

Лекция 7.

# **Автоматизация тестирования**

**Автоматизированное тестирование** — набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования.

Тест-кейсы частично или полностью выполняет специальное инструментальное средство, однако разработка тест-кейсов, подготовка данных, оценка результатов выполнения, написания отчётов об обнаруженных дефектах — всё это и многое другое по-прежнему делает человек.

# **Преимущества и недостатки автоматизации тестирования**

## Эффективность и надёжность

- **Скорость выполнения**  
Тесты запускаются в разы быстрее, чем при ручном тестировании
- **Отсутствие человеческого фактора**  
Исключены ошибки из-за усталости, невнимательности, рутины
- **Экономия при регрессионном тестировании**  
Многократный запуск требует минимального участия человека

## Расширенные возможности

- **Тестирование «нечеловеческих» сценариев**  
Сложные, высокоскоростные или объёмные тесты, недоступные человеку
- **Работа с большими данными**  
Автоматизация собирает, анализирует и визуализирует колоссальные объёмы результатов
- **Низкоуровневый доступ**  
Возможность взаимодействия с приложением и ОС на уровне API, памяти, сетевых пакетов

## Ресурсы и сложность

- **Высокие требования к персоналу**

Автоматизация — «проект внутри проекта»: нужны навыки программирования, проектирования, управления

- **Значительные затраты**

Лицензии на инструменты + разработка и сопровождение тестового кода

- **Усиленное планирование и управление рисками**

Ошибки в автоматизации могут нанести серьёзный ущерб проекту


## Проблемы поддержки и адаптации

- **Сложность выбора инструментов**

Многообразие решений → риск ошибочного выбора,  
дополнительные затраты на обучение

- **Хрупкость при изменениях**

Изменение требований, интерфейсов или технологического стека  
часто делает тесты устаревшими и требует их полной переработки









 *Автоматизация эффективна при стабильных требованиях и частых повторных запусках*

# Время разработки и выполнения тест-кейсов



Лучше день потратить, потом за 5 минут долететь?

# Разумный баланс

Критерий	Ручное тестирование	Автоматизированное тестирование 
Гибкость	 Высокая	 Низкая при изменениях
Скорость	 Низкая	 Высокая
Затраты на запуск	 Высокие (повторно)	 Низкие (после внедрения)
Первоначальные затраты	 Низкие	 Высокие

## Рекомендация:

Комбинированный подход — автоматизация для регрессии, стабильных модулей и объёмных проверок; ручное тестирование для исследовательского тестирования, новых функций и изменчивых требований.


# **Области эффективного применения автоматизации**

Автоматизация особенно ценна там, где требуется:


- высокая повторяемость
- точность без исключений
- обработка больших объёмов данных или конфигураций
- работа вне пользовательского интерфейса


# Регрессионное и дымовое тестирование

Задача	Какую проблему решает автоматизация	Пример из практики 
Регрессионное тестирование	Проверка, что старая функциональность не сломалась после изменений — рутинно, объёмно, но критично	После каждого коммита в CI/CD запускается набор из 500+ регрессионных тестов за 10 минут (вместо 2 дней ручного тестирования)
Дымовое тестирование	Быстрая проверка основных функций после каждого билда — особенно в крупных системах	В проекте с ежедневными сборками дымовой тест из 30 ключевых сценариев запускается автоматически и блокирует развёртывание при падении


 Автоматизация обеспечивает мгновенный фидбек при каждом новом сборке



# Инсталляция, настройка и конфигурации

Задача	Какую проблему решает автоматизация	Пример из практики 
Инсталляционное тестирование	Проверка корректности установки: файлы, реестр, конфиги — всё делается точно и многократно	Автоматический скрипт проверяет, что инсталлятор создаёт 120 файлов в нужных папках, прописывает 15 записей в реестр и запускает службу — на Windows, macOS и Linux
Конфигурационное / совместимостное тестирование	Тестирование сотен/тысяч комбинаций параметров, ОС, версий, устройств — невозможно вручную	Приложение тестируется на 5 версиях ОС × 4 браузера × 10 разрешений экрана = 200 комбинаций — всё за ночь в облаке

 Автоматизация гарантирует воспроизводимость среды и покрытие множества конфигураций


# Специализированные виды тестирования

Задача	Какую проблему решает автоматизация	Пример из практики 
Тестирование безопасности	Проверка уязвимостей, портов, прав доступа — без пропусков и с полным охватом	Инструмент OWASP ZAP автоматически сканирует API на 50+ типов уязвимостей при каждом релизе
Тестирование производительности	Генерация нагрузки с точностью и скоростью, недоступной человеку; сбор и анализ больших логов	JMeter имитирует 10 000 одновременных пользователей; система выдерживает 500 RPS — результаты сохраняются в Grafana
Комбинаторное тестирование	Автоматическая генерация и прогон тысяч комбинаций входных данных	Для конфигурационного файла с 5 параметрами (по 10 значений каждый) сгенерировано 100 000 комбинаций → отфильтровано до 120 с помощью pairwise-техники и протестировано за ночь

  *Эффективность, точность и масштаб — ключевые преимущества в этих областях*

# Рутинные и длительные операции


Задача	Какую проблему решает автоматизация	Пример из практики 
Длительные, рутинные, утомительные проверки	Исключает ошибки из-за усталости, обеспечивает точность при работе с большими объёмами данных или файлов	Автоматический скрипт проверяет целостность 50 000+ файлов в архиве: хэши, размеры, кодировки — за 2 часа вместо недели ручной работы
Повторяющиеся высокоточные операции	Гарантирует одинаковый результат при каждом запуске, даже если операция длится часами	Тест генерации отчётов: система создаёт 1000 PDF-документов с расчётами; автоматизация сверяет каждый с эталоном по содержимому и формату

 Человек теряет концентрацию — автоматизация нет

# Тестирование веб-инфраструктуры и стандартной функциональности


---


Задача	Какую проблему решает автоматизация	Пример из практики 
Проверка «внутренней» веб-функциональности	Быстро и надёжно проверяет ссылки, HTTP-статусы, доступность страниц	Инструмент <a href="#">Screaming Frog</a> или кастомный скрипт на Python (с requests/BeautifulSoup) проверяет 30 000+ URL: все ли возвращают 200 OK, нет ли битых ссылок
Стандартная, переиспользуемая функциональность	Высокие первоначальные затраты окупаются при многократном применении в разных проектах	Фреймворк для автоматизации авторизации, корзины и поиска — используется в 10+ e-commerce проектах команды без изменений

 Автоматизация превращает «скучные» задачи в одноразовые инвестиции

## «Технические» проверки вне UI

---

Задача	Какую проблему решает автоматизация	Пример из практики 
Проверка логирования и баз данных	Контролирует корректность записей, транзакций, индексов — то, что не видно пользователю	После выполнения операции «оплата» автоматизированный тест проверяет: 1) запись в таблицу <code>transactions</code> , 2) лог-файл содержит событие <code>PAYMENT_SUCCESS</code> , 3) баланс обновлён
Валидация генерируемых документов и файлов	Обеспечивает точное соответствие формату (XML, JSON, PDF, Excel) и содержимому	Система выставления счетов генерирует XML по стандарту UBL; автоматизация проверяет структуру, обязательные поля и цифровую подпись с помощью XSD и криптографических библиотек

 Эти проверки критичны для надёжности, но невидимы — автоматизация делает их прозрачными

**Всегда ли автоматизация  
эффективна?**

## Автоматизация не заменяет мышление

---

Автоматизация уместна там, где есть **стабильность, повторяемость и чёткие критерии.**

В остальных случаях — она может навредить.


# Задачи, требующие человеческого участия

Задача	Почему автоматизация не подходит	Пример из практики
Планирование тестирования	Требуется анализ рисков, приоритезация, понимание контекста проекта	Решение, какие модули тестировать в первую очередь, зависит от бизнес-целей — алгоритм этого не знает
Разработка тест-кейсов	Нужно творческое мышление, понимание пользовательских сценариев	Придумывание edge-case'ов (например, «что если ввести 10 000 пробелов в поле имени?») — задача для человека
Написание отчётов о дефектах	Дефект-репорт должен быть понятен разработчику и содержать контекст	Автоматизированный лог «ошибка 500» не объясняет, почему это произошло и как воспроизвести
Оценка UX / удобства использования	Требуется субъективное восприятие: удобство, интуитивность, эстетика	Пользователь раздражается от задержки в 2 секунды — машина этого не почувствует

🧠 *Компьютер пока не умеет думать, сопереживать или интерпретировать неоднозначности*

## Технические и организационные ограничения

Ситуация	Проблема автоматизации	Пример из практики 
<b>Функциональность проверяется 1–2 раза</b>	Затраты на автоматизацию не окупятся	Фича для демо-версии, которая больше не обновляется — нет смысла писать скрипты
<b>Низкая стабильность требований</b>	Каждое изменение UI/логики ломает тесты	MVP-продукт, где интерфейс меняется еженедельно — автоматизация будет отставать
<b>Сложная технологическая комбинация</b>	Нет надёжных инструментов или их интеграция слишком трудоёмка	Приложение использует устаревший ActiveX + WebSocket + кастомный протокол — нет готовых драйверов
<b>Слабые возможности инструментов</b>	Нет логирования, отладки, доступа к состоянию приложения	Инструмент может только кликать — но не видит, что происходит «внутри» системы

 *Автоматизация хаоса = автоматизированный хаос*

## Организационные риски

Ситуация	Проблема автоматизации	Пример из практики 
<b>Проблемы с базовым ручным тестированием</b>	Если ручное тестирование не налажено, автоматизация усугубит проблемы	Команда не может воспроизвести баги — автоматизированные тесты будут падать без понятной причины
<b>Нехватка времени / срыв сроков</b>	Автоматизация требует времени на разработку и отладку	За 3 дня до релиза начинать писать тесты — бесполезно; лучше протестировать вручную
<b>Ожидание мгновенного эффекта</b>	Первые недели автоматизация «съедает» ресурсы, а не экономит их	Менеджер ждёт, что после внедрения Selenium все тесты будут проходить сами — но никто не написал ни одного сценария

### Афоризм:

*«Лучше руками протестировать хоть что-то, чем автоматизированно протестировать ничего».*

## ✓ Автоматизируйте, когда:

- Тесты запускаются часто
- Требования стабильны
- Есть чёткие ожидаемые результаты
- ROI положительный

## × Не автоматизируйте, когда:

- Тест нужен один раз
- Требования «плавают»
- Нет времени на поддержку
- Задача требует субъективной оценки

Автоматизация — это инвестиция, а не волшебная кнопка.

Успех зависит не от технологии, а от понимания контекста.

# **Особенности тест-кейсов для автоматизации**

# Главная проблема тест-кейсов для автоматизации

Компьютер не понимает «интуитивно» — он понимает только точно.

Проблема	Последствия 
Тест-кейсы, написанные для человека, содержат описания вроде: <i>«Проверить, что форма работает корректно»</i> или <i>«Убедиться, что всё выглядит нормально»</i>	Такие формулировки <b>непригодны для автоматизации</b> — они не содержат: <ul style="list-style-type: none"><li>• точных шагов</li><li>• конкретных входных данных</li><li>• однозначных критериев результата</li></ul>
Специалисты по автоматизации <b>не могут и не должны</b> интерпретировать неоднозначные инструкции	Это приводит к: <ul style="list-style-type: none"><li>• задержкам</li><li>• ошибкам в реализации тестов</li><li>• конфликтам между тестировщиками и автоматизаторами</li></ul>

# Главная проблема тест-кейсов для автоматизации

Компьютер не понимает «интуитивно» — он понимает только точно.

## Решение:

Тест-кейс для автоматизации — это **мини-техническое задание**, а не описание для человека.

Он должен содержать:

- Точные шаги (например, `click(button#submit)`)
- Конкретные данные ( `login = "user@test.com"` , `password = "Pass123!"` )
- Чёткие ожидаемые результаты ( `HTTP 200` ,  
`element #welcome-text contains "Здравствуйте"` )

 *Автоматизируемый тест-кейс пишется не для того, чтобы «понять», а чтобы «выполнить».*

## Тест-кейсы, пригодные для автоматизации

Автоматизируемый тест-кейс — это инструкция для машины, а не для человека.

### ✓ Ключевые принципы:

Принцип	Пример	↓
Чёткий ожидаемый результат	<p>× «Загружается стандартная страница поиска»</p> <p>✓ «title = "Search page", присутствует форма с input[type="text"] и кнопкой "Go!", отображается logo.jpg»</p>	
Независимость от других тестов	<p>× «Из файла, созданного предыдущим тестом...»</p> <p>✓ «Создать временный файл → выполнить операцию → проверить его содержимое»</p>	
Платформенно- и инструментонезависимость	<p>× «Отправить WM_CLICK окну»</p> <p>✓ «Передать фокус в активное окно и эмулировать клик левой кнопкой мыши»</p>	
Избегайте хардкода	<p>× «Открыть http://application/»</p> <p>✓ «Открыть главную страницу (URL определяется из конфигурации)»</p>	

### 📌 Правило:

Если шаг нельзя однозначно исполнить программой — его нужно переписать.

# Рекомендации по реализации автоматизированных тестов

Хорошая автоматизация — это не только работающий код, но и легко поддерживаемый.

## ✓ Лучшие практики:

Проблема	Плохо	Хорошо	↓
Синхронизация с приложением	«Кликнуть → сразу выбрать из списка»	«Кликнуть → дождаться, пока список станет <i>enabled</i> → выбрать значение»	
Способ взаимодействия	«Отправить <code>WM_KEY_DOWN/UP</code> события»	«Эмулировать ввод с клавиатуры (не через буфер!)»	
Универсальность	Завязка на конкретный UI-фреймворк или ОС	Использование стандартных интерфейсов (HTTP, DOM, REST, CLI)	
Поддерживаемость	Жёсткая привязка к ID элементов, которые часто меняются	Использование устойчивых локаторов (например, <code>data-testid</code> , семантические XPath/CSS)	

# Рекомендации по реализации автоматизированных тестов

---

Хорошая автоматизация — это не только работающий код, но и легко поддерживаемый.

## Помните:

Автоматизированный тест должен:


- работать **стабильно** на разных средах,
- **не ломаться** при мелких изменениях UI,
- быть **понятным** другому инженеру через 6 месяцев.

 *Цель автоматизации — не просто «запустить», а «запускать надёжно и долго».*

# Источники данных для автоматизированных тест-кейсов

Хороший автоматизированный тест не содержит «зашитых» значений — он получает данные из внешних, управляемых источников.

## ✓ Рекомендуемые источники данных:

Источник	Пример использования	Преимущества 
<b>Конфигурационные файлы</b> (JSON, YAML, XML, .env)	URL приложения, учётные данные, таймауты	Легко изменять без правки кода; поддержка разных окружений (dev / test / prod)
<b>Тестовые наборы данных</b> (CSV, Excel, JSON)	Наборы входных значений для параметризованных тестов (например, валидные/невалидные email)	Поддержка data-driven testing; легко расширять
<b>Базы данных</b>	Чтение тестовых пользователей или сценариев из БД перед запуском	Централизованное управление данными; актуальность
<b>Генераторы данных</b> (Faker, custom scripts)	Генерация уникальных имён, email, номеров карт и т.д.	Избегание конфликтов (например, дубликатов); поддержка масштабирования
<b>Внешние API / сервисы</b>	Получение временных email, SMS-кодов, токенов	Интеграция с реальными условиями тестирования

### ⊘ Избегайте:

- Хардкода ( `login = "admin"` прямо в коде)
- Зависимости от состояния после других тестов
- Использования «магических» констант без пояснения

### 💡 Правило:

*Логика теста — в коде. Данные — вне кода.*

**Разделение логики и данных** значительно упрощает поддержку, локализацию и повторное использование тестов.

# **Технологии автоматизации тестирования**

## Эволюция подходов к автоматизации тестирования

№	Подход	Суть
1	Частные решения	Для каждой задачи пишется отдельная программа
2	Тестирование под управлением данными (DDT)	Входные данные и ожидаемые результаты выносятся во внешние источники
3	Тестирование под управлением ключевыми словами (KDT)	Поведение теста описывается через ключевые слова; логика хранится вне кода
4	Использование фреймворков	Комплексное решение, объединяющее модульность, данные, отчётность и управление
5	Запись и воспроизведение (Record & Playback)	Действия пользователя записываются и воспроизводятся инструментом
6	Тестирование под управлением поведением (BDT)	Тесты формулируются как бизнес-сценарии на естественном языке

# Функциональная декомпозиция

# Что такое функциональная декомпозиция?

**Функциональная декомпозиция** — это разбиение сложной задачи на набор простых, переиспользуемых подзадач (функций).

## ♦ Цель в автоматизации тестирования:

- Упростить проектирование и поддержку тестов
- Повысить читаемость кода
- Обеспечить **повторное использование** низкоуровневых действий

## ♦ Где применяется:

- В архитектуре тестовых фреймворков (Page Object, Service Layer)
- При создании библиотек ключевых слов (KDT)
- В модульных и интеграционных тестах

## 🧠 Аналогия из жизни:

Готовка блюда → разбивается на шаги: нарезать, обжарить, заправить.

Каждый шаг можно использовать в других рецептах!

💡 Декомпозиция — основа масштабируемой и поддерживаемой автоматизации.

## Задача высокого уровня:

✓ Выполнить авторизацию

## Пример функциональной декомпозиции в тестировании

### Декомпозиция на подфункции:

```
1 Произвести авторизацию
2 |— Ввести имя пользователя
3 |   |— Найти поле "Username"
4 |   |— Заполнить поле
5 |— Ввести пароль
6 |   |— Найти поле "Password"
7 |   |— Заполнить поле
8 |— Отправить данные
9 |   |— Найти кнопку "Login"
10 |   |— Нажать кнопку
11 |— Проверить результат
12 |   |— Найти элемент с приветствием
13 |   |— Сравнить текст с ожидаемым
```

### 🔄 Повторное использование:

- Найти поле + Заполнить поле → используются также при регистрации, восстановлении пароля, оформлении заказа
- Найти кнопку + Нажать кнопку → универсальны для любого UI-взаимодействия

# Функциональная декомпозиция в тестировании

## Повторное использование:

- **Найти поле** + **Заполнить поле** → используются также при регистрации, восстановлении пароля, оформлении заказа
- **Найти кнопку** + **Нажать кнопку** → универсальны для любого UI-взаимодействия

## Результат:

- Тест на авторизацию становится короче и понятнее
- Изменение логики поиска элемента — правится **в одном месте**
- Новые сценарии собираются как конструктор из готовых блоков

 Хорошая автоматизация строится из «кирпичиков», а не монолитов.

# **Частные решения (Ad Hoc Automation)**

### Описание

Для каждой задачи пишется отдельный скрипт или программа «здесь и сейчас».

### Ключевые черты

- Минимум планирования
- Решает одну конкретную проблему
- Нет общей структуры

### ✓ Плюсы:

- Быстро реализуется
- Просто для понимания

### × Минусы:

- Нет повторного использования
- Высокая стоимость поддержки
- Трудно масштабировать

⚠ Подходит только для разовых проверок или прототипирования.

# **Тестирование под управлением данными (DDT)**

# Тестирование под управлением данными (DDT)

## Описание

Логика теста фиксирована, а входные и ожидаемые данные берутся из внешних источников (CSV, JSON, Excel).

## Ключевые черты

- Один сценарий → много наборов данных
- Данные отделены от кода
- Идеален для валидации форм, API, расчётов

## Тестирование под управлением данными (DDT)

Преимущества	Недостатки
<ul style="list-style-type: none"><li>• Устранение избыточности кода тест-кейсов.</li><li>• Удобное хранение и понятный человеку формат данных.</li><li>• Возможность поручения генерации данных сотруднику, не имеющему навыков программирования.</li><li>• Возможность использования одного и того же набора данных для выполнения разных тест-кейсов.</li><li>• Возможность повторного использования набора данных для решения новых задач.</li><li>• Возможность использования одного и того же набора данных в одном и том же тест-кейсе, но реализованном под разными платформами.</li></ul>	<ul style="list-style-type: none"><li>• При изменении логики поведения тест-кейса его код всё равно придётся переписывать.</li><li>• При неудачно выбранном формате представления данных резко снижается их понятность для неподготовленного специалиста.</li><li>• Необходимость использования технологий генерации данных.</li><li>• Высокая сложность кода тест-кейса в случае сложных неоднородных данных.</li><li>• Риск неверной работы тест-кейсов в случае, когда несколько тест-кейсов работают с одним и тем же набором данных, и он был изменён таким образом, на который не были рассчитаны некоторые тест-кейсы.</li><li>• Слабые возможности по сбору данных в случае обнаружения дефектов.</li><li>• Качество тест-кейса зависит от профессионализма сотрудника, реализующего код тест-кейса.</li></ul>

# **Тестирование под управлением ключевыми словами (KDT)**

# Тестирование под управлением ключевыми словами (KDT)

## Описание

Тесты описываются через действия вроде `Login`, `ClickButton`, `VerifyText` — как «словарь» операций.

## Ключевые черты

- Поведение задаётся вне кода (например, в таблице или YAML)
- Нетехнические специалисты могут писать сценарии
- Реализация действий скрыта в библиотеках

## ✓ Плюсы:

- Гибкость и повторное использование
- Читаемость для бизнес-пользователей

## × Минусы:

- Сложно реализовать низкоуровневые или уникальные действия
- Требуется тщательного проектирования «словаря»

 Хорош при наличии стабильного UI и чёткой функциональности.

**Selenium IDE** — один из самых наглядных примеров подхода *Keyword-Driven Testing* в веб-автоматизации.

### ♦ Как это работает:

- Каждое действие записывается как **команда (ключевое слово)**:

`click`, `type`, `verifyText`, `select`, `assertElementPresent` и т.д.

- Команды применяются к **целевым элементам** (локаторам): `id=login`, `css=.submit-btn`
- Тест выглядит как **таблица или сценарий** из трёх колонок:

### **Command | Target | Value**

Command	Target	Value	↓
open	/login		
type	id=username	testuser	
type	id=password	secret123	
click	id=submit		
verifyText	css=.welcome-msg	Добро пожаловать!	

### ✓ Почему это KDT?

- Логика отделена от реализации: команда `click` скрывает всю сложность WebDriver
- Тест читаем даже нетехническим специалистам
- Легко редактировать, копировать, переиспользовать шаги

### ⚠ Ограничения:

- Подходит в основном для линейных сценариев
- Мало возможностей для сложной логики (циклы, условия) без расширений
- Лучше всего работает при стабильной разметке UI

### ✍ Идеальное применение:

Быстрое создание прототипов, обучение, smoke-тесты, документирование сценариев

# Фреймворки

## Описание

Целостная архитектура для автоматизации: объединяет DDT, KDT, управление окружением, отчётность, логирование.

## Ключевые черты

- Модульная структура (Page Object, Service Layers и др.)
- Поддержка нескольких платформ и окружений
- Встроенные механизмы восстановления и анализа

## ✓ Плюсы:

- Масштабируемость
- Долгосрочная поддерживаемость
- Командная работа

## × Минусы:

- Высокие первоначальные затраты
- Требуется опыт в проектировании

## Фреймворки для модульного тестирования

Язык	Фреймворк	Особенности	↓
Java	JUnit, TestNG	Аннотации ( <code>@Test</code> ), assertions, параметризованные тесты, интеграция с Maven/Gradle	
Python	pytest, unittest	Простой синтаксис, мощная параметризация (pytest), встроен в стандартную библиотеку (unittest)	
C# / .NET	NUnit, xUnit, MSTest	Поддержка атрибутов, параллельный запуск, глубокая интеграция с Visual Studio	
JavaScript	Jest, Mocha, Jasmine	Работа с асинхронным кодом, моки, встроенный runner и отчёты	
C++	Google Test (gtest), Catch2	Поддержка макросов, параметризованных тестов, гибкая настройка	

## Фреймворки для модульного тестирования

- Автоматический запуск и отчётность
- Утверждения (assertions) для проверки ожидаемых результатов
- Изоляция тестов (setup/teardown)
- Поддержка моков и заглушек (через дополнительные библиотеки)

 **Unit-тесты — основа надёжного кода.**

Они быстрые, дешёвые в поддержке и идеально подходят для автоматизации в CI/CD.

# **Запись и воспроизведение (Record & Playback)**

# Запись и воспроизведение (Record & Playback)

## Описание

Инструмент записывает действия пользователя (клики, ввод) и генерирует скрипт для повторного запуска.

## Ключевые черты

- Минимум программирования
- Быстрое создание первого теста
- Жёсткая привязка к UI и координатам

## ✓ Плюсы:

- Очень быстро
- Подходит для демо или PoC

## × Минусы:

- Хрупкие тесты (ломаются при любом изменении интерфейса)
- Нечитаемый код
- Почти невозможно поддерживать

⚠ Используйте только для быстрой демонстрации — не для реальных проектов!

# Selenium IDE — Record & Playback «из коробки»

**Selenium IDE** — бесплатное расширение для браузеров (Chrome, Firefox), реализующее классическую технологию *записи и воспроизведения*.

## Как это работает?

1. Пользователь **выполняет действия** в браузере (открывает страницу, вводит текст, кликает)
2. Инструмент **автоматически записывает** каждое действие как команду
3. Полученный сценарий можно **немедленно воспроизвести**, отредактировать или экспортировать

## Пример записанного шага:

Command	Target	Value	↓
open	/login		
type	id=username	test@example.com	
click	id=submit-btn		

### ✓ Преимущества:

- Мгновенное создание тестов без программирования
- Визуальный редактор с подсветкой ошибок
- Экспорт в WebDriver-совместимые языки (Python, Java, C#, JavaScript)
- Идеален для обучения, прототипирования и smoke-тестов

### ⚠ Ограничения:

- Минимальная поддержка логики (условия, циклы — только через плагины)
- Хрупкость при изменениях UI (локаторы часто ломаются)
- Не подходит для сложных или масштабных тестовых наборов

💡 Selenium IDE — отличная «точка входа» в автоматизацию, но не финальное решение для промышленных проектов.

# **Тестирование под управлением поведением (BDT)**

# Тестирование под управлением поведением (BDT)

## Описание

Тесты описываются как бизнес-сценарии на естественном языке (например, Gherkin: *Given-When-Then*).

## Ключевые черты

- Акцент на пользовательских потоках
- Совместная работа QA, аналитиков и заказчиков
- Инструменты: Cucumber, SpecFlow, Behave

## ✓ Плюсы:

- Живая документация
- Понятна нетехническим участникам
- Фокус на ценности для пользователя

## × Минусы:

- Не проверяет детали реализации
- Может пропустить технические дефекты
- Требует дублирования низкоуровневыми тестами



BDT — мост между бизнесом и разработкой, но не замена техническому тестированию.

**Gherkin** — это предметно-ориентированный язык (DSL) для описания тестовых сценариев на структурированном, естественном языке. Он использует фиксированный набор ключевых слов и строгую иерархию, чтобы сценарии были одновременно понятны людям и машинам. Файлы обычно сохраняются с расширением `.feature`.

Основные ключевые слова:


Ключевое слово	Назначение	↓
<code>Feature</code>	Описание тестируемой функциональности	
<code>Scenario</code> / <code>Scenario Outline</code>	Конкретный тестовый кейс / параметризованный кейс	
<code>Given</code>	Предусловие (начальное состояние)	
<code>When</code>	Действие пользователя или системы	
<code>Then</code>	Ожидаемый результат (проверка)	
<code>And</code> / <code>But</code>	Логические связки для дополнительных шагов	
<code>Background</code>	Шаги, повторяющиеся перед каждым сценарием в файле	
<code>Examples</code>	Таблица данных для <code>Scenario Outline</code>	

## Для чего нужен Gherkin?

1. **Единый язык коммуникации** — бизнес-аналитики, менеджеры, разработчики и тестировщики читают один и тот же текст без необходимости переводить требования в код.
2. **Живая документация** — `.feature` файлы всегда синхронизированы с реальным поведением системы, так же как и код.
3. **Основа для автоматизации** — сценарии напрямую привязываются к исполняемому коду через BDD-фреймворки.
4. **Фокус на поведении, а не на реализации** — описывается *что* должна делать система, а не *как* это технически сделано.

## Связь с BDD / BDT

- **BDD (Behavior-Driven Development)** — методология разработки, где требования формулируются как ожидаемое поведение системы, а тесты пишутся до или вместе с кодом.
- **BDT (Behavior-Driven Testing)** — тестовая составляющая BDD: проверка того, что система ведёт себя именно так, как описано в сценариях.
- **Gherkin** — это *язык*, на котором записываются сценарии BDD/BDT.
- **BDD-фреймворк** (Cucumber, Behave, SpecFlow, Playwright BDD и др.) — это *движок*, который парсит Gherkin, находит соответствующий код (step definitions) и запускает его.

 **Аналогия:** BDD/BDT — это методология проектирования зданий, Gherkin — чертёж на понятном языке, а BDD-фреймворк — бригада строителей, которая превращает чертёж в реальность.

## Простой пример (авторизация)

gherkin



```
1 Feature: Авторизация пользователя
2   Чтобы получить доступ к личному кабинету, зарегистрированный пользователь
3
4   Background:
5     Given я нахожусь на странице входа "https://example.com/login"
6
7   Scenario: Успешный вход с корректными данными
8     When я ввожу логин "testuser" и пароль "secret123"
9     And нажимаю кнопку "Войти"
10    Then я должен быть перенаправлен на главную страницу личного кабинета
11    And я вижу текст приветствия "Добро пожаловать, testuser!"
12
13   Scenario Outline: Вход с некорректными данными
14     When я ввожу логин "<login>" и пароль "<password>"
15     And нажимаю кнопку "Войти"
16     Then я вижу сообщение об ошибке "<error_message>"
17
18   Examples:
19     | login      | password | error_message |
20     | testuser   | wrong    | Неверный пароль |
21     |            | secret   | Поле логина не может быть пустым |
```

## ⚙️ Как это работает под капотом?

1. Вы создаёте `.feature` файл с сценариями на Gherkin.
2. BDD-фреймворк парсит файл и ищет для каждого шага ( `Given/When/Then` ) **step definition** — функцию на Python/Java/JS и т.д.
3. В step definition обычно вызывается Selenium, Playwright, API-клиент или моки.
4. При запуске фреймворк последовательно выполняет привязанный код, подставляя параметры из `Examples` .
5. Генерируется отчёт: зелёный/красный статус, логи, скриншоты (при падении).

## Пример привязки шага (Python + Behave):

python



```
1 from behave import given, when, then
2 from selenium.webdriver.common.by import By
3
4 @given('я нахожусь на странице входа "{url}"')
5 def step_open_login(context, url):
6     context.driver.get(url)
7
8 @when('я ввожу логин "{login}" и пароль "{password}"')
9 def step_fill_credentials(context, login, password):
10     context.driver.find_element(By.ID, "username").send_keys(login)
11     context.driver.find_element(By.ID, "password").send_keys(password)
12
13 @when('нажимаю кнопку "Войти"')
14 def step_click_login(context):
15     context.driver.find_element(By.CSS_SELECTOR, "button[type=submit]").click()
16
17 @then('я вижу текст приветствия "{text}"')
18 def step_check_greeting(context, text):
19     assert context.driver.find_element(By.ID, "greeting").text == text
```

## ✅ Когда стоит использовать Gherkin + BDD

- В командах, где важны прозрачность требований и кросс-функциональное взаимодействие
- При автоматизации E2E-сценариев, которые будут регулярно читаться не-техническими ролями
- В проектах с долгой поддержкой, где нужна «живая» документация вместо устаревающих Wiki/Confluence

## ⚠️ Когда лучше обойтись без него

- Простые unit- или интеграционные тесты (там Gherkin даёт только оверхед)
- Команды без аналитиков/РО, которые не готовы поддерживать `.feature` файлы
- Проекты, где требования меняются ежедневно (синхронизация Gherkin → код станет bottleneck)

Концепция	Роль	↓
BDD/BDT	Методология: фокус на поведении и коллаборации	
Gherkin	Язык: структурированное описание сценариев	
Step Defs	Код: привязка слов к Selenium/Playwright/API	
Runner	Движок: Cucumber, Behave, SpecFlow и др.	

Gherkin не запускает тесты сам по себе — он лишь контракт между бизнесом и автоматизацией. В паре с BDD-фреймворком и Selenium/Playwright он превращает текстовые требования в исполняемые, поддерживаемые и понятные всем участникам команды проверки.

# **Selenium WebDriver**

**Selenium WebDriver** — это программный интерфейс (API) для управления веб-браузерами из кода. Является ядром экосистемы Selenium и реализует открытый стандарт **W3C WebDriver**. В отличие от Selenium IDE, WebDriver не имеет графической оболочки: вы пишете тесты на полноценном языке программирования (Python, Java, C#, JavaScript, Ruby и др.), а WebDriver транслирует ваши команды в действия браузера.

**Selenium WebDriver** — не просто инструмент, а **гибкий фреймворк** для автоматизации веб-приложений, лежащий в основе большинства enterprise-решений.

## Какие подходы он объединяет?

- **Частные решения** → позволяет писать кастомные скрипты под любую задачу
- **Тестирование под управлением данными (DDT)** → легко интегрируется с CSV, JSON, Excel
- **Функциональную декомпозицию** → поддерживает модульность (Page Object, Service Layers)
- **Ключевые слова (KDT)** → через обёртки и библиотеки (например, создание методов `login()`, `search()`)
- **Поведенческое тестирование (BDT)** → совместим с Cucumber, SpecFlow и Gherkin

WebDriver не тестирует сам — он даёт программе возможность управлять браузером так, как это делает человек.

### ♦ Что умеет WebDriver «из коробки»?

- Открывать страницы
- Находить элементы (по ID, CSS, XPath и др.)
- Кликать, вводить текст, выбирать из выпадающих списков
- Получать свойства элементов (текст, атрибуты, видимость)
- Управлять окнами, cookies, alert'ами

### ♦ Чего НЕТ в WebDriver'е?

- Встроенной логики тестов (assert'ов, условий, циклов)
- Механизмов отчётности
- Управления данными
- Повторного запуска упавших тестов
- Параллельного выполнения

✓ Поэтому WebDriver всегда используется вместе с:

- **Языками программирования:** Python, Java, C#, JavaScript — для реализации логики
- **Тестовыми фреймворками:** Pytest, JUnit, TestNG, NUnit — для организации, запуска и отчётов
- **Дополнительными библиотеками:** Allure (отчёты), Faker (данные), Requests (API), Docker (окружение)

🧩 WebDriver — это «движок», а не «машина».

```
1 from selenium import webdriver
2 from selenium.webdriver.common.by import By
3 from selenium.webdriver.support.ui import WebDriverWait
4 from selenium.webdriver.support import expected_conditions as EC
5
6 # 1. Инициализация браузера (Chrome)
7 driver = webdriver.Chrome()
8
9 try:
10     # 2. Открыть страницу
11     driver.get("https://demo.selenium.dev/")
12
13     # 3. Дождаться появления поля ввода и ввести текст
14     wait = WebDriverWait(driver, timeout=10)
15     search_input = wait.until(EC.presence_of_element_located((By.ID, "search")))
16     search_input.send_keys("WebDriver")
17
18     # 4. Найти и кликнуть по кнопке поиска
19     search_button = driver.find_element(By.CSS_SELECTOR, "button[type='submit']")
20     search_button.click()
21
22     # 5. Проверить, что URL изменился (или появился ожидаемый элемент)
23     wait.until(EC.url_contains("/search"))
24     print(f"✅ Успех! Текущий URL: {driver.current_url}")
25
26 finally:
27     # 6. Закрывать браузер и освободить ресурсы
28     driver.quit()
```

## ✓ Для каких видов тестирования подходит?

Вид тестирования	Поддержка
Функциональное	✓ Полная (UI, формы, навигация)
Регрессионное	✓ Идеален для частых повторных запусков
Интеграционное	✓ Через взаимодействие с API + UI
Кросс-браузерное	✓ Chrome, Firefox, Edge, Safari и др.
Smoke / sanity	✓ Быстрый запуск критических сценариев
End-to-end (E2E)	✓ Реалистичные пользовательские потоки

## ⚠ Не подходит для:

- Тестирования производительности
- Глубокого тестирования безопасности
- Проверки UX/доступности без дополнительных инструментов

Selenium WebDriver — это платформа, а не «готовое решение»

✓ Именно поэтому он стал *де-факто стандартом* в автоматизации веб-тестирования:

- **Открытый и кроссплатформенный** — поддерживает все основные браузеры и ОС
- **Языконезависимый** — один и тот же протокол работает с Python, Java, C# и др.
- **Гибкий и расширяемый** — легко интегрируется в любую архитектуру тестирования
- **Поддерживается сообществом и индустрией** — W3C-стандарт, официальные драйверы от браузеров
- **Не навязывает подход** — можно строить DDT, KDT, BDT, Page Object — как нужно проекту

🏆 WebDriver не решает задачу за вас — он даёт вам инструменты, чтобы решить её правильно.

## Связь с Selenium IDE

Параметр	Selenium IDE	Selenium WebDriver	↓
Язык	Визуальный (команды)	Python, Java, JS, C#, Ruby	
Сложность	Низкая (запись/воспроизведение)	Средняя/высокая (полный контроль)	
Данные	Жёстко заданы в шагах	Внешние источники (CSV, БД, API, fixtures)	
CI/CD	Ограничено (через CLI)	Полная интеграция	
Масштаб	Прототипы, smoke	Enterprise-регрессия, E2E, нагрузочные цепочки	