Моделирование и анализ данных
2025, *15*(4), 87—103.
https://doi.org/10.17759/mda.2025150406
ISSN: 2219-3758 (печатный)
ISSN: 2311-9454 (online)

Modelling and Data Analysis
2025, *15*(4), 87—103.
https://doi.org/10.17759/mda.2025150406
ISSN: 2219-3758 (print)
ISSN: 2311-9454 (online)

Научная статья | Original paper

УДК 004.438

Оптимизация процесса создания автотестов веб-приложений с использованием LLM и структурного анализа HTML

А.М. Титеев
Московский Авиационный Институт, Москва, Российская Федерация
✉ loksader@yandex.ru

Резюме

Контекст и актуальность. Современная разработка веб-приложений требует непрерывного тестирования, однако поддержание автоматических тестов становится все более трудоемкой задачей из-за нестабильности локаторов и растущей сложности интерфейсов. Появление больших языковых моделей (LLM) открывает новые возможности для автоматизации создания тестов, но их практическое применение сопряжено с проблемами обработки больших HTML-документов и необходимостью создания поддерживаемого кода. **Цель.** Разработать и оценить эффективность метода автоматической генерации поддерживаемых тестов веб-приложений с использованием LLM на основе анализа HTML-структуры и паттерна Page Object Model(POM). **Гипотезы.** Основная гипотеза: комбинация LLM с двухэтапным подходом генерации и паттерном POM позволит создавать поддерживаемые тесты, сократив время разработки минимум в 1,5 раза при сохранении читаемости кода. Вторичная гипотеза: успешность автоматической генерации будет обратно пропорциональна сложности интерфейсных компонентов. **Методы и материалы.** В исследовании использовался подход на основе Playwright, LLM и двухэтапной процедуры генерации с промежуточной валидацией. Тестирование проводилось на четырех компонентах SPA-приложения для управления виртуальной инфраструктурой. Валидация результатов выполнялась командой из трех тестировщиков, оценивавших корректность и читаемость сгенерированных тестов. **Результаты.** Предложенный метод обеспечил высокую успешность автоматической генерации тестов и существенное сокращение временных затрат на их создание. Двухэтапная процедура с промежуточной валидацией позволила локализовать значительную часть ошибок на раннем этапе создания объектов Page Object. Автоматически

сгенерированные тесты обеспечили покрытие большей части необходимой функциональности при сохранении читаемости кода. Подтверждена обратная зависимость успешности генерации от сложности интерфейсных компонентов: стандартизированные интерфейсы показали существенно более высокие показатели успешности. **Выводы.** Предложенный метод обеспечивает существенную экономию времени на создание базового набора тестов при сохранении их качества и поддерживаемости. Рекомендуется применение подхода на ранних стадиях разработки функциональности с сохранением экспертного контроля для валидации критически важных сценариев. Метод особенно эффективен для проектов с частыми изменениями интерфейса, большим объемом регрессионного тестирования и компонентов со стандартизированными интерфейсами.

Ключевые слова: автоматизированное тестирование, генерация тестов, веб-приложение, LLM, HTML, Page Object Model, Playwright

Для цитирования: Титеев, А.М. (2025). Оптимизация процесса создания автотестов веб-приложений с использованием LLM и структурного анализа HTML. *Моделирование и анализ данных, 15*(4), 87—103. https://doi.org/10.17759/mda.2025150406

Optimizing web application test automation using LLM and structural HTML analysis

A.M. Titeev
Moscow Aviation Institute, Moscow, Russian Federation
✉ loksader@yandex.ru

Abstract

Context and relevance. Modern web application development requires continuous testing, but maintaining automated tests is becoming increasingly labor-intensive due to locator instability and growing interface complexity. The emergence of Large Language Models (LLM) opens new opportunities for test creation automation, but their practical application faces challenges in processing large HTML documents and the need to create maintainable code. **Objective.** To develop and evaluate the effectiveness of a method for automatic generation of maintainable web application tests using LLM based on HTML structure analysis and the Page Object Model(POM) pattern. **Hypotheses.** Primary hypothesis: combining LLM with a two-stage generation approach and the POM pattern will enable the creation of maintainable tests, reducing development time by at least one-third (to 67% or less) while preserving code readability. Secondary hypothesis: the success rate

of automatic generation will be inversely proportional to the complexity of interface components. **Methods and materials.** The study employed an approach based on Playwright, LLM, and a two-stage generation procedure with intermediate validation. Testing was conducted on four components of an SPA application for virtual infrastructure management. Validation of results was performed by a team of three testers who assessed the correctness and readability of generated tests. **Results.** The proposed method achieved high success rates in automatic test generation and substantial reduction in time costs for test creation. The two-stage procedure with intermediate validation enabled localization of a significant portion of errors at the early stage of Page Object creation. Automatically generated tests provided coverage of most required functionality while maintaining code readability. An inverse relationship between generation success and interface component complexity was confirmed: standardized interfaces demonstrated significantly higher success rates. **Conclusions.** The proposed method provides substantial time savings in creating a baseline test suite while maintaining quality and maintainability. The approach is recommended for early stages of feature development with expert control retained for validating critical scenarios. The method is particularly effective for projects with frequent interface changes, large volumes of regression testing, and components with standardized interfaces.

Keywords: automated testing, test generation, web application, LLM, HTML, Page Object Model, Playwright

For citation: Titeev, A.M. (2025). Optimizing web application test automation using LLM and structural HTML analysis. *Modelling and Data Analysis, 15*(4)*,* 87—103. (In Russ.). https://doi.org/10.17759/mda.2025150406

Введение

Современная разработка веб-приложений характеризуется высокой скоростью выпуска обновлений и растущей сложностью пользовательских интерфейсов, что делает автоматизированное тестирование критически важным элементом процесса обеспечения качества программного продукта. Теоретической основой исследования послужили принципы автоматизации программного тестирования, методы применения больших языковых моделей в разработке программного обеспечения, а также архитектурные паттерны проектирования тестовых фреймворков.

Появление больших языковых моделей (LLM) открыло новые возможности для автоматизации создания тестов веб-приложений. Исследование (Gur et al., 2023) продемонстрировало эффективность языковых моделей в задачах семантической классификации HTML-элементов и автономной веб-навигации в рамках ограниченного контекстного окна модели. В области взаимодействия с веб-элементами авторы работы (Pasupat et al., 2018) предложили метод сопоставления естественно-языковых команд с элементами веб-страницы, разработав набор данных для различных типов

взаимодействий. Подход к предварительному обучению на гипертекстовых данных для улучшения работы с HTML-структурами представили (Aghajanyan et al., 2021).

Значительный вклад в исследование автоматической генерации тестов внесли работы по созданию тестов на основе отчетов об ошибках (Plein et al., 2023), сравнительному анализу инструментов автогенерации модульных тестов (Bhatia et al., 2024), созданию инструмента для фаззинг тестирования с помощью LLM (Xia et al., 2024), а также оценке эффективности различных подходов к генерации unit-тестов (Tang et al., 2023). Методы улучшения взаимодействия с языковыми моделями рассмотрены в работах по влиянию формулировки запросов к языковой модели — промптов — на качество генерируемых тестов (Li, Doiron, 2023) и систематическому обзору техник промптинга (Schulhoff et al., 2024; White et al., 2023).

Однако, несмотря на многочисленные исследования в области применения LLM для тестирования, существующие подходы имеют ограничения. Во-первых, проблема обработки больших HTML-документов современных веб-приложений приводит к превышению контекстного окна языковых моделей и снижению качества генерации. Во-вторых, автоматически созданный код должен быть не только функциональным, но и понятным для человека, поскольку финальная проверка корректности выполняется специалистами по тестированию. В-третьих, отсутствуют комплексные решения, объединяющие эффективную обработку HTML-структур с созданием поддерживаемых тестов по архитектурному паттерну Page Object Model(POM) — подходу, при котором каждая страница или значимый компонент веб-приложения представляется отдельным классом, инкапсулирующим селекторы элементов и методы взаимодействия с ними, что обеспечивает централизованное управление изменениями интерфейса и повторное использование кода. В отличие от существующих решений, предлагаемый метод интегрирует большие языковые модели с системой базовых классов и архитектурным паттерном POM, что обеспечивает создание не только функционально корректного, но и поддерживаемого, читаемого кода, пригодного для использования специалистами по тестированию

Цель исследования — разработать и оценить эффективность метода автоматической генерации поддерживаемых тестов веб-приложений с использованием больших языковых моделей на основе анализа HTML-структуры и паттерна POM.

Основная гипотеза исследования — комбинация больших языковых моделей с двухэтапным подходом генерации (создание объектов POM с последующей генерацией тестовых сценариев) и применение архитектурного паттерна POM позволит создавать поддерживаемые тесты веб-приложений, сократив время разработки минимум в полтора раза при сохранении читаемости и качества кода по сравнению с традиционным ручным подходом.

Вторичная гипотеза — успешность автоматической генерации тестов будет обратно пропорциональна сложности интерфейсных компонентов: стандартизированные интерфейсы продемонстрируют значительно более высокую успешность генерации по сравнению со сложными специфичными компонентами.

Предлагаемый подход основан на разделении задачи генерации на две последовательные подзадачи: создание объектов модели страницы с валидацией и генерацию тестовых сценариев на их основе. Метод извлечения релевантных фрагментов HTML используется для эффективной работы в рамках ограничений контекстного окна языковых моделей без потери критически важного контекста. Применение базовых классов и архитектурного паттерна POM обеспечивает создание читаемого и поддерживаемого тестового кода.

Материалы и методы

Исследование выполнено с использованием экспериментального подхода для оценки эффективности автоматической генерации тестов веб-приложений на основе комбинации больших языковых моделей с архитектурным паттерном POM. Методология исследования основана на сравнительном анализе результатов генерации тестов с применением различных техник обработки HTML-структур и промптинга языковых моделей.

В качестве объекта исследования использовалось SPA-приложение для управления виртуальной инфраструктурой, включающее четыре основных функциональных компонента: система авторизации, управление виртуальными машинами, управление сетевой инфраструктурой и система хранения данных. Выбор объекта исследования обусловлен его репрезентативностью для задач автоматизации тестирования: приложение содержит разнообразные типы интерфейсных элементов (формы, таблицы, модальные окна, динамические списки), что позволяет оценить универсальность предложенного подхода.

Для реализации подхода использовался фреймворк автоматизации тестирования Playwright, выбранный на основе сравнительного анализа современных решений (Selenium, Cypress, Puppeteer, Playwright). Критериями отбора служили: скорость выполнения тестов, поддержка современных браузеров, наличие встроенных механизмов ожидания элементов, возможность работы с Shadow DOM и качество интеграции с CI/CD системами.

В качестве языковой модели применялась модель Llama 3.3 70B instruct. Для извлечения релевантных фрагментов HTML-документов использован метод Snippet Extraction (Gur et al., 2023, p. 2806), извлекающий основные(salient) элементы кода HTML и позволяющий сохранять семантические связи и иерархию элементов при сокращении объема обрабатываемых данных.

Архитектурная организация тестов основана на паттерне POM с использованием системы базовых классов, обеспечивающих инкапсуляцию логики взаимодействия с элементами интерфейса, централизованное управление селекторами и автоматическую генерацию описательных отчетов о выполнении тестов. Система базовых классов включает компоненты для основных типов элементов интерфейса (Button, Input, Table, Modal и др.), каждый из которых инкапсулирует специфическую логику взаимодействия и валидации. Выбор паттерна POM обоснован результатами промышленного исследования

(Leotta et al., 2013), которое продемонстрировало, что применение данного паттерна существенно снижает связанность между тестовыми случаями и тестируемым интерфейсом, уменьшает дублирование кода и упрощает управление локаторами элементов по сравнению с подходами без использования объектной абстракции страниц.

Ключевой особенностью предложенной методологии является разделение процесса генерации на два последовательных этапа с промежуточной валидацией, что обеспечивает раннюю локализацию ошибок.

На первом этапе для каждого компонента веб-приложения выполнялись следующие процедуры:

1. Предварительная обработка HTML-структуры:
   - Применение метода Snippet Extraction для извлечения релевантных фрагментов HTML;
   - Определение контекстного окружения целевых элементов с сохранением иерархических связей;
   - Фильтрация нерелевантной информации с сохранением структурных атрибутов.
2. Формирование промпта для генерации Page Object:
   - Включение извлеченной HTML-структуры компонента;
   - Добавление определений базовых классов фреймворка;
   - Предоставление примеров корректно реализованных Page Object;
   - Спецификация архитектурных требований и соглашений о наименовании.
3. Генерация Page Object языковой моделью:
   - Автоматическое создание класса объекта модели страницы;
   - Определение селекторов для всех значимых элементов интерфейса;
   - Реализация методов взаимодействия с использованием базовых классов.
4. Валидация сгенерированного Page Object:
   - Проверка синтаксической корректности кода;
   - Верификация корректности определенных селекторов;
   - Тестирование базовых методов взаимодействия;
   - Оценка соответствия архитектурным требованиям.
5. Исправление выявленных ошибок:
   - Документирование типа и характера ошибок;
   - Ручная коррекция некорректных селекторов или структуры класса;
   - Повторная валидация после исправлений.

Только после успешной валидации Page Object процесс переходил ко второму этапу. Это критически важно для локализации ошибок: все проблемы, связанные с пониманием структуры интерфейса и определением селекторов, выявляются и устраняются до генерации тестов.

На втором этапе для каждого компонента с валидированным Page Object выполнялись:

1. Формирование промпта для генерации тестов:
   - Включение валидированного класса Page Object в промпт;

- Предоставление спецификации функциональности компонента;
- Добавление примеров тестовых сценариев;
- Указание требований к структуре и организации тестов.

2. Генерация тестов:
   - Автоматическое создание тестов с использованием методов валидированного Page Object;
   - Генерация проверок для валидации функциональности;
   - Создание вспомогательных методов для подготовки тестовых данных.
3. Валидация сгенерированных тестов:
   - Выполнение тестов в тестовом окружении;
   - Проверка корректности тестов;
   - Оценка покрытия функциональности.
4. Классификация ошибок:
   - Ошибки в Page Object: если при выполнении теста выявлялась некорректность селектора, метода взаимодействия или структуры Page Object, объект помечался как неправильно сгенерированный, даже если прошел первичную валидацию;
   - Ошибки в тесте: некорректная логика теста, неправильная последовательность действий или ошибочные проверки при корректном Page Object;
   - Ошибки в понимании функциональности: несоответствие теста реальному поведению приложения.

Применение few-shot подхода (Brown et al., 2020) обосновано результатами исследования (Kang et al., 2023), показавшего, что предоставление языковой модели нескольких примеров корректных тестов существенно повышает качество генерации по сравнению с zero-shot режимом, то есть без предоставления примеров в промпте. В нашей реализации промпт включал 2—3 эталонных примера тестовых сценариев с корректной структурой, паттернами использования Page Object методов, что позволило модели изучить ожидаемый формат выходных данных.

Валидация результатов выполнялась командой из трех квалифицированных специалистов по тестированию программного обеспечения с опытом работы в области автоматизации тестирования. Каждый сгенерированный тест оценивался независимо всеми участниками по следующим критериям:

1. Функциональная корректность — способность теста успешно выполняться и проверять заявленную функциональность;
2. Читаемость кода — понятность структуры и логики теста для человека;
3. Поддерживаемость — простота внесения изменений при модификации интерфейса;
4. Соответствие архитектурному паттерну — корректность применения POM.

Общая оценка теста считалась положительной при достижении консенсуса не менее чем у двух экспертов по всем критериям оценки.

Временные затраты на создание тестов измерялись следующим образом:

1. Фиксировалось время создания полного набора тестов для каждого компонента отдельно.
2. Тесты группировались по уровням сложности:
   - Простой POM: < 30 строк кода;
   - Сложный POM: ≥ 30 строк кода;
   - Простой тест: < 15 строк кода;
   - Сложный тест: ≥ 15 строк кода.
3. Суммарное время для всех компонентов одного уровня сложности делилось на общее количество созданных тестов этого уровня.

Для подхода с использованием LLM применялась аналогичная методология с учетом полного цикла:

1. Время генерации промпта;
2. Время работы модели;
3. Время валидации тестов и исправления ошибок.

Для компонента авторизации было создано 6 простых Page Object. Компонент управления виртуальными машинами потребовал 8 простых и 13 сложных Page Object. Для компонента управления сетевой инфраструктурой было разработано 8 простых и 10 сложных Page Object. Наиболее объемным оказался компонент системы хранения данных, для которого потребовалось 11 простых и 17 сложных Page Object. Таким образом, общее количество созданных Page Object составило 73, из них 33 простых и 40 сложных, что отражает различную степень сложности пользовательских интерфейсов тестируемых компонентов. Сгенерировано 1232 теста исходя из необходимости покрытия всех функциональных компонентов исследуемого приложения с учетом различных сценариев использования.

Критерии сложности определялись следующим образом: тест считался простым при объеме менее 15 строк кода, Page Object — при объеме менее 30 строк кода. Временные затраты на создание тестов фиксировались для последующего сравнения эффективности автоматизированного и ручного подходов.

Результаты

Экспериментальная проверка предложенного подхода продемонстрировала существенное повышение эффективности автоматической генерации тестов веб-приложений. Применение больших языковых моделей в сочетании с архитектурным паттерном POM позволило достичь 81% успешности генерации при создании 1232 тестовых сценариев для четырех компонентов исследуемого приложения.

Первый этап генерации включал создание объектов модели страницы для каждого компонента приложения. Результаты генерации простых и сложных Page Object представлены в табл. 1 и табл. 2, демонстрирующих различную степень сложности компонентов и успешность автоматической генерации.

Таблица 1 / Table 1

Результаты генерации простых объектов POM

Simple POM generation results

Компонент / Component	Всего / Total	Успешных / Successful	Успешность / Success rate
Авторизация / Authorization	6	6	100%
Виртуальные машины / Virtual Machines	8	6	75%
Сети / Networks	8	5	63%
Хранилища / Storage	11	7	64%

Примечание: Простые Page Object содержат менее 30 строк кода.
Note: Simple Page Objects contain less than 30 lines of code.

Таблица 2 / Table 2

Результаты генерации сложных объектов POM

Complex POM generation results

Компонент / Component	Всего / Total	Успешных / Successful	Успешность / Success rate
Авторизация / Authorization	0	0	—
Виртуальные машины / Virtual Machines	13	5	38%
Сети / Networks	10	4	40%
Хранилища / Storage	17	6	35%

Примечание: Сложные Page Object содержат 30 строк кода и более.
Note: Complex Page Objects contain 30 lines or more of code.

Всего было создано 73 объекта Page Object: 33 простых и 40 сложных. Из них успешно сгенерировано 39 объектов (24 простых и 15 сложных), что составляет общую успешность 53%. Компонент авторизации показал наилучший результат с 100% успешностью генерации простых Page Object, что объясняется его относительной простотой и стандартизированной структурой интерфейса. Более сложные компоненты (виртуальные машины, сети, хранилища) продемонстрировали успешность на уровне 35—75%, причем генерация сложных Page Object оказалась значительно менее успешной по сравнению с простыми. Это связано с наличием множественных динамических элементов и сложных иерархических структур интерфейса.

На основе валидированных объектов Page Object были сгенерированы тестовые сценарии для всех компонентов приложения. Детальное распределение сгенерированных тестов по компонентам и уровням сложности представлено в табл. 3 и табл. 4.

Всего сгенерировано 1232 тестовых сценария: 325 простых и 907 сложных. Из них успешно прошли валидацию 997 тестов (286 простых и 711 сложных), что составляет общую успешность 81%. Простые тесты показали более высокую успешность генерации (88%) по сравнению со сложными тестами (78%). Компонент авторизации

продемонстрировал наивысшую успешность (92%), тогда как более сложные компоненты показали показатели в диапазоне 79—82%. Общая успешность генерации тестовых сценариев (81%) значительно превышает показатели генерации Page Object (53%), что свидетельствует об эффективности двухэтапного подхода: валидированные объекты модели страницы обеспечивают надежную основу для генерации тестов.

Таблица 3 / Table 3

Результаты генерации простых тестовых сценариев

Simple test scenario generation results

Компонент / Component	Всего тестов / Total tests	Успешных тестов / Successful tests	Успешность / Success rate
Авторизация / Authorization	35	33	94%
Виртуальные машины / Virtual Machines	92	81	88%
Сети / Networks	58	48	83%
Хранилища / Storage	140	124	89%

Примечание: простые тесты содержат менее 15 строк кода.
*Note: s*imple tests contain less than 15 lines of code.

Таблица 4 / Table 4

Результаты генерации сложных тестовых сценариев

Complex test scenario generation results

Компонент / Component	Всего тестов / Total tests	Успешных тестов / Successful tests	Успешность / Success rate
Авторизация / Authorization	16	14	88%
Виртуальные машины / Virtual Machines	284	229	81%
Сети / Networks	162	132	81%
Хранилища / Storage	445	336	76%

Примечание: сложные тесты содержат 15 строк кода и более.
Note: complex tests contain 15 lines or more of code.

Анализ временных затрат на создание тестов показал существенное сокращение времени разработки при использовании предложенного подхода по сравнению с традиционным ручным методом (табл. 5).

Таблица 5 / Table 5

Сравнение временных затрат на создание тестов

Comparison of time costs for test creation

Тип задачи / Task type	Ручное создание (минуты) / Manual creation (minutes)	С использованием LLM (минуты) / With LLM (minutes)	Коэффициент сокращения / Reduction factor
Простой Page Object / Simple Page Object	11	4	2,75

Тип задачи / Task type	Ручное создание (минуты) / Manual creation (minutes)	С использованием LLM (минуты) / With LLM (minutes)	Коэффициент сокращения / Reduction factor
Сложный Page Object / Complex Page Object	83	28	2,96
Простой тест / Simple test	7	4	1,75
Сложный тест / Complex test	23	15	1,53

Примечание: временные затраты включают полный цикл создания и валидации тестов.
Note: time costs include the complete cycle of test creation and validation.

Команда тестировщиков дополнила набор тестов до полного покрытия, доведя общее количество до 1865 тестов. Автоматически сгенерированные тесты обеспечили покрытие 66% от общего объема необходимой функциональности тестирования.

Наибольшее количество тестов сгенерировано для компонента системы хранения данных (47% от общего числа), что отражает высокую сложность и функциональную насыщенность данного модуля. Компонент управления виртуальными машинами потребовал 31% тестов, управление сетевой инфраструктурой — 18%, тогда как система авторизации, имеющая наиболее стандартизированную структуру, потребовала лишь 4% от общего числа тестов.

Таблица 6 / Table 6

Распределение типов ошибок при генерации Page Object

Distribution of error types in generation process Page Object

Тип ошибки / Error type	Количество / Count	Доля / Share
Некорректная структура класса / Incorrect class structure	21	62%
Ошибки селекторов элементов / Element selector errors	13	38%

Таблица 7 / Table 7

Распределение типов ошибок при генерации тестов

Distribution of error types in tests generation process

Тип ошибки / Error type	Количество / Count	Доля / Share
Неправильная логика теста / Incorrect test logic	98	41%
Некорректные проверки / Incorrect assertions	56	23%
Ошибки понимания функциональности / Functionality misunderstanding	47	20%
Плохая читаемость / Bad Readability	13	6%
Не соответствие архитектурному паттерну POM / Non-conformity to the POM architectural pattern	21	9%

Детальный анализ неуспешных случаев генерации выявил распределение ошибок между этапами создания Page Object и тестовых сценариев. Результаты классификации представлены в табл. 6 и табл. 7. Двухэтапная структура процесса генерации позволила локализовать ошибки на ранней стадии. Ошибки в объектах Page Object составили 34 случая из 73 (46% неуспешных). Общее количество ошибок составило 235 из 1232 сгенерированных тестов (19% неуспешных случаев).

Обсуждение результатов

Полученные результаты подтверждают выдвинутую гипотезу о возможности существенного сокращения времени разработки тестов веб-приложений при использовании комбинации больших языковых моделей с архитектурным паттерном POM. Достигнутое сокращение временных затрат превышает заявленный в гипотезе минимальный порог в 1,5 раза, что свидетельствует об эффективности предложенного подхода.

Разделение процесса генерации на два последовательных этапа с промежуточной валидацией продемонстрировало существенные преимущества. Успешность генерации тестовых сценариев (81%) значительно превысила успешность генерации объектов Page Object (53%), что подтверждает правильность архитектурного решения о локализации ошибок на ранней стадии.

Валидация Page Object перед генерацией тестов позволила избежать мультипликации ошибок за счет концентрации их в Page Object. Двухэтапный подход обеспечил изоляцию 34 ошибок в Page Object, не допустив их распространения на 1232 тестовых сценария.

Анализ результатов выявил существенную зависимость успешности генерации от сложности создаваемых компонентов. Для простых Page Object успешность составила 73%, тогда как для сложных снизилась до 37% — практически двукратное различие. Аналогичная тенденция наблюдается для тестовых сценариев: простые тесты достигли 88% успешности против 78% для сложных.

Сложные Page Object требуют анализа множественных взаимосвязанных элементов интерфейса, идентификации динамических компонентов и понимания паттернов взаимодействия между ними. Языковые модели демонстрируют затруднения в построении комплексных абстракций, что приводит к ошибкам в архитектуре класса и некорректным селекторам.

Различие в успешности между компонентами приложения (от 100% для авторизации до 46% для хранилищ) подтверждает гипотезу о том, что стандартизированные интерфейсные паттерны генерируются более надежно. Компонент авторизации, реализующий типовые формы ввода, показал максимальную успешность, тогда как специфичные интерфейсы управления инфраструктурой потребовали существенного человеческого вмешательства.

Достигнутое сокращение временных затрат демонстрирует различную эффективность подхода для задач различной сложности. Наибольший эффект наблюдается для создания сложных объектов Page Object (сокращение в 2,96 раза), что объясняется способностью языковых моделей эффективно обрабатывать большие

HTML-структуры и генерировать соответствующий код, даже несмотря на относительно низкую итоговую успешность (37%).

Важно отметить, что даже неуспешная генерация сложного Page Object экономит время разработчика: некорректный автоматически созданный код служит основой для доработки, сокращая время на анализ HTML-структуры и начальную реализацию класса. Это объясняет парадокс высокой временной эффективности при низкой успешности генерации: языковая модель правильно идентифицирует большинство элементов интерфейса и создает базовую структуру класса, требующую лишь точечных исправлений.

Меньшее сокращение времени для простых и сложных тестовых сценариев является следствием того, что сгенерированные тесты все равно должны быть запущены, проверены и, в случае ошибки, исправлены человеком. Данный результат согласуется с концепцией human-in-the-loop, описанной в обзоре (Wang et al., 2024): языковые модели эффективно автоматизируют создание тестовых случаев и базовую оценку покрытия, однако верификация корректности поведения программы и валидация test oracle остаются в компетенции человека-эксперта. Автоматическая генерация эффективно справляется с созданием базовых проверок и типовых сценариев взаимодействия, однако сложные edge cases и специфичная логика все еще требуют человеческой экспертизы, что объясняет меньшую эффективность для сложных сценариев.

Предложенный подход генерирует структурный каркас тестов и Page Object с помощью LLM, сокращая время на рутинную работу с HTML-разметкой и написание шаблонного кода, в то время как человек фокусируется на критической валидации функциональной корректности. Это распределение ответственности объясняет наблюдаемое сокращение времени в 1,5—3 раза вместо полной замены ручного труда: временные затраты на валидацию и исправление остаются необходимым компонентом процесса тестирования, обеспечивающим качество конечного результата.

Сравнение с альтернативными подходами к автоматизации генерации тестов веб-приложений демонстрирует различные стратегии решения проблемы. Подход, предложенный (Fard, Mirzaaghaei, Mesbah, 2014), использует извлечение знаний из существующих Selenium-тестов — метод, комплементарный предлагаемому в данной работе. В то время как данный метод эффективен при наличии базы существующих тестов для повторного использования паттернов взаимодействия и тестовых данных, предложенный нами подход с использованием LLM решает задачу первоначального создания тестов при отсутствии накопленной базы знаний. Потенциальным направлением дальнейших исследований является гибридный метод, комбинирующий извлечение паттернов из существующих тестов с генеративными возможностями языковых моделей для создания более качественных промптов.

Применение архитектурного паттерна POM в сочетании с системой базовых классов обеспечило создание структурированного и поддерживаемого кода. Покрытие 66% необходимой функциональности автоматически сгенерированными тестами представляет значительное достижение, учитывая сложность задач понимания семантики интерфейса и генерации соответствующих проверок.

Использование системы базовых классов (Button, Input, Table, Modal и др.) критически важно для обеспечения консистентности сгенерированного кода. Языковая модель,

получая определения этих классов в промпте, генерирует код, соответствующий установленным архитектурным паттернам. Это обеспечивает унификацию подхода к взаимодействию с элементами интерфейса и упрощает последующую поддержку тестов.

Оценка читаемости кода командой из трех экспертов показала, что сгенерированные тесты не требуют существенного рефакторинга для понимания логики. Это критически важно, поскольку конечными пользователями автоматически созданных тестов являются специалисты по тестированию, которые должны понимать, модифицировать и поддерживать этот код.

Полученные результаты согласуются с выводами исследования (Yuan et al., 2024), где проводилась оценка читаемости (readability) и пригодности (usability) тестов, сгенерированных ChatGPT. Авторы отмечают, что даже при высоком качестве сгенерированного кода необходим механизм валидации и исправления, что подтверждается и нашим двухэтапным подходом с промежуточной проверкой объектов Page Object. Критическое различие состоит в том, что наш метод фокусируется на веб-приложениях и паттерне POM, тогда как ChatTester ориентирован на модульные тесты Java-кода, что подчеркивает важность адаптации LLM-подходов к специфике предметной области тестирования.

Выявленные категории ошибок указывают на специфические ограничения современных языковых моделей в задачах автоматизации тестирования. Преобладание ошибок в логике тестов (41% от ошибок этапа генерации тестов) свидетельствует о сложности автоматического понимания заложенных требований и ожидаемого поведения системы. Языковая модель способна корректно идентифицировать элементы интерфейса и генерировать код взаимодействия с ними, однако определение правильной последовательности действий и ожидаемых результатов требует понимания контекста, который часто отсутствует в HTML-разметке.

Ошибки в определении селекторов элементов (38% от ошибок Page Object) отражают проблему нестабильности локаторов в современных веб-приложениях. Динамически генерируемые идентификаторы, теневой DOM и асинхронная загрузка контента создают дополнительные сложности для автоматической идентификации элементов.

Некорректная структура класса (62% от ошибок Page Object) указывает на недостаточное понимание языковой моделью принципов организации объектов модели страницы. Модель может корректно определить отдельные элементы интерфейса, но испытывает затруднения с группировкой связанных элементов, определением уровня абстракции и созданием эффективной иерархии методов.

Заключение

Проведенное исследование продемонстрировало практическую эффективность метода автоматизации создания тестов веб-приложений на основе комбинации больших языковых моделей с архитектурным паттерном POM.

Предложенный метод обеспечил высокую успешность автоматической генерации тестов и существенное сокращение временных затрат на их создание. Двухэтапная

процедура генерации с промежуточной валидацией позволила локализовать значительную часть ошибок на раннем этапе создания объектов Page Object, предотвратив их распространение на тестовые сценарии. Автоматически сгенерированные тесты обеспечили покрытие большей части необходимой функциональности при сохранении качества и читаемости кода по оценке команды экспертов.

Метод обеспечивает существенную экономию времени на создание базового набора тестов, что критически важно для проектов с большим объемом регрессионного тестирования и частыми изменениями интерфейса. Подход наиболее эффективен для компонентов со стандартизированными интерфейсами, где успешность генерации достигает максимальных значений. Для сложных специфичных интерфейсов рекомендуется гибридный подход с сохранением экспертного контроля.

Целесообразно применение метода на ранних стадиях разработки функциональности для быстрого получения базового покрытия с последующей доработкой тестов по мере стабилизации интерфейса. Критически важна валидация сгенерированных объектов Page Object перед созданием тестовых сценариев, поскольку ошибки на архитектурном уровне приводят к необходимости модификации множественных зависимых тестов. Использование системы базовых классов и стандартизированных паттернов обеспечивает консистентность кода и упрощает последующую поддержку тестов.

Ограничения. Исследование проводилось на компонентах одного SPA-приложения с использованием конкретной языковой модели, что может ограничивать обобщаемость результатов. Валидация выполнялась ограниченной командой экспертов, что вносит элемент субъективности в оценку качества. Требуется верификация эффективности подхода на более широком спектре веб-приложений, различных типах интерфейсов и с использованием других языковых моделей для подтверждения универсальности метода.

Limitations. The study was conducted on components of a single SPA application using a specific language model, which may limit the generalizability of the results. Validation was performed by a limited team of experts, introducing an element of subjectivity in quality assessment. Verification of the approach's effectiveness across a broader spectrum of web applications, various interface types, and with different language models is required to confirm the universality of the method.

Список источников / References

1. Aghajanyan, A., Okhonko, D., Lewis, M., Joshi, M., Xu, H., Ghosh, G., Zettlemoyer, L. (2021). Hyper-text pre-training and prompting of language model. Article. https://doi.org/10.48550/arXiv.2107.06955
2. Bhatia, S., Gandhi, T., Kumar, D., Jalote, P. (2024). Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools. Article. https://doi.org/10.48550/arXiv.2312.10622
3. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R.,

Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D. (2020). Language Models are Few-Shot Learners. Article. https://doi.org/10.48550/arXiv.2005.14165

4. Fard, A.M, Mirzaaghaei, M., Mesbah, A. (2014). Leveraging existing tests in automated test generation for web applications. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14), New York, NY, USA: ACM, 67—78. https://doi.org/10.1145/2642937.2642991
5. Gur, I., Nachum, O., Miao, Y., Safdari, M., Huang, A., Chowdhery, A., Narang, S., Fiedel, N., Faust, A. (2023). Understanding HTML with Large Language Models. *Findings of the Association for Computational Linguistics: EMNLP 2023,* 2803—2821. https://doi.org/10.18653/v1/2023.findings-emnlp.185
6. Leotta, M., Clerissi, D., Ricca, F., Spadaro, C. (2013). Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study. IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, Luxembourg, 2013, pp 108—113. https://doi.org/10.1109/ICSTW.2013.19
7. Li, V., Doiron, N. (2023). Prompting Code Interpreter to Write Better Unit Tests on Quixbugs Functions. Article. https://doi.org/10.48550/arXiv.2310.00483
8. Pasupat, P., Jiang, T.-S., Liu, E.Z., Guu, K., Liang, P. (2018). Mapping natural language commands to web elements. Article. https://doi.org/10.48550/arXiv.1808.09132
9. Plein, L., Ouédraogo, W.C., Klein, J., Bissyandé, T.F. (2023). Automatic Generation of Test Cases Based on Bug Reports: A Feasibility Study with Large Language Models. Article. https://doi.org/10.48550/arXiv.2310.06320
10. Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C., Li, Y., Gupta, A., Han, H., Schulhoff, S., Dulepet, P.S., Vidyadhara, S., Ki, D., Agrawal, S., Pham, C., Kroiz, G., Li, F., Tao, H., Srivastava, A., Da Costa, H., Gupta, S., Rogers, M.L., Goncearenco, I., Sarli, G., Galynker, I., Peskoff, D., Carpuat, M., White, J., Anadkat, S., Hoyle, A., Resnik, P. (2024). The Prompt Report: A Systematic Survey of Prompting Techniques. Article. https://doi.org/10.48550/arXiv.2406.06608
11. Tang, Y., Liu, Z., Zhou, Z., Luo, X. (2023). ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. Article. https://doi.org/10.48550/arXiv.2307.00588
12. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q. (2024). Software Testing with Large Language Models: Survey, Landscape, and Vision. IEEE Transactions on Software Engineering, 50(4), 911—936. https://doi.org/10.1109/TSE.2024.3368208
13. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. Article. https://doi.org/10.48550/arXiv.2302.11382
14. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., Zhang, L. (2024). Fuzz4All: Universal Fuzzing with Large Language Models. ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering(ICSE '24). Article 126, 1—13. https://doi.org/10.1145/3597503.3639121
15. Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., Lou, Y. (2024). Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering, 1(FSE), Article 76,* 1703—1726. https://doi.org/10.1145/3660783

Информация об авторах
Александр Максимович Титеев, аспирант кафедры вычислительной математики и программирования, Московский Авиационный Институт(МАИ), Москва, Российская Федерация, ORCID: https://orcid.org/0009-0003-7754-1550, e-mail: loksader@yandex.ru

Information about the authors
Alexander M. Titeev, Postgraduate Student, Department of Computational Mathematics and Programming, Moscow Aviation Institute (MAI), Moscow, Russian Federation, ORCID: https://orcid.org/0009-0003-7754-1550, e-mail: loksader@yandex.ru

Вклад авторов
Все авторы приняли участие в обсуждении результатов и согласовали окончательный текст рукописи.

Contribution of the authors
All authors participated in the discussion of the results and approved the final text of the manuscript.

Конфликт интересов
Авторы заявляют об отсутствии конфликта интересов.

Conflict of interest
The authors declare no conflict of interest.

Поступила в редакцию 09.10.2025 — Received 2025.10.09
Поступила после рецензирования 25.10.2025 — Revised 2025.10.25
Принята к публикации 05.11.2025 — Accepted 2025.11.05
Опубликована 28.12.2025 — Published 2025.12.28