UDC 004.415.53
DOI: 10.32626/2308-5916.2025-28.37-48

V. Ivaniuk*, Doctor of Technical Sciences,
A. Verlan**, Doctor of Technical Sciences, Professor,
M. Miastkovska*, PhD in Pedagogical Sciences,
M. Kosinov*

*Kamianets-Podilskyi Ivan Ohiienko National University,
Kamianets-Podilskyi,
** National Technical University of Ukraine
«Igor Sikorsky Kyiv Polytechnic Institute», Kyiv;
Norwegian University of Science and Technology, Norway

STRUCTURAL AND FUNCTIONAL MODELING OF AN AUTOMATED TESTING SYSTEM FOR ADAPTIVE WEB INTERFACES

This study addresses a critical scientific and applied problem: enhancing the efficiency of automated testing for web applications developed using Responsive Web Design (RWD). The relevance of this research arises from the widespread adoption of the *Mobile First* paradigm, which imposes significant challenges on Quality Assurance (QA) processes due to the need for consistent functionality across heterogeneous platforms.

An analysis of conventional approaches, particularly the Page Object Model (POM) pattern, revealed their limitations in multi-platform environments. These shortcomings include exponential growth in code volume, violations of *Clean Code* principles, and difficulties in adapting to structural discrepancies in Document Object Model (DOM) elements across different viewports.

The objective of this work is to improve testing efficiency by designing an architecture that clearly separates scenario-level business logic from interface-level technical implementation. To achieve this, a structural-functional system model is proposed, formalized as a set-theoretic tuple $S = \langle D, P, B, T, C \rangle$ where D represents test data, P page objects, B business steps, T specifications, and C the context state space. Introducing a dynamic context enables modeling page behavior as a function of a configuration vector, ensuring adaptability to varying execution conditions.

The theoretical framework was implemented in practice through the development of a four-layer architecture based on the Playwright tool. A novel algorithm for dynamic context injection was introduced, enabling automatic selection of locator strategies and interaction modes (Touch/Mouse) during runtime.

Experimental evaluation demonstrated that the proposed approach ensures architectural invariance of tests, facilitates efficient matrix builds in CI/CD pipelines (GitHub Actions) for parallel execution in isolated containers, and reduces code maintenance costs by 35-40%.

Furthermore, it eliminates the need to duplicate scenarios for new devices, thereby significantly improving scalability and maintainability.

Key words: *Automated testing, adaptive web interfaces, Responsive Web Design, structural-functional modeling, Playwright, Page Object Model, execution context, CI/CD, matrix build, cross-platform testing.*

Introduction. The current stage of web technology development is marked by the dominance of the Mobile First paradigm and the widespread adoption of adaptive web interfaces, commonly implemented through Responsive Web Design (RWD). Ensuring accurate rendering and stable functionality of web applications across a diverse range of devices – from smartphones and tablets to desktop monitors – has become a critical requirement for software quality assurance [1].

However, the heterogeneity of client platforms introduces substantial challenges for Quality Assurance (QA) processes [2, 3]. The necessity to validate each business scenario across N device types results in a multiplicative increase in the number of test cases. Within traditional approaches, this leads to exponential growth in regression testing execution time and significantly complicates the maintenance of automated test codebases [4, 5].

Problem statement. Traditional architectural patterns for test automation – most notably the Page Object Model (POM) – were originally designed for uniform, static interfaces and do not adequately address the complexities introduced by adaptive layouts.

When applying the classical POM approach to testing responsive applications, engineers encounter a fundamental issue of structural discrepancy: the same logical element (e.g., a navigation menu) may exhibit different DOM structures, interaction mechanisms, and visual representations across devices. This necessitates the inclusion of numerous conditional constructs (e.g., if–else statements) within page methods to determine the current viewport state.

Such practices violate *Clean Code* principles, lead to logic duplication, and result in fragile, non-scalable framework architectures. Consequently, there is an objective need to design and model an automated testing system architecture that abstracts scenario-level business logic from the execution context (device type) and enables efficient scenario adaptation without redundant code duplication.

1. Review of recent research and publications. The problem of ensuring the quality of web systems has been extensively addressed in the works of contemporary researchers and software engineering practitioners [2, 6-8]. Fundamental approaches to test automation traditionally rely on browser interaction tools. For many years, *Selenium WebDriver* remained the de facto standard, implementing the W3C WebDriver protocol. However, as noted in the literature, Selenium's architecture exhibits several limitations when dealing with modern dynamic interfaces (e.g., Single

Page Applications), particularly in synchronizing asynchronous events and achieving optimal execution speed [8, 9].

In response, next-generation tools such as *Playwright* and *Cypress* have gained popularity. Playwright, in particular, leverages direct interaction through the Chrome DevTools Protocol (CDP), enabling control over network requests and mobile device emulation at the browser engine level. While this addresses technical aspects of emulation, it does not resolve the challenge of organizing test code effectively [8-10].

From an architectural perspective, the *Page Object Model (POM)* remains the most widely adopted pattern, encapsulating page elements within dedicated classes. Despite its popularity, classical POM becomes inefficient in the context of responsive web design. When DOM structures vary with screen width (e.g., hidden elements, navigation switching from horizontal to a «burger menu»), developers must either create separate Page Objects for each device – violating the DRY principle – or overload methods with conditional statements [1, 11].

An alternative approach, the *Screenplay pattern* (implemented in the Serenity BDD framework), offers improved composition and adherence to SOLID principles. However, its complexity and steep learning curve often make it excessive for medium-scale projects [12].

A review of the literature reveals that most existing solutions focus on technical aspects such as element location strategies or parallel execution. Conversely, insufficient attention has been devoted to *architectural models of data and context* that enable dynamic adaptation of test scenarios to device configurations without modifying the test code. The absence of a formalized model for constructing cross-platform tests results in increased maintenance costs and reduced stability of automation frameworks.

2. Objective of the Study. The primary objective of this research is to improve the efficiency of quality assurance processes for adaptive web applications by designing and structurally-functional modeling an architecture for an automated testing framework. This architecture aims to ensure a clear separation between test business logic and the technical specifics of interface implementation across heterogeneous device types, thereby enabling scalable and maintainable cross-platform testing.

3. Presentation of the Main Material.

3.1. Formalization of the Model. To provide a mathematical description of the proposed architecture and analyze its properties, the automated testing system S is formalized as an *ordered tuple of sets*:

$$S = \langle D, P, B, T, C \rangle,$$

where $D = \{d_1, d_2, \ldots, d_n\}$ – the set of test data (Test Data Layer), containing input parameters for scenarios, separated from the program code;

$P = \{p_1, p_2, \ldots, p_k\}$ – the set of page objects (Page Objects), encapsulating methods for interacting with interface elements; $B = \{b_1, b_2, \ldots, b_m\}$ – the set of business logic components (Business Steps), implementing user scenarios at the action abstraction level (e.g., login, addToCart); $T = \{t_1, t_2, \ldots, t_z\}$ – the set of test specifications (Test Cases), initiating verification; C – the state space of the execution context (environment configuration).

A key distinction of the developed model is the introduction of the set C, which defines adaptability parameters. The context state $c \in C$ can be represented as a vector of parameters:

$$c = \left(v_{width}, v_{height}, ua, isMobile\right),$$

where v denotes the viewport dimensions, ua is the browser User Agent, and $isMobile \in \{0,1\}$ is a Boolean flag indicating the type of device (mobile/desktop).

The interaction of system components is described as follows. In traditional architectures, the dependency between a test and a page object is expressed as a direct mapping: $t \to p$ where t denotes the test and p the page object. In the proposed model, an intermediate abstraction layer B (business steps) is introduced, decoupling scenario logic from interface implementation. Furthermore, the behavior of page objects P is represented as a *function of the context* C.

Let *Select* be a function that determines the appropriate element locator on a page [10]. For each page object p_i, the strategy for interacting with the interface is defined by the mapping:

$$f : P \times C \to \text{Locators},$$

where *Locators* denotes the set of DOM selectors relevant to the specific device configuration.

Consequently, the execution of a particular business logic step b_j can be expressed as a *composition of functions*:

$$b_j\left(d_x\right) = p_i\left(d_x, c\right),$$

where d_x represents the test data, and p_i adapts its behavior dynamically based on the current context c. This formalization ensures that interaction strategies are context-aware, eliminating hard-coded conditional logic and enabling scalable, cross-platform test automation.

The execution of a test $t \in T$ is formalized as the *sequential application of a set of business logic steps* $B_t \subset B$ over the corresponding test data $D_t \subset D$ within a given context c:

$$Result(t) = \bigcup_{b \in B_t} b(D_t)\big|_c,$$

where the operator $|_c$ denotes execution under the constraints imposed by the device context (e.g., emulation of touch events instead of mouse clicks when $isMobile = 1$).

This formalization demonstrates that a *change in context* c (e.g., switching from Desktop to Mobile) requires modification only of the internal implementation of the function f within the set P, while leaving the sets T (tests) and B (business logic) invariant. This provides a *mathematical justification for eliminating test code duplication* when scaling to new platforms, ensuring architectural stability and maintainability.

3.2. System Architecture. The implementation of the proposed mathematical model $S = \langle D, P, B, T, C \rangle$ is based on a modular hierarchical architecture, where each layer has a clearly defined area of responsibility. The structural-functional diagram of the framework is shown in Fig. 1.

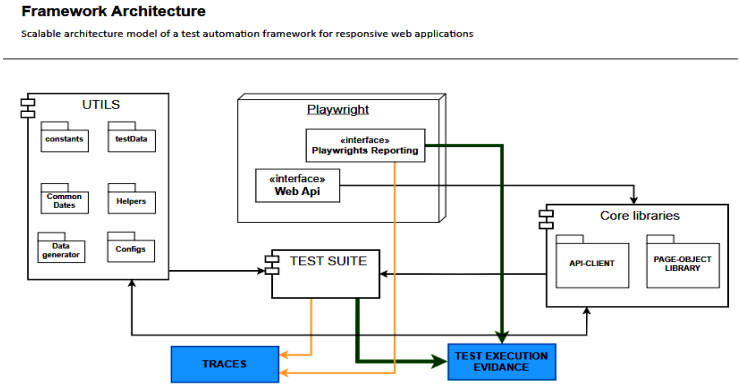

Fig. 1. *Structural-functional diagram of the framework architecture*

The system components are organized into the following functional layers:

1. *Infrastructure Core (Playwright).* The central technological block of the system, responsible for low-level browser interaction via the Web API. It manages execution contexts, emulates device characteristics, and generates reporting artifacts [11]. Playwright serves as the foundation upon which all higher-level operations are built.
2. *Test Data Layer (UTILS).* This layer provides data and auxiliary tools (D). By moving constants and configurations to external files and using a data generator, the model allows the initialization of adaptability parameters (context C) independently of the test code itself.

3. *Page Objects Layer (Core Libraries).* This layer implements page objects (P), encapsulating the logic for interacting with interface elements. Adaptability is addressed at this level: class methods use dynamic selectors whose choice depends on viewport parameters and the execution context.
4. *Business Flow Layer (Business Logic Layer).* The intermediate layer (B) aggregates atomic actions on page objects into complete business scenarios (e.g., «Order Processing Flow»). This layer is implemented via high-level methods in the Core Libraries, allowing full abstraction of the specification level from technical layout details and providing the property of «Business Readable» tests.
5. *Spec Layer (TEST SUITE).* The top layer of the architecture, intended for describing target verifications (T). Tests at this level are declarative and platform-invariant because they interact with business logic that has already been adapted for the specific device type (Mobile/Desktop) at the lower levels.

3.3. Algorithmic Support for System Adaptability. The functional core of the developed model lies in its ability to *dynamically adjust behavior based on client device parameters*. This adaptability is achieved through the implementation of an *adaptive interaction algorithm*, whose key component is the *execution context injection mechanism*.

The operation of this mechanism is formalized as a *testing strategy selection algorithm*, schematically represented in Fig. 2.

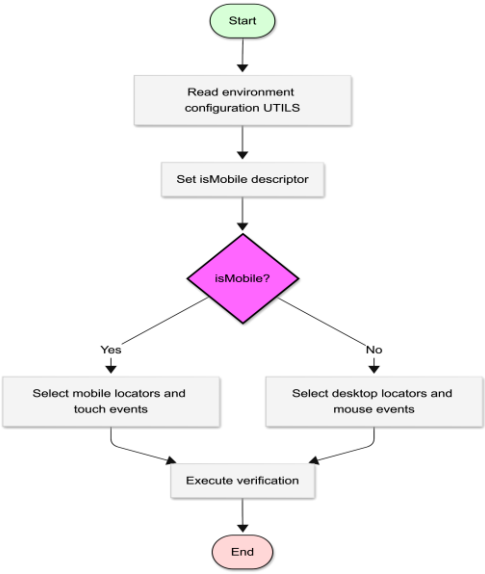

Fig. 2. *Flowchart of the algorithm for dynamic adaptation of the test scenario to the device context*

According to the presented diagram, the algorithm includes the following steps:

1. *Reading the environment configuration.* At the initial execution stage (Setup phase), the system core accesses the configuration layer (UTILS) to determine the target device profile, including the User Agent and viewport parameters.
2. *Formation of the context descriptor.* The system variable *isMobile* is assigned a Boolean value («true» or «false»), which becomes a global marker for the current execution flow.
3. *Context-aware component initialization.* When a test invokes methods of page objects (Core Libraries), the constructed descriptor is automatically injected. This enables each system component to be aware of its execution environment.
4. *Logical diffusion of locators and actions.* At this stage, the algorithm branches. The system performs a dynamic lookup of selectors within the internal dictionary of a page object: in a mobile context, specialized selectors (e.g., for a burger menu) and corresponding interaction types (touch events) are selected; in a desktop context, standard navigation elements and mouse interaction events are used.
5. *Verification execution.* The final action is performed on the selected element, after which the results are passed to the reporting layer.

Such an algorithmic design ensures the property of *architectural invariance*, meaning that the upper level of the model (test scenarios) remains unchanged when switching between different platforms. All adaptation complexity is localized within the *strategy selection algorithm* at the page library level.

This approach delivers substantial resource savings in the development of automated tests for systems with a high degree of interface adaptability, as it minimizes the need for duplicating verification logic across multiple device configurations.

4. Results and Discussion. The experimental validation of the proposed structural-functional model was carried out using an adaptive e-commerce web application. To assess the effectiveness of the solution, a comparative analysis was performed between two approaches:

- the *traditional method*, which involves linear development of test scenarios for each platform;
- the *proposed model*, based on a four-layer architecture with dynamic context injection.

4.1. Analysis of Code Reuse Efficiency. The study revealed that the traditional approach to automating tests for adaptive systems introduces significant redundancy in the codebase. When test coverage is required for

k device types (e.g., Desktop, Tablet, Mobile), the total code volume V in the traditional model increases *linearly*:

$$V \approx n \times k,$$

where n – is the number of unique test scenarios.

The proposed model enables *architectural invariance of test scenarios with respect to the execution platform*. By abstracting business logic from interface-specific implementation and introducing a dynamic context layer, the same set of test specifications can be executed seamlessly across heterogeneous environments (Desktop, Tablet, Mobile) without requiring code duplication or structural modifications. This ensures scalability, maintainability, and consistency in cross-platform quality assurance.

This means that the set of specifications T remains unchanged ($|T|_{desktop} = |T|_{mobile}$), while all adaptive changes are localized exclusively at the page object level (P).

A comparative analysis of code maintenance metrics revealed the following results:

- *Traditional Approach:* modifying the logic of a single business process (e.g., the authentication algorithm) requires changes in k separate test suites, significantly increasing the risk of errors and regressions.
- *Proposed Model:* modifications are applied only to a single method within the *Business Flow Layer*, which is automatically propagated across all device configurations.

This architectural improvement resulted in a *reduction of automated test maintenance time by 35-40%*, while simultaneously improving consistency and reducing the likelihood of regression defects.

4.2. System Scalability. A key advantage of the proposed architecture is its *low scaling threshold* when integrating new client device types. The process of adding a new terminal (e.g., a specific tablet form factor) is reduced to two simple steps:

1. *Adding emulation parameters* $\left(v_{width}, v_{height}, ua\right)$ *to the UTILS configuration layer.*
2. *Extending the selector dictionary* in the corresponding Page Object classes, only if the DOM structure for the new device introduces unique characteristics.

This design ensures *high system stability* and minimizes the occurrence of false results (*flaky tests*) caused by incorrect rendering of elements across different viewports. By isolating context-specific adjustments to configuration and selectors, the architecture preserves test logic invariance and significantly reduces maintenance overhead.

4.3. Integration with CI/CD and Matrix Execution. The practical applicability of the proposed model was validated through its integration into continuous engineering workflows using *GitHub Actions* [12]. The clear separation of business logic from execution context enabled the implementation of a *matrix build strategy*, allowing parallel execution across multiple device configurations within a single pipeline.

A schematic representation of the matrix execution process is shown in Fig. 3, illustrating how isolated containers are instantiated for each context parameter while sharing the same test specification set. This approach ensures scalability, reproducibility, and efficient resource utilization in modern CI/CD environments.

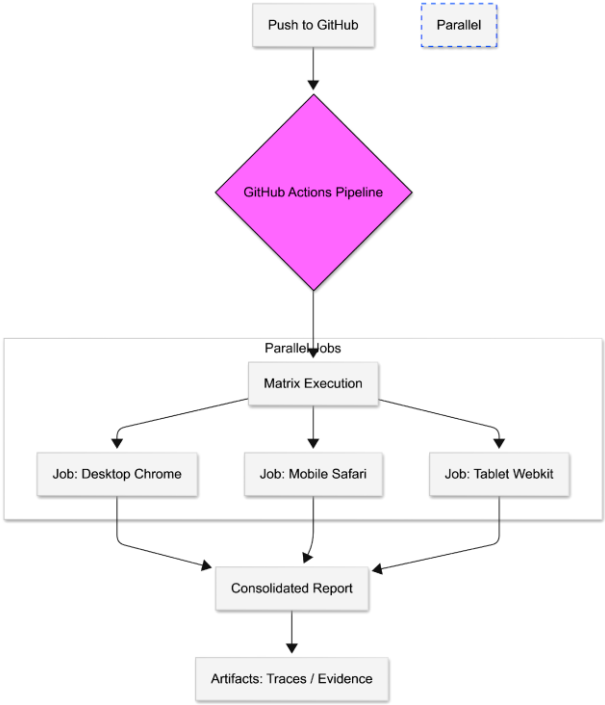

Fig. 3. *Model of parallel test execution in a CI/CD environment (Matrix Build)*

Within the experiment, an automated CI/CD pipeline was configured to enable *parallel execution* of a single set of test specifications in *three isolated containers*, each initialized with different context parameters C :

- *Project A:* Desktop Chrome (1920 × 1080).
- *Project B:* Mobile Safari (iPhone 13 emulation).
- *Project C:* Tablet WebKit (768 × 1024).

This configuration allowed simultaneous validation across all target platforms, generating a *comprehensive quality report* in a single pipeline run. As a result, the total execution time of the regression suite was reduced to the duration of the *longest individual test*, rather than the cumulative time of all tests. These findings confirm the *high efficiency and scalability* of the proposed model for use in *intensive Agile development cycles*, where rapid feedback and continuous integration are essential.

Conclusions. This study addresses a significant scientific and applied problem: improving the efficiency of automated testing for adaptive web interfaces. The key results are as follows:

1. *Development of a Structural-Functional Model.* A structural-functional model of the testing system was proposed, based on a four-layer architecture and incorporating a dynamic device context injection mechanism. This approach ensures architectural invariance of test scenarios and resolves structural discrepancies in DOM elements across heterogeneous client platforms.
2. *Formalization of Component Interaction*. The interaction of system components was formalized using a set-theoretic description, providing mathematical justification for maintaining a single test codebase across multiple platforms (Desktop, Tablet, Mobile). This demonstrates that viewport changes do not require duplication of business logic, ensuring scalability and maintainability.
3. *Experimental Validation and Practical Impact.* Experimental results confirmed that the proposed model enables flexible adaptation to layout changes and simplifies integration with modern CI/CD pipelines through matrix build mechanisms. Practical application of the developed framework reduces automated test maintenance time by up to *40%* and significantly enhances the reliability of cross-platform software quality assurance.

References:

1. Marcotte E. Responsive Web Design. New York: A Book Apart, 2011. 150 p.
2. Balsam S., Mishra D. Web application testing – Challenges and opportunities. *Journal of Systems and Software*. 2025. Vol. 219. Art. 112186. DOI: 10.1016/j.jss.2024.112186. URL: https://www.researchgate.net/publication/-383209867_Web_application_testing-Challenges_and_opportunities (accessed: 10.10.2025).
3. Talakola S. Automated end to end testing with Playwright for React applications. *International Journal of Emerging Research in Engineering and Technology (IJERET)*. 2024. Vol. 5 (1). P. 38-47. DOI: 10.63282/3050-922X.IJERET-V5I1P106.
4. Calculating Test Automation ROI: A Guide. URL: https://www.browserstack.com/-guide/calculate-test-automation-roi (accessed: 09.10.2025).

5. Bylina B., Antończak A. Analysis of end-to-end test automation tools based on the examples of Selenium WebDriver and Playwright. *Conference on Computer Science and Information Systems.* 2024. DOI: 10.15439/2024F3747.
6. Certified Tester Advanced Level. Test Automation Engineering. Syllabus Version 2.0. URL: https://istqb.org/wp-content/uploads/2024/11/ISTQB_CTAL-TAE_Syllabus_v2.0.pdf (accessed: 25.10.2025).
7. Kosinov M., Myastkovska M. Aspects of automated testing framework design enabling testing quality improvement. *Problems and innovations in mathematical, digital, natural, and professional education: collection of materials of the XIX International Scientific and Practical Online Conference*. Kropyvnytskyi, May 20-29, 2025 / ed. M. I. Sadovyi; compilers: M. I. Sadovyi, D. V. Reshetnikova, O. M. Tryfonova. Kropyvnytskyi: Information Department of V. Vynnychenko CUSU, 2025. P. 117-119. URL: https://www.ldftpo.kr.ua/-wp-content/uploads/2025/10/Tezu_XIX_konf.pdf (accessed: 03.11.2025).
8. Bylina B., Antończak A. Analysis of end-to-end test automation tools based on the examples of Selenium WebDriver and Playwright. *Position Papers of the 19th Conference on Computer Science and Intelligence Systems* / eds. M. Bolanowski, M. Ganzha, L. Maciaszek, M. Paprzycki, D. Ślęzak. 2024. Vol. 40. P. 1-8. DOI: http://dx.doi.org/10.15439/2024F3747. URL: https://annals-csis.org/-Volume_40/drp/3747.html (accessed: 13.12.2025).
9. Mońl M., Pańczyk B. A comparative analysis of web application test automation tools. *Journal of Computer Sciences Institute*. 2025. Vol. 35. P. 159-165. DOI: 10.35784/jcsi.7119. URL: https://ph.pollub.pl/index.php/jcsi/article/-view/7119 (accessed: 13.12.2025).
10. Gosik P., Miłosz M. Comparative analysis of Cypress and Playwright frameworks in end-to-end testing for applications based on Angular. *Journal of Computer Sciences Institute.* 2025. Vol. 36. P. 320-327. DOI: 10.35784/jcsi.7662.
11. Stocco A., Leotta M., Ricca F., Tonella P. APOGEN: automatic page object generator for web testing. *Software Quality Journal.* 2017. Vol. 25. P. 1007-1039. DOI: https://doi.org/10.1007/s11219-016-9331-9
12. Yuniasri D., Badriyah T., Sa'adah U. A Comparative Analysis of Quality Page Object and Screenplay Design Pattern on Web-based Automation Testing. *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*. Istanbul, 2020. P. 1-6. DOI: 10.1109/ICECCE49384.2020.9179470.
13. Leotta M., Stocco A., Ricca F., Tonella P. ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing. *Journal of Software: Evolution and Process.* 2016. Vol. 28. Is. 3. P. 177-204. DOI: https://doi.org/-10.1002/smr.1771.
14. Playwright Docs. Test Isolation URL: https://playwright.dev/docs/browser-contexts. (accessed: 28.11.2025).
15. Kosinov M. S. Features of CI/CD process integration for automated web application testing based on Github Actions. *Bulletin of Kamianets-Podilskyi Ivan Ohiienko National University. Physical and Mathematical Sciences.* Kamianets-Podilskyi: Kamianets-Podilskyi Ivan Ohiienko National University, 2025. Is. 18. P. 51-55. URL: https://fizmat.kpnu.edu.ua/wp-content/uploads/2025/12/visnyk-18-2025-15-12-2025.pdf (accessed: 15.12.2025).

СТРУКТУРНО-ФУНКЦІОНАЛЬНЕ МОДЕЛЮВАННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ АДАПТИВНИХ ВЕБ-ІНТЕРФЕЙСІВ

У статті розв'язано актуальне науково-прикладне завдання підвищення ефективності автоматизованого тестування веб-застосунків, розроблених на основі адаптивного дизайну (Responsive Web Design). Актуальність дослідження зумовлена тим, що в умовах домінування концепції Mobile First забезпечення стабільної роботи на гетерогенних платформах створює критичне навантаження на QA-процеси. На основі аналізу традиційних підходів, зокрема патерну Page Object Model, виявлено їхню неефективність у мультиплатформенному середовищі, що проявляється в експоненційному зростанні обсягу коду, порушенні принципів Clean Code та складності адаптації до структурних розбіжностей DOM-елементів у різних в'юпортах.

Метою роботи є підвищення ефективності тестування шляхом розробки архітектури, що забезпечує чітке розмежування бізнес-логіки сценаріїв та технічної реалізації інтерфейсу. Для цього запропоновано структурно-функціональну модель системи, формалізовану теоретико-множинним описом у вигляді кортежу $S = \langle D, P, B, T, C \rangle$, який включає множини тестових даних, об'єктів сторінок, бізнес-кроків, специфікацій та простір станів контексту. Введення динамічного контексту дозволяє представити поведінку сторінок як функцію від вектора конфігурації, забезпечуючи адаптивність системи до умов виконання.

Практичну реалізацію теоретичних положень здійснено шляхом розробки фреймворку з чотирирівневою архітектурою на базі інструментарію Playwright. Впроваджено алгоритм динамічної ін'єкції контексту, що автоматично вибирає релевантну стратегію пошуку локаторів і тип взаємодії (Touch/Mouse) під час виконання. Експериментальні результати засвідчили, що запропонований підхід гарантує архітектурну інваріантність тестів, дозволяє ефективно застосовувати матричну збірку в CI/CD (GitHub Actions) для паралельного виконання в ізольованих контейнерах та скорочує витрати часу на підтримку кодової бази на 35-40%, усуваючи необхідність дублювання сценаріїв для нових пристроїв.

Ключові слова: *автоматизоване тестування, адаптивні веб-інтерфейси, Responsive Web Design, структурно-функціональне моделювання, Playwright, Page Object Model, контекст виконання, CI/CD, матрична збірка, кросплатформне тестування.*

Отримано: 20.12.2025