

<https://doi.org/10.31891/csit-2026-3-3>

Yurii PONOCHOVNYI

Dr.S. on Engineering, Professor, Professor
of the Department of Information Systems
and Technologies

Poltava State Agrarian University

<https://orcid.org/0000-0002-6856-2013>

e-mail: yuriy.ponochovnyy@pdau.edu.ua

Oleksii KUDENKO

Master of Science in Electronic
Communications and Radio Engineering
Poltava State Agrarian University

<https://orcid.org/0009-0007-0733-6980>

e-mail: kydster95@gmail.com

Received: 18/06/2026

Accepted: 14/08/2026

Published: 30/09/2026

© Copyright
2026 by the author(s)



This is an Open Access article distributed
under the terms of the [Creative Commons
CC-BY 4.0](https://creativecommons.org/licenses/by/4.0/)

UDC 004.415.53:004.42

PATTERN PAGE OBJECT MODEL FOR MODULAR TEST AUTOMATION ARCHITECTURE

Modern web applications evolve through short development cycles and frequent interface redesigns, which often make UI test suites fragile because scenarios depend directly on page structure, selectors, timing, and navigation logic. The paper examines the Page Object Model (POM) as an architectural foundation for modular test automation rather than a technique for storing locators. The study combines analysis of recent empirical literature on GUI test maintenance with a case illustration based on a Playwright and TypeScript framework. POM is compared with the Screenplay pattern and Component Object approach by separation of concerns, abstraction granularity, readability, reuse, scalability, and implementation complexity. The proposed interpretation distinguishes test scenarios, page interfaces, reusable component objects, fixtures, services, data factories, and utilities as separate architectural layers, formalising change propagation, cohesion, reuse, and maintenance cost through six numbered indicators. Attention is paid to TypeScript contracts, dependency injection, Playwright locators, fixtures, and isolated browser contexts as means of maintaining explicit dependencies and predictable module boundaries. A qualitative maintainability model, consistent with ISO/IEC 25010, explains how centralising interface knowledge reduces change propagation. The illustration shows page objects exposing business operations while component objects encapsulate recurring fragments such as navigation panels, modals, and notifications. POM improves maintainability only when responsibilities are limited to interface interaction and state representation; oversized classes, assertion-heavy methods, deep inheritance, and uncontrolled cross-page dependencies turn POM into a source of coupling. A modular POM implementation should be treated as an adaptable architectural layer whose effectiveness depends on cohesion, explicit contracts, composition, and controlled abstraction.

Keywords: Page Object Model, test automation architecture, modularity, Playwright, TypeScript, UI testing, maintainability, component objects.

Introduction

Contemporary web products are released incrementally: interfaces are redesigned without altering business rules, components are replaced, and navigation flows change frequently. Automated user-interface tests must supply fast regression feedback after every such change, yet as software artefacts they are exposed to the same architectural pressures as the system under test. When test code consists of a linear sequence of low-level browser commands, a single selector change, renamed field, or modified dialog can force edits across dozens of files.

UI tests are particularly sensitive to four sources of variability: document structure, element identification, asynchronous state transitions, and the composition of interactions into cross-page workflows. Modern automation tools such as Playwright reduce timing-related instability through automatic waiting and retryable locators, and recommend user-facing attributes (roles, labels) for identification [1, 15], but they do not by themselves determine how a large test codebase should be decomposed.

The Page Object Model (POM) is the most established pattern for separating test intent from page-specific interaction logic: a page object represents an interface to a page (or a significant part of one), and test scenarios call its methods instead of manipulating selectors directly. Selenium documentation frames POM as a mechanism for reducing duplication and localising page-specific changes [2]. In practice, however, POM is frequently reduced to a container of locators, or, conversely,

accumulates navigation logic, data generation, assertions, API calls, and business rules until each class becomes a procedural script disguised as an object. The central problem addressed in this paper is therefore not whether page objects exist, but whether their responsibilities, dependencies, and abstraction boundaries are designed coherently – a problem with direct practical consequences for the cost of maintaining industrial test suites.

This paper investigates the conditions under which POM can serve as the foundation of a modular test automation architecture for Playwright/TypeScript frameworks – not merely as page-level locator storage, but as a layered system that also includes component objects, fixtures, services, data factories, and utilities. The following research questions are addressed.

RQ1. Which architectural characteristics allow POM to reduce the propagation of UI changes through a test suite?

RQ2. How does POM differ from the Screenplay and Component Object approaches in abstraction, reuse, and implementation complexity?

RQ3. How can Playwright and TypeScript mechanisms support explicit and maintainable POM module boundaries?

RQ4. Which POM implementation practices increase coupling and should therefore be treated as architectural anti-patterns?

The aim of the study is to develop a layered architectural interpretation of POM, integrate it with recent empirical findings on GUI test maintenance and code review, and validate the resulting principles through a Playwright/TypeScript case illustration and a set of qualitative maintainability indicators. The paper makes three contributions: (1) a layered interpretation of POM as an abstraction boundary between business-oriented tests and browser interaction; (2) an integration of page objects with component objects, fixtures, services, and typed contracts, illustrated on a Playwright/TypeScript case; and (3) qualitative change-impact indicators, formalised as numbered relationships, that can later be operationalised in empirical studies. The work extends the authors' earlier framework-modelling material [17], narrowing the scope to modular UI architecture with a new comparative and analytical treatment rather than reproducing the previous structure.

Analysis of Recent Research and Publications

Page Object Model. As originally described by Fowler [9], a page object stores knowledge about how automation observes and manipulates a page, while a test expresses why an operation is performed – an adapter between the test domain and the browser API. A minimal implementation contains a reference to a page or component root, locators for relevant controls, operations representing meaningful interactions, and optional state queries. The value of the pattern is information hiding: a test calling `signIn(credentials)` does not need to know whether the control is a button, a link, or a custom widget, and can remain stable while the underlying markup changes.

This benefit can be expressed through a change propagation set (Fig. 1). Let U be the set of interface elements and, for $u \in U$, let $D(u)$ be the set of test modules directly encoding knowledge about it. In a script-oriented design, the number of edits required after a local UI change is approximately proportional to $|D(u)|$. In a page-object design, that knowledge is centralised, so the expected propagation becomes

$$P(u) = |M(u)|, \quad (1)$$

where $M(u)$ is the (small) set of page or component modules that encapsulate the element. POM cannot make $P(u)$ in (1) equal to zero – a genuine interface change still requires adaptation – but it keeps $P(u)$ small and predictable. Centralisation alone, however, is insufficient: a single class with hundreds of methods can exhibit low duplication yet low cohesion, so page-object boundaries must still align with stable interface responsibilities.

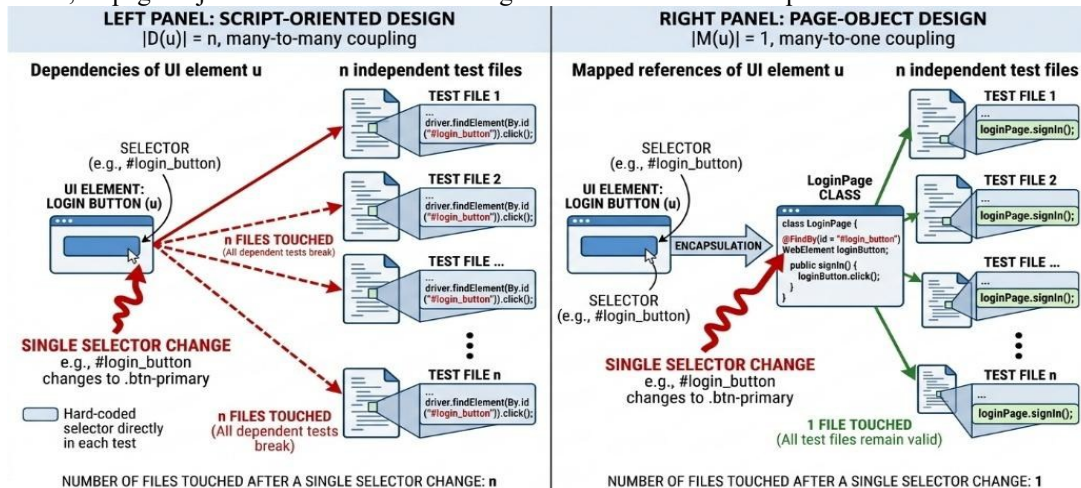


Fig. 1. Change-propagation model of a UI element under script-oriented and page-object designs

Empirical evidence on the manual, artefact-oriented side of POM maintenance remains limited but growing. Fulcini et al. derived a set of guidelines and a companion static linter for GUI test code from a semi-systematic review of open-source Selenium projects, and explicitly flagged incomplete or inconsistent adoption of the Page Object pattern itself as one of the recurring "test smells" that undermines the maintainability gains POM is supposed to provide [18]. Many of the anti-patterns discussed later in this paper (Table 2) are, in this sense, test-code counterparts of the classical code smells catalogued by Fowler [11], and should be read as refactoring targets rather than as isolated implementation mistakes. In a complementary direction, Bauer et al. synthesised code-review guidance for source and low-level test code into 33 review guidelines mapped to GUI-based testing artefacts, indicating that even correctly structured page objects require dedicated review practices distinct from those used for production code [19]. Most recently, Karagöz et al. investigated whether large language models (GPT-4o and DeepSeek Coder) can automatically generate page objects for existing web applications, reporting element-recognition rates above 70% but full-page accuracy of only 32.6–54.0% against manually written ground truth [20]. This result independently confirms, from a generative rather than an architectural angle, that a page object is not a mechanical locator dump but an artefact requiring judgment about operation boundaries and abstraction, and it marks an emerging research direction that a modular, well-specified POM architecture would make substantially easier to target automatically.

Component Object Model. Modern interfaces are assembled from recurring fragments – headers, menus, filters, tables, cards, alerts, modals – that appear in multiple contexts. Modelling an entire page as one indivisible object duplicates these fragments or forces a complex base-page hierarchy. A component object represents a reusable fragment scoped to a root locator (for example, `NotificationPanel.expectSuccess(message)`), independent of the page(s) that host it. Pages then become compositions: a `CataloguePage` contains a `Header`, a `FilterPanel`, a `ProductGrid` (itself composed of `ProductCard` instances), and a `NotificationPanel`. Component modelling also supports responsive interfaces, where a desktop navigation bar and a mobile menu implement the same behavioural contract behind different locators, and a fixture or page object selects the correct implementation for the active project.

Screenplay Pattern. The Screenplay pattern organises tests around actors, abilities, tasks, interactions, and questions: a test describes what an actor attempts and observes, rather than calling page-object methods. Serenity BDD presents Screenplay as a user-centred approach supporting readable, scalable, and maintainable automation [3, 4]. An actor may combine web, API, and database abilities within a single task (for example, `CreateOrder`) without that task being owned by one page, which is advantageous for workflows spanning many pages or channels. The cost is additional conceptual and structural complexity: teams must manage a larger vocabulary and many small classes or functions. POM and Screenplay are therefore not mutually exclusive – page and component objects can serve as lower-level adapters used by actor tasks, while a POM-based framework may adopt Screenplay-like business tasks above the page layer without implementing the full pattern.

Recent empirical research on test automation architecture and maintainability. Beyond pattern-level literature, several 2022–2026 empirical studies bear directly on the claims made in this paper. Coppola and Alégroth conducted a systematic literature review of the GUI-based testing research domain and compiled a taxonomy of 55 metrics across 17 categories, motivated by low reproducibility and comparability among published studies caused by the absence of commonly accepted measurement practices [21]. This taxonomy is a natural reference point for the future empirical validation outlined in the Research Analysis section, and it supports the decision, taken in this paper, to present the change-propagation, cohesion, reuse, and maintenance-cost relationships explicitly as qualitative indicators rather than validated metrics. On the language side, Bogner and Merkel compared the software quality of 604 paired JavaScript and TypeScript GitHub projects and found that TypeScript projects showed better code quality and understandability by several static indicators, yet counter-intuitively exhibited a higher bug-fix commit ratio and slightly longer bug-fix times than their JavaScript counterparts [22]. This nuances the treatment of TypeScript contracts below: static typing measurably reduces a category of integration errors, but it does not by itself guarantee faster defect resolution, so typed contracts should be read as complementary to, not a substitute for, architectural discipline. Finally, Nama surveyed extensions of POM and related creational/structural patterns (dynamic POM variants, Factory, Singleton, Façade, Composite, Strategy) for large, multi-tenant test suites and reported case-study reductions in maintenance cost and execution time after adopting a dynamic POM variant, although the evaluation was a single-company case study rather than a controlled experiment [23].

Table 1 compares the three architectural approaches surveyed above; the ratings are qualitative and represent typical implementation tendencies rather than universal properties.

Table 1

Comparative characteristics of test automation patterns

Criterion	Page Object Model	Component Objects	Screenplay
Primary abstraction	Page or view	Reusable UI fragment	User goal and capability
Main unit of reuse	Page operation	Component behaviour	Task, interaction, question
Learning curve	Low to medium	Medium	Medium to high
Suitability, small suites	High	Medium	Low to medium
Suitability, large/cross-channel suites	Medium	Medium to high	High
Risk of oversized modules	High if uncontrolled	Lower	Low per task, but many modules
Composition support	Medium	High	High
Implementation complexity	Low	Medium	High

POM offers the most direct transition from procedural UI tests to modular architecture; component objects refine its granularity and should be treated as complementary; Screenplay changes the central metaphor of the framework and becomes particularly valuable when business tasks, personas, or multiple interaction channels dominate the test design. Selenium documentation explicitly frames such practices as context-sensitive guidelines rather than universal "best practices" [5], and the same principle applies to POM: its value must be evaluated through responsibilities and change patterns, not through the mere presence of page classes.

Thus, despite the substantial body of literature on POM as a design pattern and the growing body of empirical work on GUI test maintenance, test smells, and code review [18, 19, 21], a systematic architectural account of POM as a layered system – one that explicitly relates page objects to component objects, fixtures, services, data factories, and typed contracts, and quantifies the resulting change-propagation, cohesion, reuse, and maintenance-cost trade-offs – remains largely absent from both the practitioner-oriented and the academic literature. This is the gap addressed in the remainder of this paper.

Presentation of the Main Research Material

Research Methodology. The study uses a qualitative, design-science-oriented methodology consisting of four stages. First, the comparative analysis presented above identifies the concepts, strengths, and limitations of POM, Component Objects, and Screenplay, focusing on architectural consequences rather than syntax. Second, architectural modelling defines a layered POM framework and specifies dependency rules between tests, page objects, component objects, fixtures, services, data factories, and utilities [17]. Third, a case illustration demonstrates the model using Playwright and TypeScript; the example is not presented as a controlled industrial experiment but as an architectural illustration derived from practical framework design. Fourth, a qualitative evaluation applies maintainability criteria – consistent with the maintainability quality characteristic defined in ISO/IEC 25010 [13] – to compare a script-oriented structure with modular POM: localisation of interface knowledge, cohesion of modules, explicitness of dependencies, potential for reuse, change propagation, test readability, support for isolated execution, and complexity introduced by abstraction.

The independent architectural variable is the degree of modularisation; the dependent qualities are maintainability and reusability. Because the paper does not report controlled measurements from multiple projects, causal performance claims are avoided; instead, the proposed indicators define a foundation for future empirical validation, for which the metrics taxonomy of Coppola and Alégroth [21] provides a ready-made measurement vocabulary.

Modularity Principles in POM Implementation. Separation of test intent and interaction mechanics is the starting principle: a test scenario should describe observable behaviour, not the implementation details of controls. A non-modular fragment mixes navigation, element identification, workflow execution, and verification in one place:

```
test("registered user can sign in", async ({ page }) => {
  await page.goto("/login");
  await page.locator("#email").fill("user@example.com");
  await page.locator("#password").fill("secret");
  await page.locator("button[type='submit']").click();
  await expect(page.locator("h1")).toHaveText("Dashboard");
});
```

A modular variant delegates interaction to page objects while keeping the assertion – the test oracle in the sense of Meszaros [10] and Ammann and Offutt [12] – inside the test itself:

```
test("registered user can sign in", async ({ loginPage, dashboardPage }) => {
  await loginPage.open();
  await loginPage.signIn({ email: "user@example.com", password: "secret" });
  await expect(dashboardPage.heading).toHaveText("Dashboard");
});
```

Moving every assertion into page objects can improve short-term brevity but hides the test oracle and makes failures harder to interpret. A practical responsibility rule is: tests define expectations and business flow; page objects define page interaction and page state; component objects define reusable-fragment interaction; fixtures define construction, setup, and teardown; services define non-UI system interaction; data factories define test-data creation.

Encapsulation of interface knowledge follows the same logic: a Playwright locator is not merely a string but a dynamic object connected to auto-waiting and retryability, and the tool recommends locating elements through user-visible semantics such as roles, labels, and text [1, 15]. A page object centralises the page path, the semantic identity of its controls, the order of interactions needed for an operation, and the type of accepted input – while it must not generate credentials, call an authentication API, or decide what post-login result should be expected.

Cohesion and controlled class size determine whether centralisation actually helps. Page boundaries should reflect a coherent interaction area: methods in one page object should operate on substantially overlapping state. Cohesion can be considered qualitatively through the ratio

$$C_m = R/N, \tag{2}$$

where R is the number of method pairs that operate on related controls or state and N is the total number of possible method pairs; formula (2) is not an exact universal metric, but it illustrates that cohesion decreases as unrelated behaviours accumulate in one class. Signals that a page object should be decomposed include repeated method-name prefixes (headerOpenMenu, headerCloseMenu), independent UI fragments with their own loading or error states, groups of locators reused on other pages, frequent merge conflicts between unrelated tests, constructors with many service dependencies, and a large proportion of methods irrelevant to most callers.

Composition over inheritance is the next principle. A narrow technical BasePage (for example, a waitForLoadingToFinish() helper) is acceptable; problems arise when inheritance is used to model composition, such as placing navigation, modal, notification, table, and form methods in BasePage, forcing every page to inherit operations it may not contain (Fig. 2). Composition keeps dependencies explicit: a DashboardPage instantiates a HeaderComponent and a NotificationComponent scoped to the regions that actually exist within it, so each component can be tested and maintained separately.

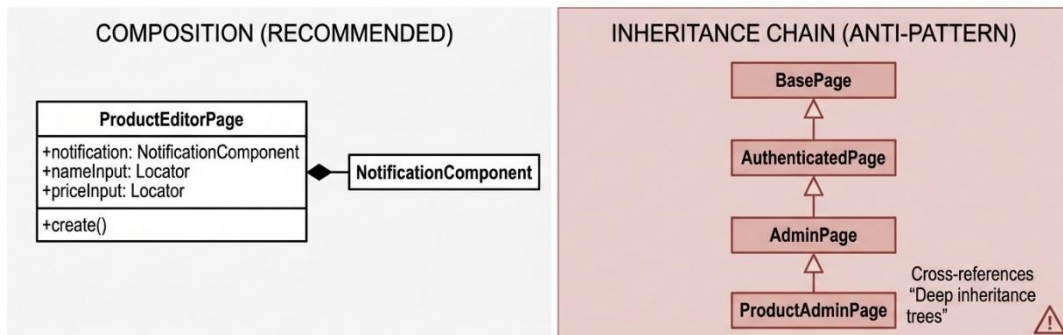


Fig. 2. Composition versus inheritance in POM class design

Explicit typed contracts complete the picture. TypeScript interfaces define compile-time contracts between modules [6, 7], and strict compiler options reduce implicit and nullable states [8]. Input models make operations self-documenting (interface Credentials { email: string; password: string; }); union types restrict supported variation; and behavioural interfaces such as Navigation allow interchangeable implementations. Typing does not remove runtime UI failures, but it eliminates a category of integration mistakes – a benefit confirmed, with important caveats regarding fix time, by the large-scale empirical comparison of JavaScript and TypeScript projects [22].

Dependency direction closes the set of principles: a modular architecture requires a stable dependency direction, summarised in Fig. 3 – tests depend on pages/business flows, which depend on components, which depend on Playwright abstractions. Fixtures construct these modules but should not become implicit global service locators; API services and data factories are parallel support modules that tests and fixtures may use directly. Page objects should not import test files, components should not import concrete pages, and utilities should not depend on business modules – circular dependencies indicate unclear boundaries.

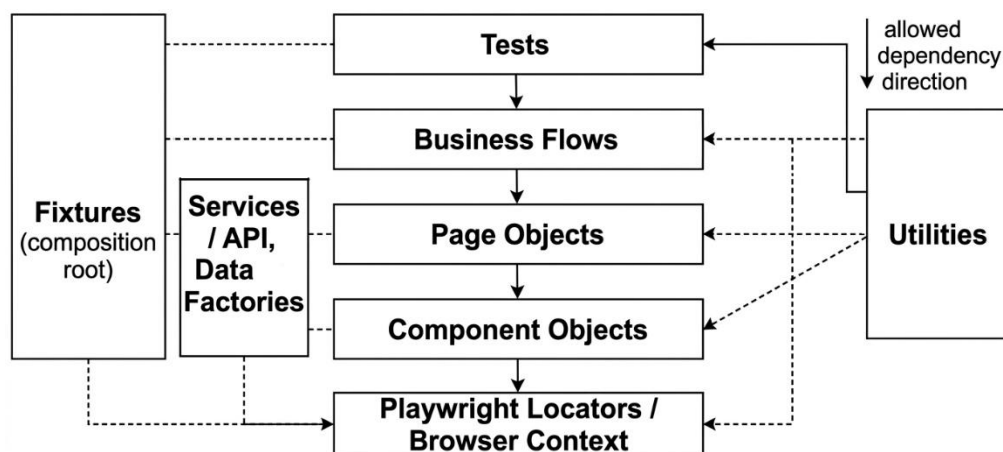


Fig. 3. Layered dependency architecture of a modular Playwright/TypeScript POM framework

Reuse at the correct abstraction level is the final principle: three reuse levels can be distinguished – interaction reuse (shared components and locator operations), workflow reuse (business-flow objects combining multiple pages), and data reuse (factories and builders generating valid input). For example, a CheckoutFlow may coordinate cart, address, delivery, and payment pages; it should not be forced into one page object because the workflow crosses navigation boundaries.

Case Illustration: Playwright and TypeScript Framework. The illustration concerns a web application with authentication, a dashboard, product management, and recurring interface elements. The framework must support desktop and mobile projects, UI test isolation, API-assisted data preparation, reusable page and component modules, parallel execution, and CI execution with diagnostic artefacts. Playwright provides browser contexts that isolate cookies and local storage between tests [16], while fixtures define reusable setup and teardown [14] – capabilities that align naturally with modular design, since construction and lifecycle concerns can be separated from page behaviour.

A representative project layout separates concerns by directory: tests/e2e (scenarios grouped by feature), pages (page objects such as login.page.ts, dashboard.page.ts, products/products.page.ts), components (header, notification, modal, product-card), flows (authentication.flow.ts, product-management.flow.ts), services (auth.api.ts, products.api.ts), fixtures (app.fixture.ts, data.fixture.ts), models (typed contracts such as credentials.ts, product.ts), data (factories, e.g. product.factory.ts), and utils.

The notification component is scoped to a region rather than the whole page and returns observable text, leaving the expected value to the test:

```
export class NotificationComponent {
  readonly message: Locator;
  constructor(private readonly root: Locator) {
    this.message = root.getByRole("status");
  }
  async text(): Promise<string> {
    return (await this.message.textContent())?.trim() ?? "";
  }
}
```

A page object composes such components rather than redeclaring their locators – for example, ProductEditorPage holds a NotificationComponent scoped to its own notification region, alongside its own nameInput, priceInput, and publishButton locators, and a create(product) convenience operation built from smaller fillProduct()/publish() methods.

Fixtures act as the composition root of the UI modules (Fig. 4), constructing page objects around the isolated page so that tests do not instantiate them manually, all objects share the same isolated context [16], constructors remain explicit and testable, and lifecycle management is centralised [14]:

```
export const test = base.extend<AppFixtures>({
  loginPage: async ({ page }, use) => { await use(new LoginPage(page)); },
  productsPage: async ({ page }, use) => { await use(new ProductsPage(page)); },
  productEditorPage: async ({ page }, use) => { await use(new ProductEditorPage(page)); },
});
```

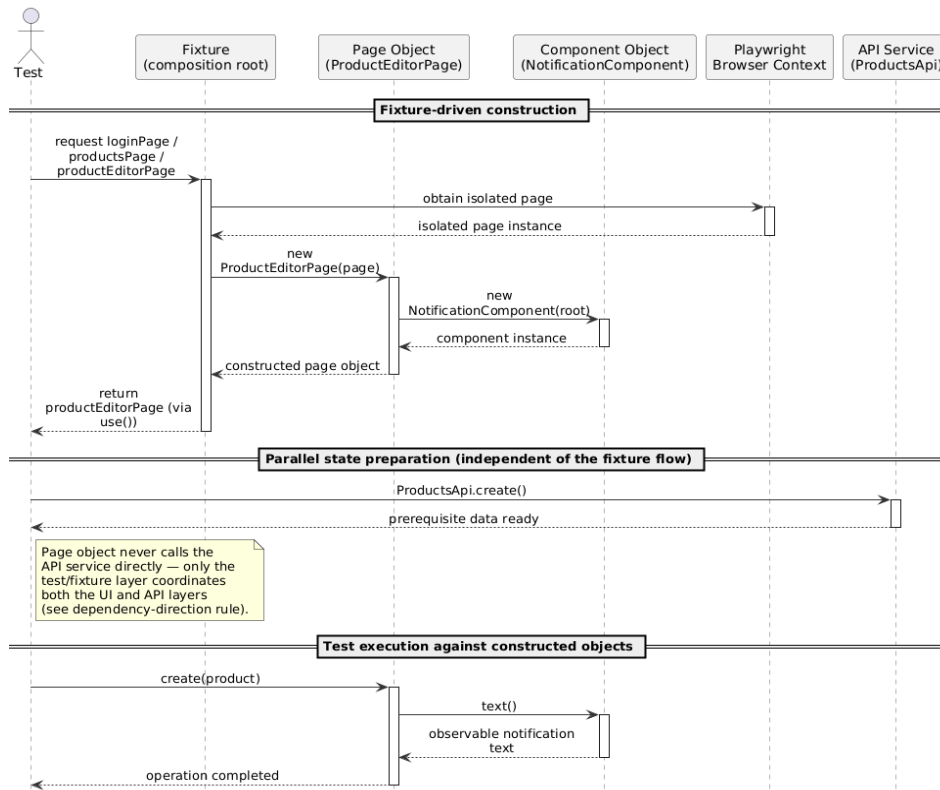


Fig. 4. Sequence of test instantiation through the fixture composition root

Creating prerequisite data through the UI can make tests slow and couple them to unrelated screens, so an API service (for example, `ProductsApi.create()/delete()`) prepares state instead. The page object must not import this service; the test or fixture coordinates both layers. A modular design cannot compensate for an unstable locator policy: Playwright ties locators to auto-waiting and retryability and recommends, in priority order: (1) role and accessible name; (2) label, placeholder, text, alternative text, or title; (3) a stable test identifier; (4) narrowly scoped CSS only when semantic locators are unavailable [1, 15]. A complete scenario then reads as intent rather than implementation:

```
test("administrator can publish a product", async ({
  authenticatedAdmin, productsPage, productEditorPage,
}) => {
  const product = buildProduct();
  await productsPage.open();
  await productsPage.startCreatingProduct();
  await productEditorPage.create(product);
  await expect(productEditorPage.notification.message)
    .toHaveText("Product published successfully");
  await expect(productsPage.linkToProduct(product.name)).toBeVisible();
});
```

Impact on Maintainability and Reusability. Change localisation can be estimated for n tests submitting the same form, each containing k locators. A script-oriented suite has approximately

$$L_s = n \cdot k \quad (3)$$

locator occurrences. A page-object suite centralises them to

$$L_p = k + n \cdot c, \quad (4)$$

where c is the number of calls to stable page operations that do not encode selector details. Comparing (3) and (4), a selector change then requires editing one module rather than up to n call sites, provided page objects are genuinely shared and tests do not bypass them with direct selectors.

Reuse is captured by a possible indicator

$$R_m = N_c / N_m, \quad (5)$$

where N_c is the number of distinct caller modules and N_m the number of public methods of the reused module. A high value of R_m in (5) may indicate effective reuse, but should be interpreted together with cohesion (2), since a large "utility" class may have many callers yet still represent poor design. Component objects typically provide the highest reuse, since recurring fragments cross page boundaries; page operations provide reuse within a feature area; business flows provide reuse across navigation sequences.

Readability and diagnostic value follow from the same design: POM improves readability when method names express user actions and reduces it when methods encode vague implementation sequences.

Maintenance cost is captured by a conceptual model

$$M = F \cdot (E + A + V), \quad (6)$$

where F is the frequency of relevant interface changes, E the effort to identify affected code, A the adaptation effort, and V the verification effort after adaptation. POM primarily reduces E in (6) by localising knowledge and can reduce A when operations remain stable; it may also reduce V because the affected modules and scenarios are clearer. However, excessive abstraction increases navigation effort through the codebase and may itself increase E .

Scalability is thus not provided by POM alone: as a suite grows, scalability requires component objects for recurring fragments, domain folders for feature ownership, flow objects for cross-page tasks, API services for state preparation, typed factories for data, fixtures for lifecycle composition, linting rules for architectural consistency, and CI projects for browser and environment variation.

Architectural Anti-Patterns. Table 2 summarises the recurring POM anti-patterns discussed in the analysis above. Each represents a way in which page objects can technically exist while failing to create genuine modularity, and each corresponds to a test-code counterpart of the code smells originally catalogued by Fowler [11].

Research Analysis. The analysis supports treating POM as an architectural layer rather than a locator-storage convention. Regarding RQ1, POM reduces change propagation through localisation (1), cohesive interfaces (2), and dependency inversion from tests toward stable behavioural operations (Fig. 3); selector centralisation is necessary but not sufficient. Regarding RQ2, POM offers the lowest implementation complexity and the most direct mapping to pages (Table 1); Component Objects improve POM through finer-grained composition and reuse; Screenplay offers stronger user-goal modelling and cross-channel scalability but introduces a larger conceptual framework. Regarding

RQ3, Playwright locators, isolated contexts [16], fixtures [14], and diagnostics align naturally with modular POM, while TypeScript adds explicit contracts and compile-time checking of module interaction, though with the fix-time caveat reported in [22]. Regarding RQ4, the principal risks are low cohesion, hidden dependencies, excessive inheritance, assertion opacity, and mixing UI interaction with setup or business services (Table 2), consistent with the test-smell and code-review findings in [18, 19].

Table 2

POM architectural anti-patterns

Anti-pattern	Manifestation	Consequence
Locator-only page objects	Class exposes public locators but no meaningful operations	Tests remain tightly coupled to interaction order; complement with cohesive operations (e.g. <code>signIn()</code>)
God page object	One class accumulates every operation available after authentication	Becomes an application-wide service locator and violates cohesion (2); decompose by page, component, or feature
Assertions hidden in every action	Methods such as <code>clickSave()</code> silently assert unrelated conditions	Reduces scenario transparency and prevents reuse in negative tests; keep outcome-specific assertions in tests
Deep inheritance trees	<code>BasePage > AuthenticatedPage > AdminPage > ProductAdminPage</code> (Fig. 2)	Couples pages through inherited implementation; composition and interfaces are generally more adaptable
Cross-page instantiation	A page method constructs and returns the next page object internally	Creates hidden lifecycle and navigation coupling; a deliberate, fixture-based construction policy (Fig. 4) is clearer
Business logic inside UI adapters	Page objects compute discounts, generate random users, query a database	Mixes concerns that change for reasons unrelated to the interface; belongs in separate services (Fig. 3)
Generic utility expansion	A <code>CommonPage</code> or <code>Helpers</code> module collects unrelated methods	Reuse based solely on syntactic similarity creates unstable dependencies

The proposed model has limitations. It is a qualitative architectural analysis and case illustration, not a multi-project controlled experiment. Formulas (1)–(6) describe relationships and possible indicators; they are not empirically calibrated prediction models, and their operationalisation could draw on the metrics taxonomy of Coppola and Alégroth [21]. The example application is intentionally generic and does not represent all interface types, such as canvas-based editors or real-time collaborative applications. A future empirical study could compare two implementations of the same suite – direct Playwright scripts and modular POM – measuring changed files, changed lines, repair time, and failure rate after interface modifications.

Conclusions and Prospects for Further Research

The Page Object Model remains relevant to modern browser automation, but its value depends on architectural interpretation. A page object should provide a cohesive interface to a page or view, hide change-sensitive interaction details, and expose operations that correspond to stable user behaviour; merely moving selectors into classes does not create modularity. A maintainable Playwright and TypeScript framework can use POM as the central abstraction layer while assigning recurring UI fragments to component objects, setup and teardown to fixtures, system-level interactions to API services, data construction to factories, and cross-page workflows to flow modules, as illustrated by the layered architecture in Fig. 3 and formalised through indicators (1)–(6).

The comparative analysis (Table 1) shows that POM has a favourable balance between accessibility and structural benefit; Component Objects strengthen its composition and reuse; Screenplay provides a more comprehensive user-goal model for complex multi-channel systems at the cost of additional overhead. Recent empirical literature on GUI test maintenance, code review, and test smells [18, 19, 21] confirms, independently of this paper's own case illustration, that the mere presence of page objects does not guarantee their quality – a finding fully consistent with the anti-pattern catalogue in Table 2.

The principal practical conclusion is that POM should be evaluated by change localisation, cohesion, explicit dependencies, and reuse rather than by the existence of page classes. Further research should formalise and empirically validate architectural quality indicators for automated test code, building on the metrics taxonomy of [21] and the TypeScript quality findings of [22]; relevant directions include a controlled multi-project comparison of script-oriented and modular POM suites, an assessment of LLM-assisted page-object generation and maintenance [20] within a layered POM architecture, and an extension of the anti-pattern catalogue (Table 2) into an automatically checkable linting ruleset in the spirit of [18].

ADDITIONAL INFORMATION

AUTHOR CONTRIBUTIONS

Conceptualization, Y.P.; methodology, Y.P. and O.K.; formal analysis (change-propagation, cohesion, and maintenance-cost indicators, formulas (1)–(6)), Y.P.; software (case illustration, Playwright/TypeScript implementation), O.K.; investigation, O.K.; related-work analysis, Y.P. and O.K.; writing – original draft preparation, O.K.; writing – review and editing, Y.P.; visualization, O.K.; supervision, Y.P.; project administration, Y.P. All authors have read and agreed to the published version of the manuscript.

DECLARATION ON THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE TOOLS

In preparing this work, the authors used Google Translate and LanguageTool for grammar and spelling checks, paraphrasing, and rephrasing, and used an AI assistant, within the limits permitted by the institutional AI use policy, for translation support, grammar checking, literature search, and debugging of program code. After using these tools/services, the authors reviewed and edited the content and take full responsibility for the content of this publication.

REFERENCES

1. Microsoft. Locators. Playwright Documentation. URL: <https://playwright.dev/docs/locators>
2. Selenium Project. Page object models. Selenium Documentation. URL: https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/
3. Serenity BDD. Your first Screenplay scenario. Serenity BDD User Manual. URL: <https://serenity-bdd.github.io/docs/tutorials/screenplay>
4. Serenity BDD. Screenplay fundamentals. Serenity BDD User Manual. URL: https://serenity-bdd.github.io/docs/screenplay/screenplay_fundamentals
5. Selenium Project. Test practices: Encouraged behaviors. Selenium Documentation. URL: https://www.selenium.dev/documentation/test_practices/encouraged/
6. Microsoft. Object types (interfaces). TypeScript Documentation. URL: https://www.typescriptlang.org/docs/handbook/2/object_types.html
7. Microsoft. Classes. TypeScript Documentation. URL: <https://www.typescriptlang.org/docs/handbook/2/classes.html>
8. Microsoft. The basics. TypeScript Documentation. URL: <https://www.typescriptlang.org/docs/handbook/2/basic-types.html>
9. Fowler M. PageObject. martinowler.com. URL: <https://martinowler.com/bliki/PageObject.html>
10. Meszaros G. xUnit Test Patterns: Refactoring Test Code. Boston: Addison-Wesley, 2007. 883 p.
11. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston: Addison-Wesley, 2018. 448 p.
12. Ammann P., Offutt J. Introduction to Software Testing. 2nd ed. Cambridge: Cambridge University Press, 2016. 332 p.
13. ISO/IEC 25010:2023. Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – Product quality model. Geneva: ISO, 2023. URL: <https://www.iso.org/standard/78176.html>
14. Microsoft. Fixtures. Playwright Documentation. URL: <https://playwright.dev/docs/test-fixtures>
15. Microsoft. Auto-waiting. Playwright Documentation. URL: <https://playwright.dev/docs/actionability>
16. Microsoft. Isolation. Playwright Documentation. URL: <https://playwright.dev/docs/browser-contexts>
17. Fulcini T., Garaccione G., Coppola R., Ardito L., Torchiano M. Guidelines for GUI testing maintenance: a linter for test smell detection. Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST 2022). Singapore, 2022. P. 17–24. <https://doi.org/10.1145/3548659.3561306>
18. Bauer A., Coppola R., Alégroth E., Gorschek T. Code review guidelines for GUI-based testing artifacts. Information and Software Technology. 2023. Vol. 163. Art. 107299. <https://doi.org/10.1016/j.infsof.2023.107299>
19. Karagöz B., Ricca F., Biagiola M., Stocco A. Towards automated page object generation for web testing using large language models. Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation (ICST '26). 2026. <https://doi.org/10.48550/arXiv.2602.19294>
20. Coppola R., Alégroth E. A taxonomy of metrics for GUI-based testing research: A systematic literature review. Information and Software Technology. 2022. Vol. 152. Art. 107062. <https://doi.org/10.1016/j.infsof.2022.107062>
21. Bogner J., Merkel M. To type or not to type? A systematic comparison of the software quality of JavaScript and TypeScript applications on GitHub. Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22). 2022. P. 658–669. <https://doi.org/10.1145/3524842.3528454>
22. Nama P. Leveraging design patterns in automation testing: A paradigm shift beyond conventional approaches. American Journal of Computer Architecture. 2024. Vol. 11, No. 4. P. 42–47. URL: <http://article.sapub.org/10.5923.j.ajca.20241104.02.html>

Юрій ПОНОЧОВНИЙ, Олексій КУДЕНКО

Полтавський державний аграрний університет

ПАТЕРН PAGE OBJECT MODEL У МОДУЛЬНІЙ АРХІТЕКТУРІ АВТОТЕСТІВ

Сучасні вебдодатки розвиваються через короткі цикли розробки та часті переробки інтерфейсу, що часто робить набори тестів інтерфейсу користувача крихкими, оскільки сценарії безпосередньо залежать від структури сторінки, селекторів, часу та логіки навігації. У статті розглядається модель об'єктів сторінок (POM) як архітектурна основа для модульної автоматизації тестування. POM порівнюється зі шаблоном Screenplay та підходом Component Object. Запропонована інтерпретація розрізняє сценарії тестування, інтерфейси сторінок, об'єкти компонентів повторного використання, фікстури, сервіси, фабрики даних та утиліти як окремі архітектурні шари, формалізуючи поширення змін, зв'язність, повторне використання та вартість обслуговування за допомогою пронумерованих індикаторів. Розглянуто контракти TypeScript, впровадження залежностей, локатори Playwright, фікстури та ізольовані контексти браузера як засоби підтримки явних залежностей та передбачуваних меж модулів. Якісна модель підтримки, що відповідає ISO/IEC 25010, пояснює, як централізація знань про інтерфейс зменшує поширення змін. POM покращує зручність обслуговування, коли відповідальність обмежується взаємодією інтерфейсу та представленням стану; надрозмірні класи, методи з інтенсивними твердженнями, глибоке успадкування та неконтрольовані міжсторінкові залежності перетворюють POM на джерело зв'язування. Модульну реалізацію POM розглянуто як адаптивний архітектурний рівень, ефективність якого залежить від зв'язності, явних контрактів, композиції та контрольованої абстракції.

Ключові слова: Page Object Model, архітектура автоматизованого тестування, модульність, Playwright, TypeScript, UI-тестування, супроводжувальність, компонентні об'єкти.