburdened by small, discrete steps. For example:

```
LoginPage.setUserName("test user");
LoginPage.setPassword("test-password");
LoginPage.clickSubmit();
LandingPage.searchForItem("test-item");
LandingPage.selectCurrentItem();
DetailsPage.selectSize('XL');
DetailsPage.selectQuantity(1);
DetailsPage.selectColor("white");
DetailsPage.selectReoccuringPurchase(false);
DetailsPage.addToCart();
```

This example might look simple and easy to read, but this is just a few basic steps in what might be a much longer and more complex test. In a realistic user journey that leverages page objects exposing this granular of interface, the eventual test might be many hundreds of individual steps.

Not only does this make the test hard to read, it violates Chekhov's Gun Principle, as while all the steps are necessary, many of them are probably not relevant. In the example above, we must select a size, color, and quantity, but these selections are not important for the test — we just need *any* valid selection.

As we discussed in the previous section, moving from an imperative to a declarative interface can significantly reduce the superfluous "noise" in a test. The example above would go from:

```
DetailsPage.selectSize('XL');
DetailsPage.selectQuantity(1);
DetailsPage.selectColor("white");
DetailsPage.selectReoccuringPurchase(false);
DetailsPage.addToCart();
```

To:

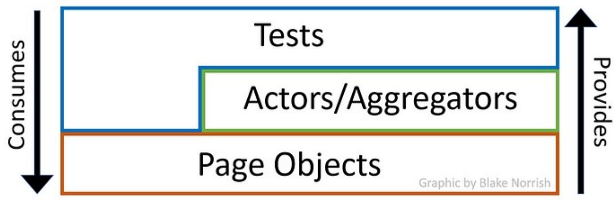
```
DetailsPage.addValidItem();
```

This collapses all the "how" steps of adding an item into a simple "what" step and significantly reduces test noise.

Unfortunately, if you have many* journey tests and many of them repeat the same basic set of steps (logging in, selecting a valid item, etc) then moving from an imperative to a declarative style interface might not be sufficient. What you *really* need is an even higher level of abstraction, one that actually *spans* page objects.

This is where the actor or aggregator pattern comes in. The names are interchangeable and I have seen this pattern called many other things as well, but whatever the name they all serve the same purpose. To simplify, I'll use 'actor'.

Actors combine actions across page objects to expose these common aggregated activities to pages in reusable chunks. They provide a higher level interface for tests to consume. Thus, every test doesn't need to re-implement this sequence, they just consume the interface exposed by the actor.



This pattern of inserting a layer between page objects and tests is actually quite common, although many people use it without realizing it. Automators see a bunch of repeated steps in a bunch of tests, create a "helper" that collects those steps into one location, and use the helper. Voilà, actor pattern. (more on this in the DRY vs DAMP section!)

In the example above, we could aggregate the entire set of steps across all three page objects into something like:

```
ShopperActor.logonAndSelectItem();
```

This method could take arguments to give the test slightly more control, but you get the idea. An important note is that actors should *not* leverage the underlying UI library (selenium, playwright, etc) like page objects do, they *only* leverage page objects to drive UI behavior. Otherwise, they are really just bloated page objects.

Leveraging actors for common action sequences and only using page objects directly for more fine grained control can make complicated user journey tests significantly more readable.

*- but why do you have a lot of journey tests in the first place?

Static vs Instance Methods— Chaining Page Object Calls

In the examples above, all page object interactions were implemented as static methods within page object classes. This is usually how page objects start, and what I see most often from automation teams. A different approach is to implement page objects as instance methods that return an instance of the page object that matches where the browser should be after that interaction. This allows "chaining" of steps and can reduce visual clutter in your test.

This might sound complex but is actually quite simple. For example, the DetailsPage.selectSize() method would return a reference to 'this', as after selecting a size, as we still expect to be on the detail page. If we did this with all methods, it would look like this:

```
new DetailsPage()
    .selectSize('XL')
    .selectQuantity(1)
    .selectColor("white")
    .selectReoccuringPurchase(false)
    .addToCart();
    .checkOut();
```

Notice that the 'addToCart' method would return an instance of a *different* page object (CartPage or something, based on this example) because after clicking the 'addToCart' button we expect the browser to be on the cart page, not still on the details page.

Implementing your page object interface to support method chaining can even be done in functional languages like javascript. Regardless of how it is implemented, it is a valuable tool for improving test readability when using page objects.

The Debate on Assertion Location

One contentious discussion often heard during the implementation of page objects is *where to put assertions.* One school of thought: *Assertions belong in tests, never in page objects.* Tests use general purpose getters to pull the information necessary, and the actual assertion on that data lives in in the test. This is a clean separation, as it is tests that should own the knowledge of what makes a test pass or fail.

This works well in theory, but in practice can quickly cluttern tests with a significant number of assertions. Thus, a second approach is to embed assertions into page objects, *but make them obvious when reading the page object interface.*

For example:

```
New ItemDetailsPage()
    .addtoCart(item)
    .checkout()
    .assertCartIsEmpty();
```

In this example, it is obvious that the last step is an assertion, and that it is asserting that the cart is empty. You should **never** hide functional assertions inside steps in ways that obfuscate what is being asserted. If months after your test has been automated, someone reads it and can't immediately and definitively know what the test is asserting, you have failed. Remember: code is read 10x more than it is written — optimize for readability.

There is a different type of assertion that I think can exist within page objects: assertions that are not *functional* assertions but simply checks that the current state of the browser is what is expected.

For example, when using instantiated page objects (new ItemDetailPage()), you can make it a requirement that when the object is instantiated, it verifies that the browser is on the expected page. Thus, the constructor of the ItemDetailPage will look for and verify some element or page title. This allows tests to "fail fast" when something goes wrong, even without an explicit assertion at the test level.

There is a lot of leeway when designing assertions into your page object model and tests, and everyone has their own opinion. Be consistent with your approach and ensure that every person implementing automation understands and follows the direction chosen.

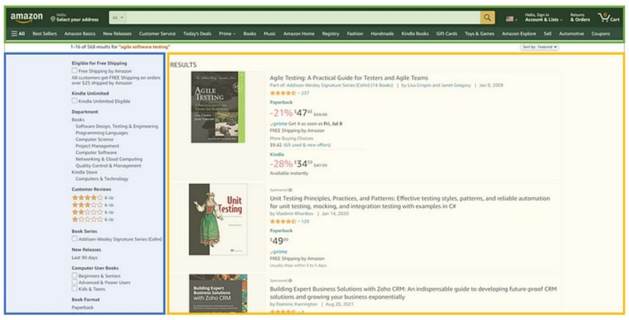
Page Objects vs Page Components

Page objects generally map to pages. However, most modern web applications are not built as a set of unique pages. Instead, pages are aggregated out of a set of reusable components. For example, a header component might exist at the top of every page, and a cart component might exist on the right side of most shopping related pages.

It does not make sense to duplicate the knowledge of how to find elements in a header component across all page objects that include the header. Instead, create *page components* that map to smaller parts of pages (the header in this example).

What constitutes a component? It very much depends on the architecture and layout of your specific web application. UI frameworks like React are organized around reusable components, so oftentimes this is a great starting point. However, React components tend to be much more fine-grained than what you will want in your UI automation.

Here is an example of the Amazon.com search results page divided into three components: 1) the header component (green), the search filter component (blue), and the search results component (orange). This isn't the only way you could create page components out of this page, but it's probably a good start.


amazon
RESULTS
Agile Testing: A Practical Guide for Testers and Agile Teams
Unit Testing Principles, Practices, and Patterns: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C#
Building Expert Business Solutions with Zoho CRM: An indispensable guide to developing future-proof CRM solutions and growing your business exponentially

One other design choice with page components: some like to leverage page components no differently than page objects, calling them directly from tests. A different approach is to *compose* page components within page objects. Thus, in the example above you would write something like:

```
SearchResultsPage.FilterComponent.setStarRating(5);
```

This second approach has the drawback of adding a layer of knowledge that must be maintained as the web application evolves (which page components are part of which pages) but has the benefit of making it very obvious which page component to use. In a large web application, there could be hundreds of different components and several that are related to search results. In the example above the automator doesn't have to search for the correct one, they immediately and definitively know it's the one attached to the SearchResultsPage.

Page Objects and Inheritance

Should page object classes inherit from a common page class? If so, what goes into that base class, and why?

I have seen some *clever* uses of inheritance in page object design. Never have I seen a case where the value of the cleverness made up for the complexity it created. In test automation as in general software development, avoid being clever.

One unfortunately common use case of inheritance in page object class hierarchy is to give all page objects access to "helper methods" that wrap the underlying UI automation library (eg: Selenium). Some automators feel the interface exposed by these libraries is either too complex, or not powerful enough, and create a layer between the library API and the page objects that use them.

In many cases the motivation to do this is just ignorance of the API library. If you find yourself wrapping every call to the underlying API, you have to ask why the API wasn't just written like yours to begin with. Maybe you just don't understand how that API works.

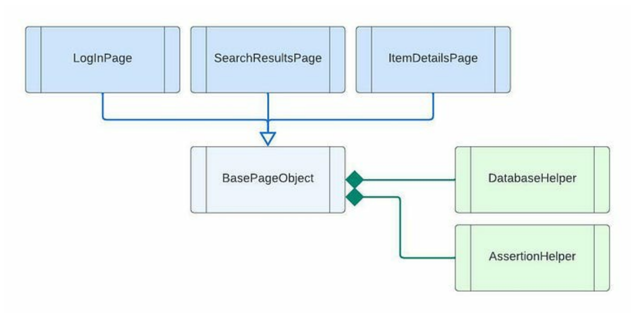
Another common but often dubious example of inheritance in page object classes is creating base classes for pages that are very similar, but have different locators. For example, a SearchResultsPage that has been localized into English, French, and German.

In this example, a FrenchSearchResultsPage would inherit from a BaseSearchResultsPage. The BaseSearchResultsPage would define the interface used by every SearchResultsPage, and implement any methods that are not affected by the localization. Each language-specific page would then implement methods specific to it.

In practice, this use of inheritance does not add a significant amount of value, other than proving the implementer likes to use inheritance. The same outcome could have been achieved using page components and simple composition, or just allowing for some minimal duplication across the language specific pages.

The most dubious use of inheritance in page objects is to simply collect every possibly useful helper class into one spot, for the convenience of the page object implementor. For example, the BasePageObject would either aggregate or compose database helpers, assertion helpers, test data helpers, and every other helper in existence in the repository. Thus, every page object derived from the base page object immediately gets access to everything and anything it could possibly want.

A simplified class diagram of this design. In practice there are usually *many* more of the green classes on the right:



While this can seem helpful, it creates bloated page objects and makes the BasePageObject into a type of God Object.

The general guidelines for page object class design: prefer composition over inheritance, prefer simplicity even at the cost of small amounts of duplication, and *always* avoid being clever for the sake of being clever.

Interface Reusability — Separation What vs How

Page objects provide an interface to tests that should greatly expedite future test development. Think of them as providing a set of lego bricks that tests can use to quickly construct new and interesting tests.

Unfortunately, many page object implementers make their interface too customized to their specific test, at the cost of being generally useful for other, future tests. Rather than sloped bricks, flat bricks, and wall bricks, they just give you a house brick which can only ever be a house. Great if you need a house, absolutely useless if you want to create a dragon.

One common example of this is when page objects arrogate the knowledge of *what* to do. For example, in the simple login case:

```
LoginPage.logIn();
```

In the above method the LoginPage assumes it knows what user to login as. This creates a very specific lego brick. A much more general purpose method would be:

```
LoginPage.logInAs("test user", "test-password");
```

This example is almost too obvious. When implementing a complex set of page objects to support an automation suite, there are many cases where it is tempting to let page objects make decisions on what data to send, and when. Resist this urge! Page objects should only know how to translate test intent into browser actions, they should not make decisions on what to click or what data to send. That knowledge is entirely the responsibility of the Test.

A more realistic but equally insidious example of this is when page objects access some type of global state to pull in test data (login data, etc.). The page object does not *directly* decide what to do, but it does 'phone a friend' to get that information.

In the login case, the LoginPage might have a reference to data created in a BeforeSuite or BeforeEach method and saved in a global "current user information" store for later use. This might seem advantageous, but it still overloads the responsibility of the page object.

When used extensively, this pattern also makes tests incredibly challenging to read. It hides what is going on and forces the reader to click into a lot of different places to understand the expected behavior of the test. This overlaps with the DRY vs DAMP topic we will cover later.

Craft page object interfaces that are simple to understand and useful across a range of tests, that allow the next test implementer to build new tests quickly and efficiently.

Centralized Locator Class vs Page Object Locators

Each page object will contain many locators for relevant elements on that page. Seeing all the locators across all the pages, it is tempting to pull these locators into some sort of centralized Locators class, and have every page object reference that.

For example, in the LoginPage we would see the following code, rather than the Locator:

```
Locators.LoginPage.SubmitButtonLocator
```

While collecting similar things that change together is a goal of all software design, this strategy goes a bit far. While there are language constructs that can help manage the organization of hundreds or even thousands of locators, using clean and consistent Locator implementation across all page objects should allow anyone to find and update Locators quickly and easily. Centralized Locators more often create more complexity than they are worth.

DRY vs DAMP — BeforeEach and BeforeAll

DRY — Don't Repeat Yourself, is a common principle used to guide software design. New coders are taught to look for repeated sections of code or logic and to move these into some reusable thing… a function, class, library, etc. In 99% of software development, DRY is very beneficial and appropriate.

Unfortunately, automated tests fall into that other 1%. That's right, it is often better to repeat yourself in automated tests. Why is this?

In tests, the duplication of *test steps* within tests is quite common, but the benefit to readability of *actually seeing every step within the test* outweighs the benefit of having those steps refactored into a single location.

For example: dozens of tests might all need to log in then add an item to a cart, before going on their merry way. These initial common steps might look like a great candidate to extract to remove duplication, but in fact should be left right there in the test.

This principle is called DAMP — Descriptive and Meaningful Phrases. It prioritizes verbosity for readability over duplication, and should be your guiding principle in tests.

"HEY NOW!" you might say. "What about the actor pattern??!!" Isn't using something like:ShopperActor.logonAndSelectItem(); exactly the DRY and anti-DAMP thing you are saying to avoid?

Good question, but not really. Yes, that actor takes a bunch of steps and aggregates them to be reused by tests, but this is more about changing the level of abstraction that is appropriate for optimal test readability than about removing duplication. DAMP improves readability, and we used the Actor specifically to improve readability. Actors are actually very compatible with DAMP.

Here is a test (pun!) you can use to see if your tests are DAMP: Zoom in on *just* your test so that it is all you can see. Now, take your fingers off the keyboard and away from the mouse, so you can't click on anything, can't open any other files, and can't get to the implementation of any of the functions called in your test. Then: ask someone who doesn't know the intent of the test to describe what the test does (or feign ignorance and do it yourself). If your test is DAMP, it will be easy, obvious, and with zero ambiguity.

This is why Actors don't violate DAMP. Reading ShopperActor.logonAndSelectItem(); tells me exactly what I need to know to understand what the test is doing.

Placing test steps into BeforeEach and BeforeAll behavior are especially pernicious violations of DAMP. These features (every execution framework has them) should *never* be used to execute common test steps. Doing this rips the test apart and puts its pieces in multiple locations. I don't care if *every* test has to login first, don't login in your BeforeEach.

What *should* go into BeforeEach and BeforeAll? Any setup that is not a functional step in the test — stuff that if present in the test itself would clutter the test and decrease readability. Funny how it always comes back to readability, doesn't it?

While the DRY vs DAMP principle applies to *all* test automation, it is relevant to the page object conversation because it overlaps with the Actor pattern. DAMP tests and actors provide very similar benefits: they limit the test function to only and exactly the steps necessary to make the test readable..

Like in everything else, there is still a huge amount of flexibility available and critical thinking required when attempting to create DRY or DAMP code — what is DRY to one person might seem a bit soggy to the next. Test automation is not formulaic, and you will have to use your noggin and your intuition.

The DRY vs DAMP discussion deserves more space than I can allocate here. Google it and you'll get a lot of content on this specific topic. Here's a post by Vladimir Khorikov that I particularly like.

Coupling in Page Object Interfaces

Coupling is a design principle that describes the amount of dependency or interconnectedness between parts of a system. If two parts have significant dependency on each other, if changes in one part require significant changes in another, we say those two parts are coupled. Low coupling is considered advantageous.

One aspect of page objects that impacts coupling between tests and pages is the type and number of arguments passed between tests (or actors) and page objects. Often we hit situations where a large amount of data must be sent from test to page object. Do we pass this in as a list of 10 variables? Or do we create a single object and pass this in?

For example, let's say we are creating a declarative method on an account creation page:

```
AccountCreationPage.createAccount("full name",
    "test_username",
    "test_password",
    false, // isAdmin
    "867-5309",
    "Test_city",
    "Test_state",
    …);
```

Or:

```
AccountCreationPage.createAccount(accountDescriptor);
```

The second implementation using the accountDescriptor object creates higher coupling between the test class and the page object class. Using primitive types does not.

Which should you choose? Unfortunately I can't tell you. It depends on the nature of your application and even the language your automation is implemented in. In a weakly typed language like javascript where you can create objects on-the-fly with object literal notation the math on coupling is significantly different than in statically and strongly typed languages (C#,java,etc).

There is no one right answer to how you should construct the interfaces between your page objects and consumers. However, understand that if you create a significant amount of these intermediary objects, they inherently couple tests and page objects together. This may create additional work when one side of this coupling needs to change. However, this may still be the correct way to go. Isn't programming fun?

ScreenPlay and other Page Object Alternatives

The page object pattern is not the only kid on the UI automation block, and many people advocate for alternatives approaches or at least significant alterations to the basic page object design we are familiar with.

For example, the Screenplay Pattern is an alternative that revolves around actors, tasks, activities, and actions. It attempts to solve the same problems as the page object pattern, but in a different way. The Cypress Blog advocates for App Actions, although I disagree with their "page object problems" and their implementation relies on Cypress/DevTools ability not available to WebDriver based frameworks.

The Lean Page Object (middle of article) pattern is still a page object, but advocates for returning locators from page object methods. In this way, the consumer of the page object must own what to do with the elements, and page objects become an extremely thin layer for finding things on a page.

Is the screenplay pattern *better* than the page object pattern? I don't know, is French better than English? Is American English better than British English? The value of a language is only measurable relative to the audience you are communicating with… if your team really understands page objects, I don't see any benefit of changing the language of your UI automation. If you are starting from scratch, it might be worth a conversation.

Further Reading

As I said before, you get a million hits when you google "page object". However, every self respecting test automator should at least read the Martin Fowler article on the subject.

After that, the best strategy to gain a deep understanding of page objects is not reading about page objects (other than this blog, of course!), it is reading about and understanding software design principles and design patterns. Test automation *is* software development, and all the expertise relevant to software is relevant to test automation. Read about SOLID, read more about DRY vs DAMP, learn all the other design patterns and how they are used, learn about functional programming and how these concepts apply differently, the list of topics is almost endless.

Software development is easy to learn but hard to master, and this is part of what makes it so absolutely, fantastically fun.

Happy automating.

Liked this? Clap to send dopamine!

Follow me for more on quality assurance, quality engineering, software development, and related musings.

Some of my own personal favorites:

On Untestable Software

Your testing problem is really a testability problem

medium.com

Don't Automate Test Cases

How directly automating test cases leads to unwieldy and bloated automation suites that provide little or no value.

medium.com

Quality Engineer Learning Roadmap

A beginner's guide to the skills, tools, and technologies you need for a career as a Quality Engineer or SDET

medium.com

Quality Assurance Test Automation Front End Development Software Testing Design Patterns

252 3

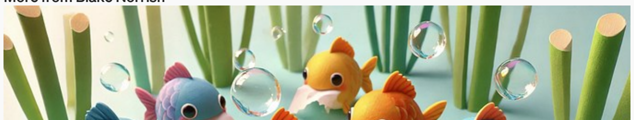
Written by Blake Norrish

Follow

2.7K Followers

Quality Engineer, Software Developer, Consultant, Pessimist — Currently Sr Director of Quality Engineering at Slalom Build.

More from Blake Norrish

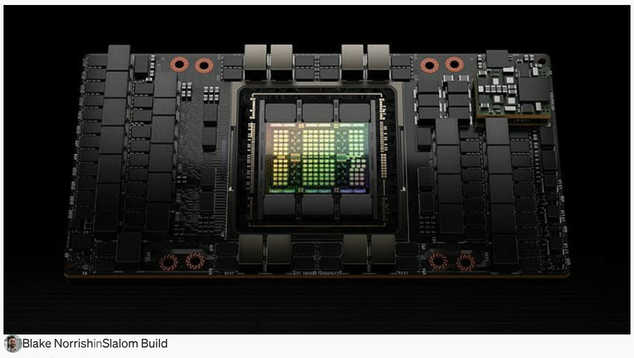


Blake NorrishinTowards Data Science

AI Is Eating Your Algorithms

How simple prompt engineering can replace custom software

Jul 8 90

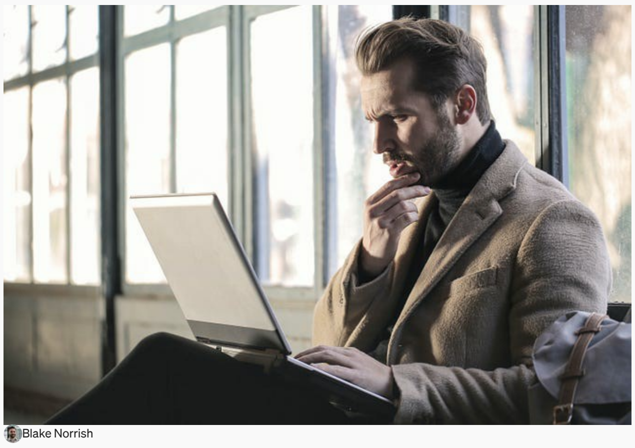


Blake NorrishinSlalom Build

RAG for Quality Engineers

Building RAG is easy, building quality RAG is hard

Mar 28 442 5

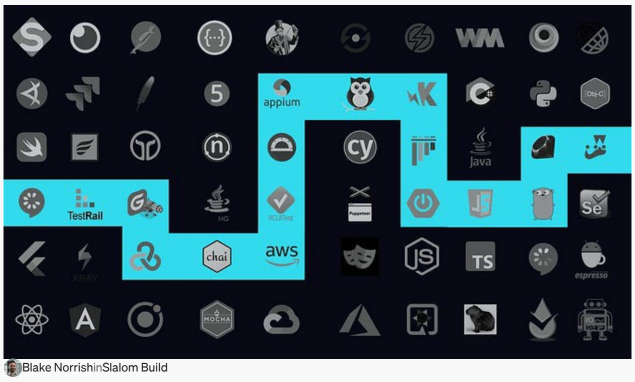


Blake Norrish

How to Think Like a Tester

It's not an innate skill that only some are born with, nor is it a trivial activity that anyone can do. Here are five tips on how you can...

Oct 14, 2021 315 4



Blake NorrishinSlalom Build

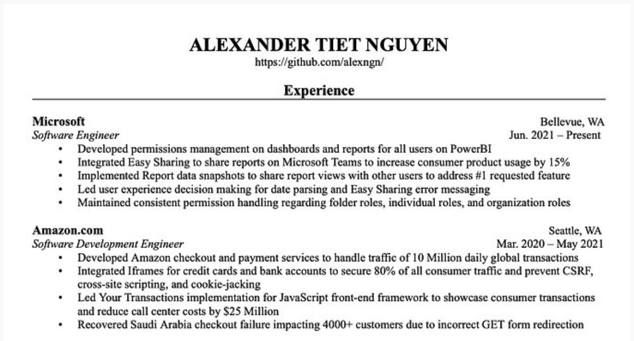
Quality Engineer Learning Roadmap

A beginner's guide to the skills, tools, and technologies you need for a career as a Quality Engineer or SDET

Feb 19, 2021 1.6K 15

See all from Blake Norrish

Recommended from Medium

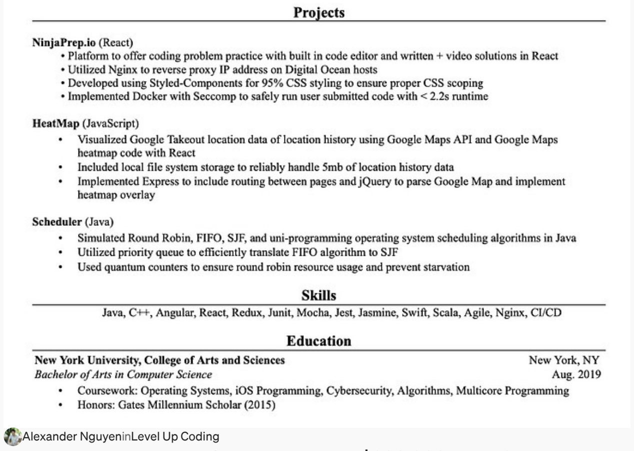


ALEXANDER TIET NGUYEN
https://github.com/alexngn/

Experience

Microsoft Bellevue, WA
Software Engineer Jun. 2021 – Present
- Developed permissions management on dashboards and reports for all users on PowerBI
- Integrated Easy Sharing to share reports on Microsoft Teams to increase consumer product usage by 15%
- Implemented Report data snapshots to share report views with other users to address #1 requested feature
- Led user experience decision making for date parsing and Easy Sharing error messaging
- Maintained consistent permission handling regarding folder roles, individual roles, and organization roles

Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021
- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection



Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Scheduler (Java)
- Simulated Round Robin, FIFO, SJF, and uni-programming operating system scheduling algorithms in Java
- Utilized priority queue to efficiently translate FIFO algorithm to SJF
- Used quantum counters to ensure round robin resource usage and prevent starvation

Skills

Java, C++, Angular, React, Redux, Junit, Mocha, Jest, Jasmine, Swift, Scala, Agile, Nginx, CI/CD

Education

New York University, College of Arts and Sciences New York, NY
Bachelor of Arts in Computer Science Aug. 2019
- Coursework: Operating Systems, iOS Programming, Cybersecurity, Algorithms, Multicore Programming
- Honors: Gates Millennium Scholar (2015)

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

Jun 1 · 14.8K · 226

Abhay Parashar in The Pythoneers

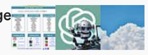
17 Mindblowing Python Automation Scripts I Use Everyday

Scripts That Increased My Productivity and Performance

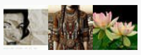
3d ago · 4.3K · 34

Lists

General Coding Knowledge
20 stories · 1423 saves



ChatGPT
21 stories · 731 saves



Medium's Huge List of Publications Accepting Submissions
334 stories · 3158 saves

Samantha-Kun in ILLUMINATION

I Slept With All Kinds Of Men For 3 Years

And that's what I learned about them

Jul 3 · 5.3K · 170

Afan Khan in JavaScript in Plain English

Microsoft is ditching React

Here's why Microsoft considers React a mistake for Edge.

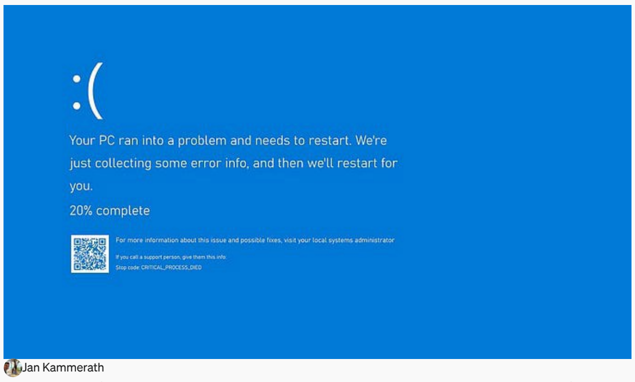
Jun 6 · 2.9K · 66

Venu Vignesh M

Data-Driven Testing with Cucumber: Simplifying Software Testing

Data-Driven Testing with Cucumber: Simplifying Software Testing

Feb 28



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit your local systems administrator

If you call a support person, give them this info:

Stop code: CRITICAL_PROCESS_DIED

Jan Kammerath

Inside The Outages: A Dangerous Null Pointer Exception Deployed On Friday

The world went into shock when cyber security firm "Crowdstrike", a provider of endpoint protection software, released an update on Friday...

6d ago · 3.9K · 98

See more recommendations