АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

Автоматизация тестирования — лучший способ повысить эффективность, покрытие тестами и скорость выполнения тестирования ПО.

Цель автоматизации — сократить количество тестовых случаев, которые нужно запустить вручную, а не полностью исключить ручное тестирование

Автоматизированное тестирование ПО важно потому, что:

- ✓ Ручное тестирование всех рабочих процессов, всех полей, всех негативных сценариев требует много времени и денег.
- ✓ Трудно вручную тестировать многоязычные сайты.
- ✓ Автоматизация тестирования в тестировании ПО не требует человеческого вмешательства. Можно запустить автоматизированный тест без присмотра (в течение ночи).
- ✓ Автоматизация тестирования увеличивает скорость выполнения тестов.
- ✓ Автоматизация помогает увеличить охват тестированием.
- ✓ Ручное тестирование может стать скучным и, следовательно, подверженным ошибкам.

Какие тестовые случаи автоматизировать?

Тестовые случаи для автоматизации можно выбрать, используя следующий критерий для повышения рентабельности инвестиций в автоматизацию:

- ✓ **Критически важные для бизнеса** тестовые случаи.
- ✓ Тестовые случаи, **которые выполняются многократно**.
- ✓ Тестовые случаи, **которые очень утомительны** или сложны для выполнения вручную.
- ✓ Тестовые случаи, **требующие много времени**.

Следующая категория тестовых случаев не подходит для автоматизации:

- ✓ Тестовые случаи, которые были разработаны заново и не были выполнены вручную хотя бы один раз.
- ✓ Тестовые случаи, требования к которым часто меняются.
- ✓ Тестовые случаи, которые выполняются на индивидуальной основе.

ПРОЦЕСС АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ

Шаг 1) Выбор инструмента тестирования

Шаг 2) Определите область автоматизации

Шаг 3) Планирование, проектирование и разработка

Шаг 4) Выполнение теста

Шаг 5) Техническое обслуживание

Шаг 1. Выбор инструмента тестирования

Выбор инструмента тестирования во многом зависит от технологии, на которой построено тестируемое приложение.

Шаг 2. Определите область автоматизации

Область автоматизации — это область вашего тестируемого приложения, которая будет автоматизирована. Следующие пункты помогают определить область:

- ✓ Функции, которые важны для бизнеса
- ✓ Сценарии с **большим объемом данных**
- ✓ **Общие функции** для всех приложений
- ✓ Техническая осуществимость
- ✓ Степень повторного использования бизнес-компонентов
- ✓ **Сложность** тестовых случаев
- ✓ Возможность использования одних и тех же тестовых случаев для кросс-браузерного тестирования.

Шаг 3. Планирование, проектирование и разработка

На этом этапе вы создаете стратегию и план автоматизации, которые содержат следующую информацию:

- ✓ Выбраны инструменты автоматизации
- ✓ Конструкция фреймворка и ее особенности
- ✓ Элементы автоматизации, входящие и выходящие за рамки
- ✓ Подготовка испытательного стенда автоматизации
- ✓ График и хронология написания и выполнения сценариев
- ✓ Результаты автоматизированного тестирования

Шаг 4. Выполнение теста

На этом этапе автоматизации выполняются скрипты.

Скриптам нужны входные тестовые данные, прежде чем они будут установлены для запуска. После выполнения они предоставляют подробные тестовые отчеты.

Выполнение может осуществляться напрямую с помощью инструмента автоматизации или через инструмент управления тестированием, который вызовет инструмент автоматизации.

> *Пример: Центр качества — это инструмент управления тестированием, который, в свою очередь, вызывает QTP (*QuickTest Professional*) для выполнения скриптов автоматизации.*

Скрипты могут быть выполнены на одной машине или на группе машин.

Выполнение может быть выполнено ночью, чтобы сэкономить время.

Шаг 5. Тестирование подхода к обслуживанию автоматизации

Подход к обслуживанию автоматизации тестирования — это фаза тестирования автоматизации, проводимая для проверки того, нормально или нет работают новые функции, добавленные в программное обеспечение.

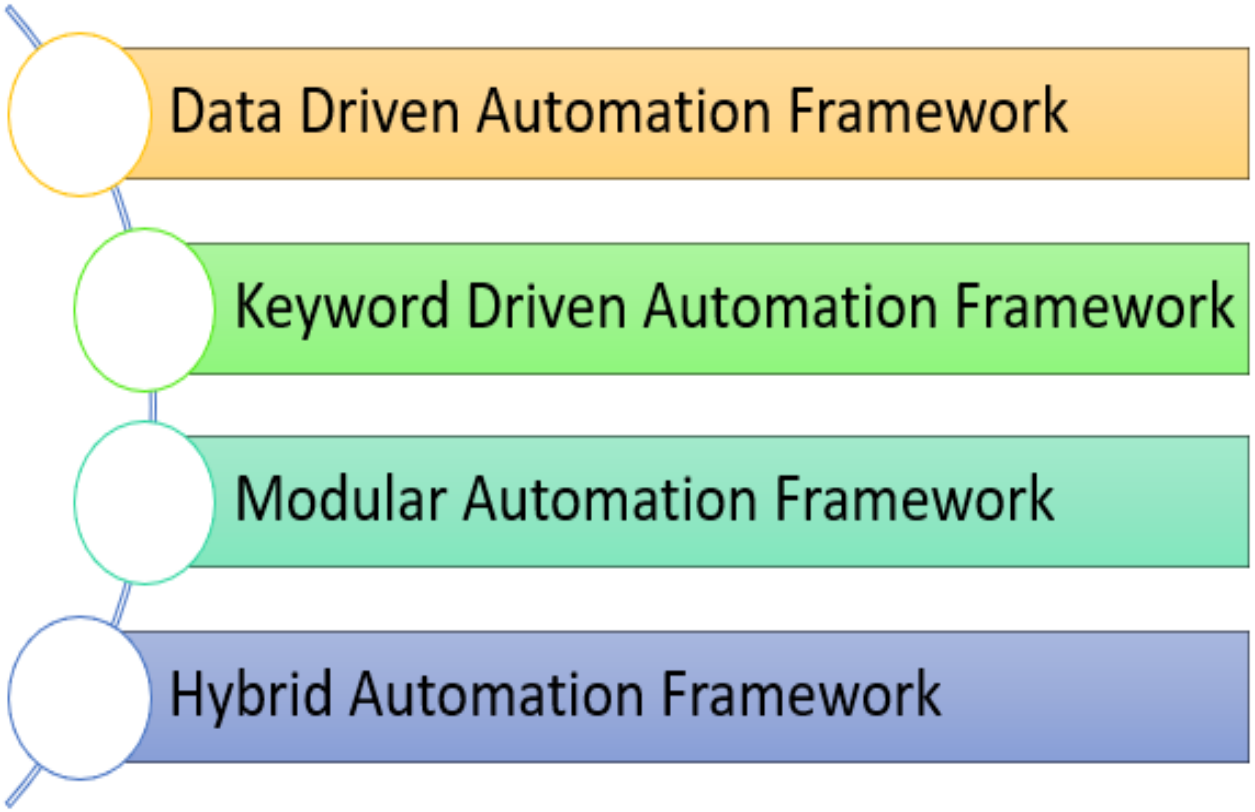
Обслуживание в тестировании автоматизации выполняется, когда добавляются новые сценарии автоматизации и их необходимо пересматривать и поддерживать, чтобы повысить эффективность сценариев автоматизации с каждым последующим циклом выпуска.

Фреймворк для автоматизации

Фреймворк — это набор руководств по автоматизации, которые помогают в:

- ✓ Поддержание последовательности тестирования
- ✓ Улучшает структурирование теста
- ✓ Минимальное использование кода
- ✓ Меньше обслуживания кода
- ✓ Улучшение возможности повторного использования
- ✓ Нетехнические тестировщики могут быть вовлечены в код
- ✓ Период обучения использованию инструмента может быть сокращен
- ✓ Включает данные везде, где это уместно

При автоматизированном тестировании ПО используются четыре типа фреймворков:



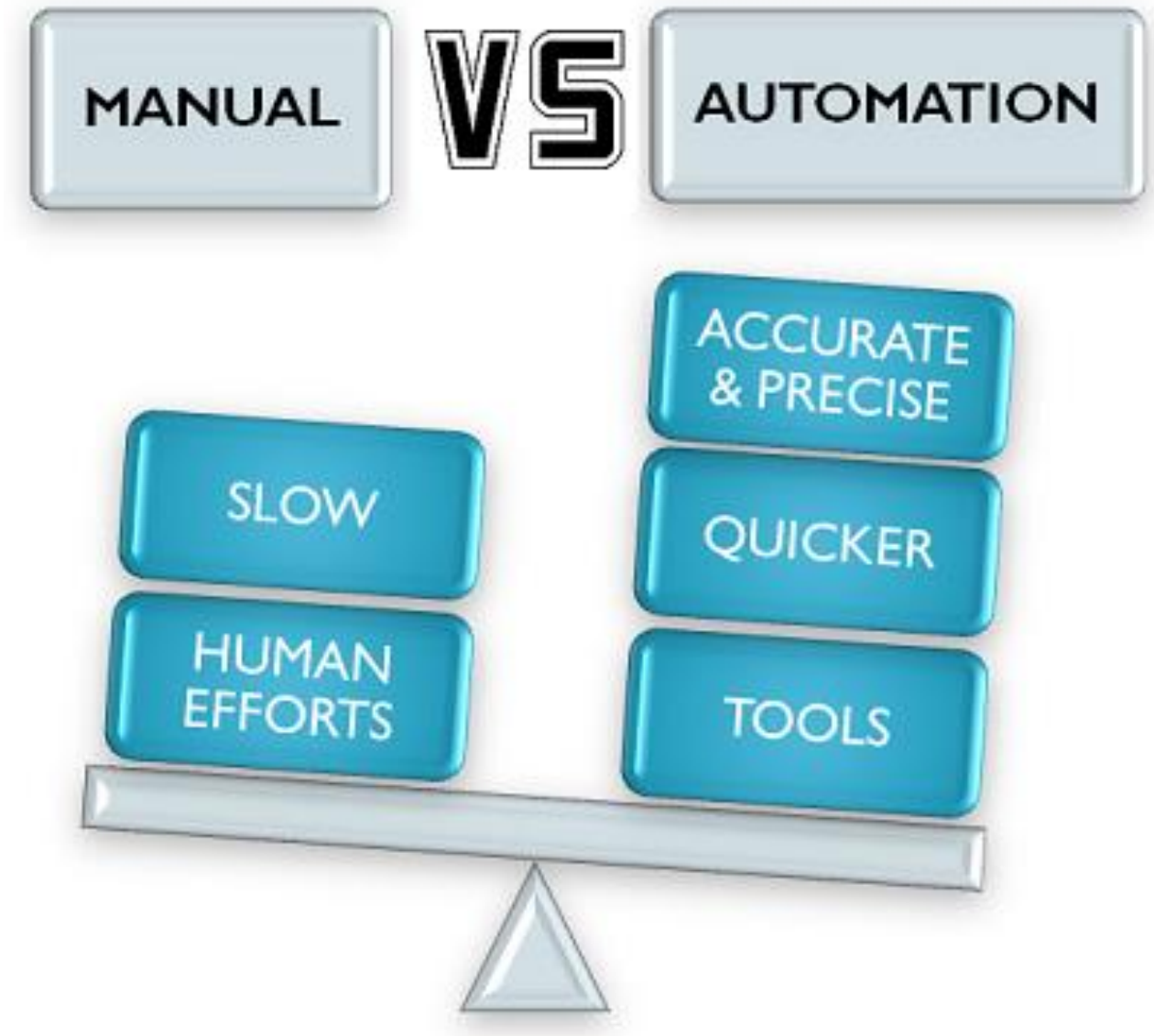
- Структура автоматизации на основе данных
- Фреймворк автоматизации на основе ключевых слов
- Модульная структура автоматизации
- Гибридная структура автоматизации

Преимущества автоматизации тестирования:

- На 70% быстрее ручного тестирования
- Более широкий охват тестирования функций приложения
- Надежные результаты
- Обеспечить последовательность
- Экономит время и деньги
- Повышает точность
- Вмешательство человека во время выполнения не требуется.
- Увеличивает эффективность
- Лучшая скорость выполнения тестов
- Тестовые сценарии, пригодные для повторного использования
- Тестирует часто и тщательно
- Раннее время выхода на рынок

Минусы автоматизированного тестирования:

- Без человеческого фактора **сложно получить представление о визуальных аспектах пользовательского интерфейса**, таких как цвета, шрифты, размеры, контрастность или размеры кнопок.
- Инструменты для проведения автоматизированного тестирования могут быть дорогими, что **может увеличить стоимость проекта** тестирования.
- Инструмент тестирования автоматизации пока не является полностью надежным. **Каждый инструмент автоматизации имеет свои ограничения**, что снижает область автоматизации.
- Отладка тестового скрипта — еще одна важная проблема в автоматизированном тестировании. **Поддержка тестов обходится дорого.**


MANUAL
VS
AUTOMATION
SLOW
HUMAN EFFORTS
ACCURATE & PRECISE
QUICKER
TOOLS

Параметр	Автоматизированное тестирование	Ручное тестирование
Время обработки	Автоматизированное тестирование значительно быстрее ручного подхода.	Ручное тестирование занимает много времени и требует человеческих ресурсов.
Исследовательское тестирование	Автоматизация не допускает выборочного тестирования	Исследовательское тестирование возможно в ручном тестировании
Первоначальные инвестиции	Первоначальные инвестиции в автоматизированное тестирование выше. Хотя окупаемость инвестиций в долгосрочной перспективе выше.	Первоначальные инвестиции в ручное тестирование сравнительно ниже. Окупаемость инвестиций ниже по сравнению с автоматизированным тестированием в долгосрочной перспективе.

Параметр	Автоматизированное тестирование	Ручное тестирование
Изменение пользовательского интерфейса	Даже для незначительного изменения пользовательского интерфейса AUT необходимо изменить сценарии автоматизированного тестирования, чтобы они работали так, как ожидается.	Небольшие изменения, такие как изменение идентификатора, класса и т. д. кнопки, не помешают выполнению ручного тестирования.
Инвестиции	Требуются инвестиции в инструменты тестирования, а также в инженеров по автоматизации.	Необходимы инвестиции в человеческие ресурсы.
Экономически эффективно	Невыгодно для регрессии малого объема	Невыгодно для регрессии большого объема.

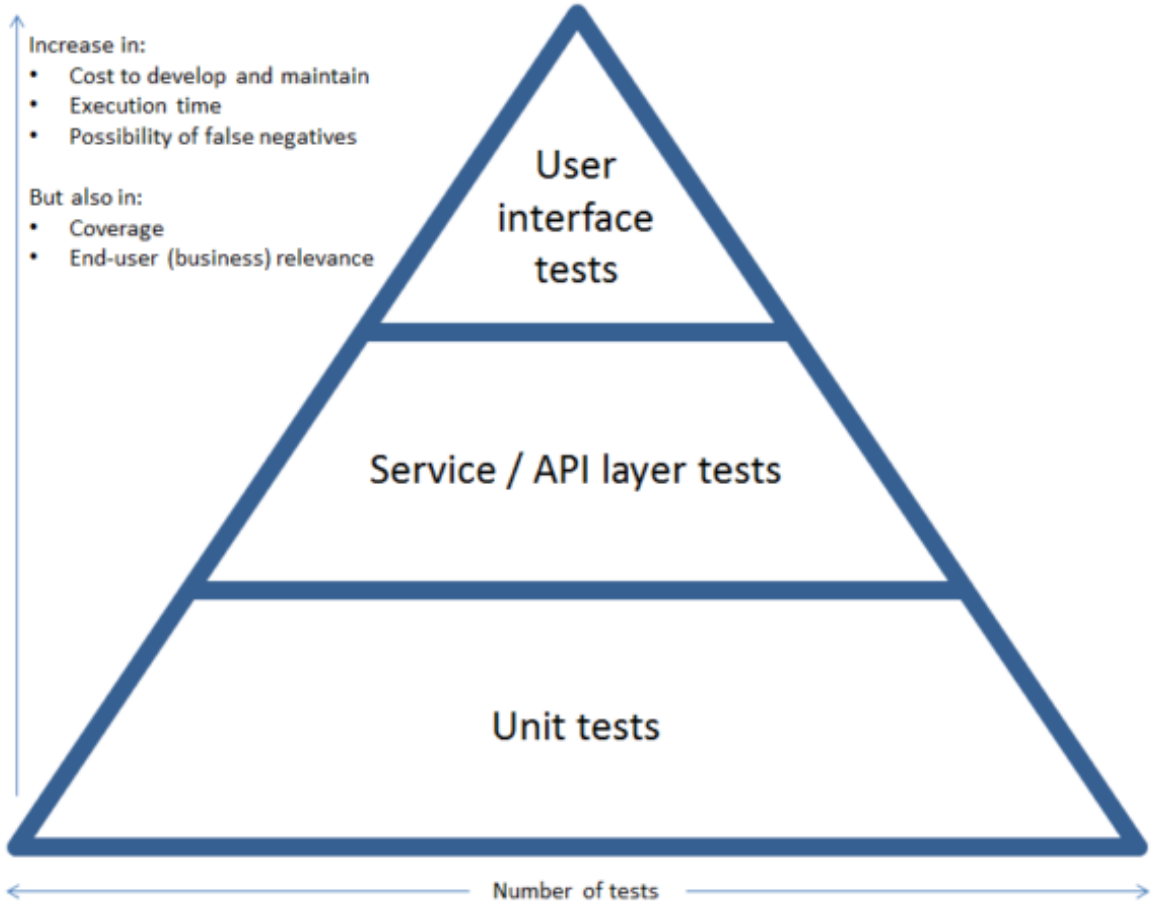
Параметр	Автоматизированное тестирование	Ручное тестирование
Тестирование производительности	Тесты производительности, такие как нагрузочное тестирование, стресс-тестирование, пиковое тестирование и т. д., должны в обязательном порядке тестироваться с помощью инструмента автоматизации.	Тестирование производительности невозможно выполнить вручную.
Параллельное выполнение	Это тестирование можно выполнять на разных операционных платформах параллельно, что сокращает время выполнения теста.	Ручные тесты можно выполнять параллельно, но для этого потребуется увеличить человеческие ресурсы, что является дорогостоящим.

Параметр	Автоматизированное тестирование	Ручное тестирование
Знание программирования	Знание программирования является обязательным условием при автоматизированном тестировании.	Нет необходимости в программировании при ручном тестировании.
Документация	Автоматизированные тесты действуют как документ, предоставляющий обучающую ценность, особенно для автоматизированных тестовых случаев. Новый разработчик может изучить тестовые случаи и быстро понять кодовую базу.	Ручные тестовые случаи не представляют никакой обучающей ценности

УРОВНИ АВТО-ТЕСТИРОВАНИЯ

- ✓ Уровень модульного тестирования (unit tests);
- ✓ Уровень функционального тестирование (non-UI tests);
- ✓ Уровень тестирования через пользовательский интерфейс (UI tests).

Пирамида автоматизации тестирования



МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

> Процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы (Unit –тесты).

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода.

Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

```
class CalculatorTests
{
        public void Sum_2Plus5_7Returned()
        {
                // arrange
                var calc = new Calculator();
                // act
                var res = calc.Sum(2,5);
                // assert
                Assert.AreEqual(7, res);
        }
}
```

ФРЕЙМВОРКИ для UNIT-ТЕСТОВ

NON-UI ТЕСТИРОВАНИЕ

> Процесс в программировании, позволяющий проверить работоспособность приложения используя программный интерфейс приложения (application programming interface, API).

Не всегда всю бизнес логику приложения можно протестировать через GUI слой. Это может быть особенностью реализации, которая прячет бизнес логику от пользователей. Именно по этой причине для команды тестирования может быть реализован доступ напрямую к функциональному слою, дающий возможность тестировать непосредственно бизнес логику приложения, минуя пользовательский интерфейс.

```
package CountriesRestTests;

…
public class GetTest {

@Test
public void getRequestFindCapital() throws JSONException {

// выполняем запрос get для доступа ко всем параметрам ответа
Response resp = get("http://restcountries.eu/rest/v1/name/belarus");

JSONArray jsonResponse = new JSONArray(resp.asString());

// получение параметра capital (столицы Беларуси)
String capital = jsonResponse.getJSONObject(0).getString("capital");

// проверка, что столицей является Минск
AssertJUnit.assertEquals(capital, “Minsk");
}
}
```

NON-UI ТЕСТИРОВАНИЕ: ИНСТРУМЕНТЫ

SoapUI

REST-assured

UI ТЕСТИРОВАНИЕ

Процесс в программировании, позволяющий проверить работоспособность и внешний вид приложения используя графический интерфейс приложения.

SELENIUM

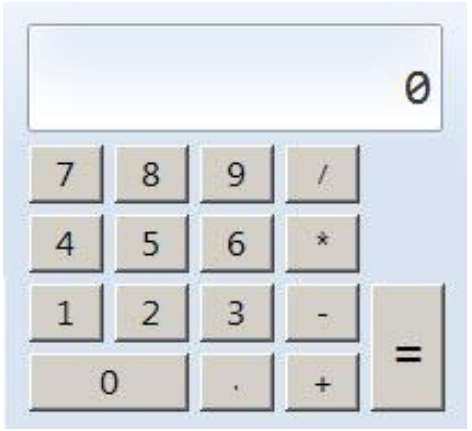
```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;
public class Example  {
    public static void main(String[] args) {
        WebDriver driver = new HtmlUnitDriver();
        driver.get("http://www.google.com");
        WebElement element = driver.findElement(By.name("q"));
        element.sendKeys("Lecture for automated testing!");
        element.submit();
        System.out.println("Page title is: " + driver.getTitle());
        driver.quit(); } }
```

ИНСТРУМЕНТЫ UI АВТОМАТИЗАЦИИ

ПОДХОДЫ В UI АВТОМАТИЗАЦИИ

- ✓ Data-driven подход;
- ✓ Keyword-driven подход;
- ✓ Behavior-driven подход.

DATA-DRIVEN ПОДХОД



2 + 3 = 5

4 x 2 = 8

Число 1 — Оператор — Число 2 — Результат

Число 1	Оператор	Число 2	Результат
5	-	1	4
6	+	3	9
32	/	4	8
3	x	7	21

KEYWORD-DRIVEN ПОДХОД

Включает в себя набор ключевых слов, которые можно повторно использовать в разных тестах. Ключевое слово – это комбинация действий пользователя с объектом теста.

Планирование + Реализация

- Создание карт объектов
- Создание набора ключевых слов

- Реализация фреймворка, обеспечивающего ключевые слова

Объект	Действие	Данные
LoginPage	Open	
InputLogin	enterText	user
InputPassword	enterText	password
ButtonLogin	click	
HomePage	assertLoggedIn	

BEHAVIOR-DRIVEN ПОДХОД

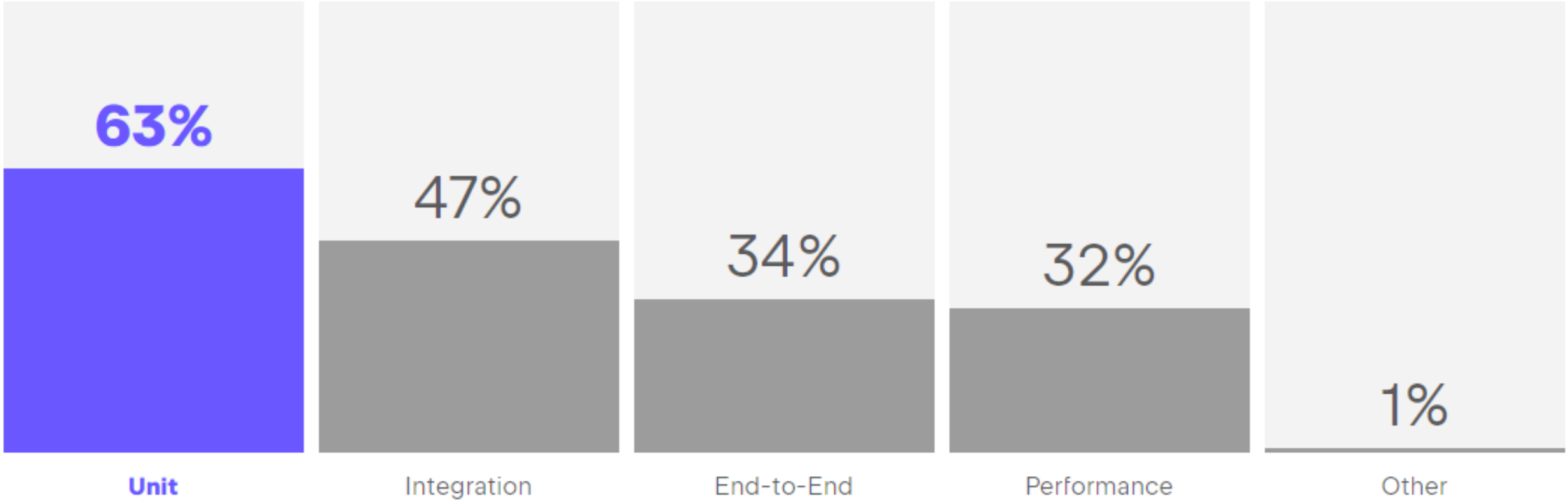
Это когда тесты пишутся на простом (не техническом) языке, который понятен и представителям бизнеса, и техническим специалистам.

```
Feature: Log in functionality

 Scenario: Successful login
   Given I am on Login page
   And User already saw classes tour
   When I login with username 'test@test.com' and password 'shadow123'
   Then I want to see 'Main' page
   And I want to see that I am logged in


 Scenario: Unsuccessful login
   Given I am on Login page
   And User already saw classes tour
   When I login with username 'abcd@ed.ba' and password '111111'
   Then I want to see error message "Invalid email or password."
```

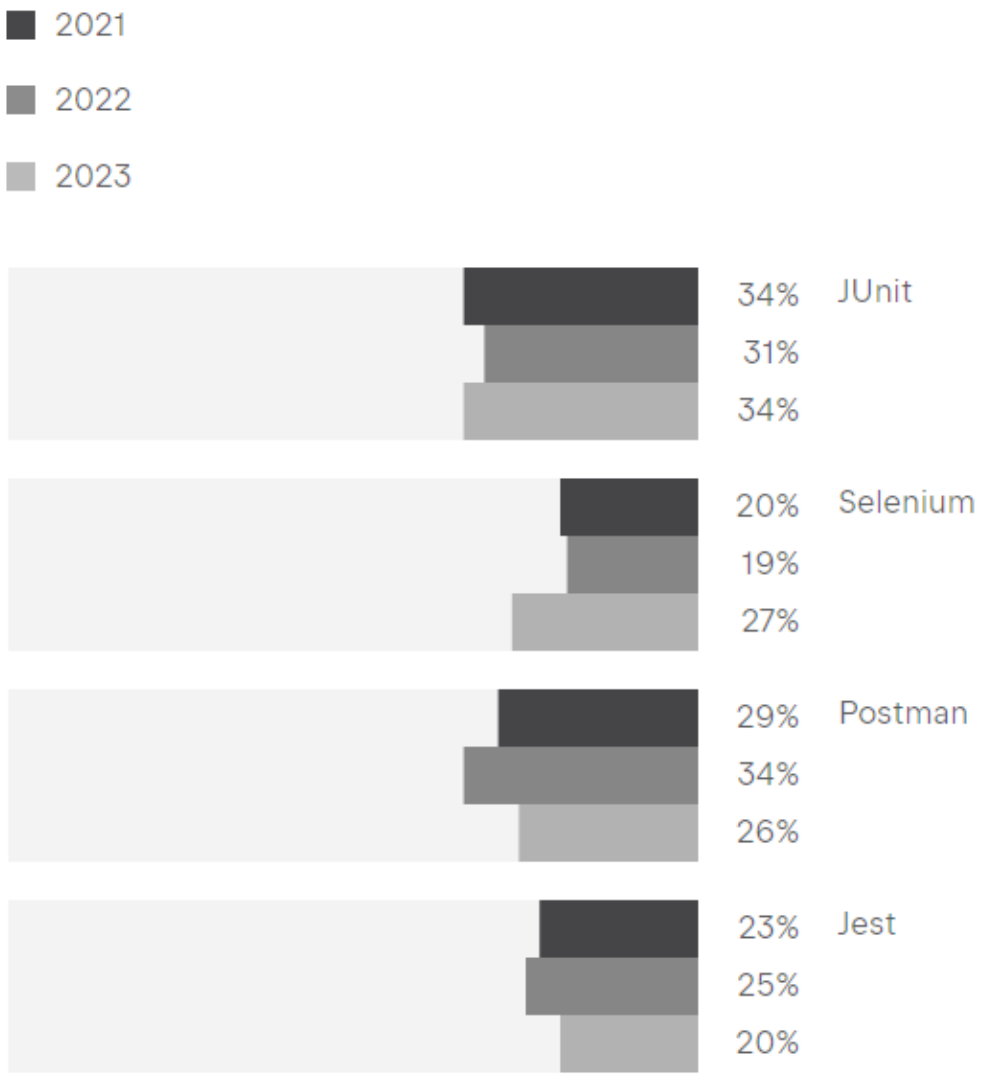
Исследования JetBrains 2023

https://www.jetbrains.com/lp/devecosystem-2023/testing/

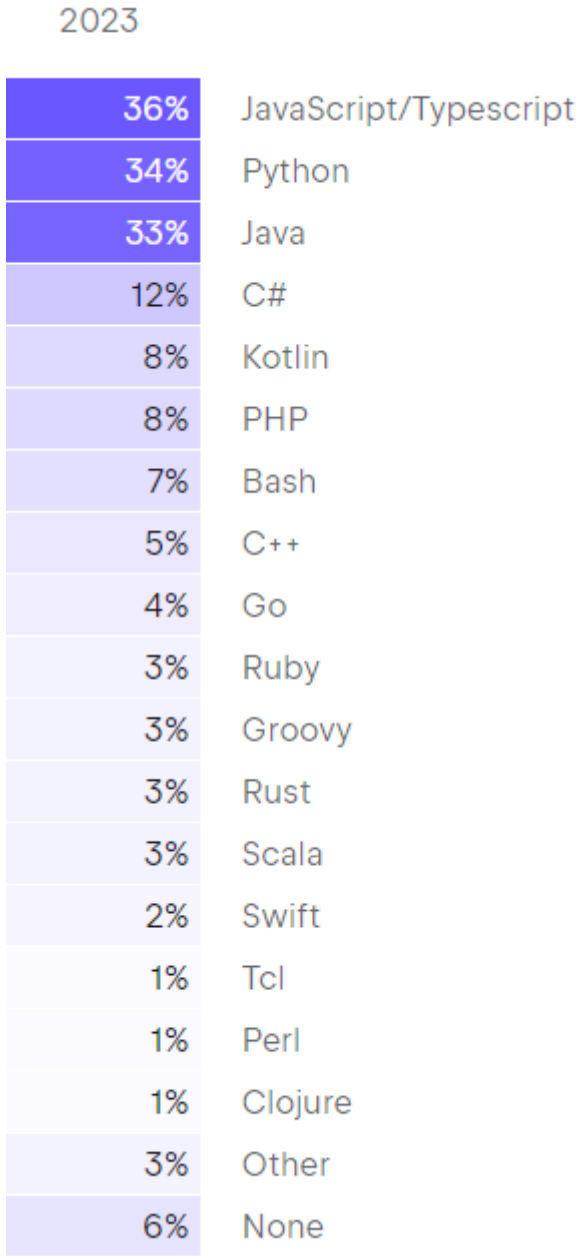
What types of tests do you have in your projects?



Which test frameworks, tools, and technologies do you use, if any?



Which programming languages do you use for test automation in your project?



Selenium

https://www.selenium.dev/

Selenium автоматизирует браузеры. Вот и все!

Что вы сделаете с этой силой, зависит только от вас.

В 2018 году **WebDriver стал рекомендацией World Wide Web Consortium (W3C)**. Что это значит? Основные поставщики браузеров (Mozilla, Google, Apple, Microsoft) поддерживают WebDriver и постоянно работают над улучшением браузеров и кода управления браузерами, что приводит к более единообразному поведению в разных браузерах, делая ваши скрипты автоматизации более стабильными.

Основными понятиями в Selenium Webdriver являются:

- ✓ **Webdriver** - самая важная сущность, ответственная за управление браузером. Основной ход скрипта строится именно вокруг экземпляра этой сущности.
- ✓ **Webelement** - вторая важная сущность, представляющая собой абстракцию над веб-элементом (кнопки, ссылки, поля ввода и др.). Webelement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текущего статуса.
- ✓ **By** - абстракция над локатором веб-элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

WebDriver: Basic concepts

Java

1 WEBDRIVER

2 WEBELEMENT

3 BY

```
public void simpleScenario() {
    WebDriver driver = new ChromeDriver();
    driver.get("http://www.google.com/");
    By searchBtnSelector = By.xpath("//input[@name='btnK']");
    WebElement searchButton = driver.findElement(searchBtnSelector);
    searchButton.click();
    driver.quit();
}
```

Working with browser

Looking for elements

Working with elements

Java

Python

Для поиска элементов доступно два метода:

1. Первый - найдёт только первый элемент, удовлетворяющий локатору:

```
WebElement element = driver.findElement(By.id("#element_id")));
```

```
elem = driver.find_element(By.ID, "element_name")
```

2. Второй - вернёт весь список элементов, удовлетворяющих запросу:

```
List elements = driver.findElements(By.name("elements_name"))
```

```
elems = driver.find_elements(By.ID, "element_name")
```

Типы локаторов

Java

Поскольку Webdriver - это инструмент для автоматизации веб приложений, то большая часть работы с ним это работа с веб элементами. **WebElements - ни что иное, как DOM объекты, находящиеся на веб странице.** А для того, чтобы осуществлять какие-то действия над DOM объектами / веб элементами необходимо их точным образом определить(найти).

```
WebElement element = driver.findElement(By.<Selector>);
```

Таким образом в Webdriver определяется нужный элемент.

By - класс, содержащий статические методы для идентификации элементов.

Java

Python

1. By.id

```
WebElement element = driver.findElement(By.id("element_id"));
```

```
elem = driver.find_element(By.ID, "element_name")
```

2. By.name

```
WebElement element = driver.findElement(By.name("element_name"));
```

```
elem = driver.find_element(By.NAME, "element_name")
```

3. By.className

```
WebElement element = driver.findElement(By.className("element_class"));
```

```
elem = driver.find_element(By.CLASS_NAME, "element_name")
```

4. By.TagName

```
List<WebElement> elements = driver.findElements(By.tagName("a"));
```

5. By.LinkText

```
WebElement element = driver.findElement(By.linkText("text"));
```

6. By.PartialLinkText

```
WebElement element = driver.findElements(By.partialLinkText("text"));
```

7. By.cssSelector

Java

Python

```
<div class='main'>
    <p>text</p>
    <p>Another text</p>
</div>
```

```
WebElement element=driver.FindElement(By.cssSelector("div.main"));
```

```
elem = driver.find_element(By.CSS_SELECTOR, "div.main")
```

8. By.XPath

```
<div class='main'>
    <p>text</p>
    <p>Another text</p>
</div>
```

```
WebElement element = driver.findElement(By.xpath("//div[@class='main']"));
```

```
elem = driver.find_element(By.XPATH, «//div[@class='main']")
```

Пример

```
<body>
  <header> … </header>
  <main role="main">
    <section class="jumbotron text-center"> … </section>
    <div class="album py-5 bg-light">
      <div class="container">
        <div class="row"> flex
          <div class="col-sm-4">
            <div class="card mb-4 box-shadow"> … </div> flex
          </div>
          <div class="col-sm-4">
            <div class="card mb-4 box-shadow"> flex
              <img class="card-img-top" data-src="holder.js/100px225?theme=thumb&bg=55595c&fg
              [100%x225]" src="images/serious_cat.jpg" data-holder-rendered="true"> == $0
              <div class="card-body"> … </div>
            </div>
          </div>
```

Селекторы, которые генерирует браузер по кнопке "скопировать css селектор" или расширения, зачастую строят полный путь, начиная от body — а это очень нестабильный селектор, писать такие в своем коде это плохая практика.

При малейшем изменении структуры страницы все ваши селекторы потеряют актуальность.

Сгенерированный	body > main > div > div > div > div:nth-child(2) > div > img
Написанный вручную	div.col-sm-4:nth-child(2) img

Взаимодействие с веб-элементами

Над элементом можно выполнить только 5 основных команд:

- click (applies to any element)
- send keys (only applies to text fields and content editable elements)
- clear (only applies to text fields and content editable elements)
- submit (only applies to form elements)
- select (see Select List Elements)

```
# Отметить checkbox "I'm the robot".
option1 = driver.find_element(By.CSS_SELECTOR, value: "[for='robotCheckbox']")
option1.click()
```

Действия с клавиатурой

https://www.selenium.dev/documentation/webdriver/actions_api/keyboard/

Функция send_keys() принимает различные ключи в качестве параметра. Поэтому необходимо импортировать ключи перед использованием этой функции.

```
1   from selenium import webdriver
2   from selenium.webdriver.common.keys import Keys
3   from selenium.webdriver.common.by import By
4
5   driver = webdriver.Chrome()
6   driver.get("http://www.python.org")
7   elem = driver.find_element(By.NAME, value: "q")
8   elem.clear()
9   elem.send_keys("pycon")
10  elem.send_keys(Keys.RETURN)
11  assert "No results found." not in driver.page_source
12  driver.close()
```

Ввод в текстовое поле

Нажатие клавиши Ввод

Действия с клавиатурой

С помощью клавиатуры можно выполнить только 2 действия: **нажатие клавиши и отпускание нажатой клавиши**. Помимо поддержки символов ASCII, каждая клавиша клавиатуры имеет представление, которое можно нажимать или отпускать в определенных последовательностях.

В Selenium можно выполнять все операции с клавиатурой. Класс **selenium.webdriver.common.keys** содержит различные методы, которые можно использовать для этой цели.

```
ActionChains(driver)\
    .key_down(Keys.SHIFT)\
    .send_keys("a")\
    .key_up(Keys.SHIFT)\
    .send_keys("b")\
    .perform()
```

ActionChains — это способ автоматизации низкоуровневых взаимодействий, таких как движения мыши, действия кнопок мыши, нажатия клавиш и взаимодействия с контекстным меню. Это полезно для выполнения более сложных действий, таких как наведение и перетаскивание.

Генерация действий пользователя.

Когда вы вызываете методы для действий на объекте ActionChains, действия сохраняются в очереди в объекте ActionChains. Когда вы вызываете perform(), события запускаются в том порядке, в котором они поставлены в очередь.

ActionChains можно использовать в цепочке:

```
ActionChains(driver).move_to_element(menu).click(hidden_submenu).perform()
```

Или действия можно выстроить в очередь одно за другим, а затем выполнить:

```
actions = ActionChains(driver)
actions.move_to_element(menu)
actions.click(hidden_submenu)
actions.perform()
```

https://www.selenium.dev/selenium/docs/api/py/webdriver/selenium.webdriver.common.action_chains.html

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Firefox(executable_path="C:\geckodriver.exe")
driver.get("url")

actions = ActionChains(driver)
actions.send_keys(value=username)
actions.send_keys(keys.TAB)
actions.send_keys(value=password)
actions.send_keys(keys.ENTER)
actions.perform()
driver.quit()
```

Действия с мышью

https://www.selenium.dev/documentation/webdriver/actions_api/mouse/

С помощью мыши можно выполнить только 3 действия: нажать кнопку, отпустить нажатую кнопку и переместить мышь.

```
clickable = driver.find_element(By.ID, "click")
ActionChains(driver)\
    .click(clickable)\
    .perform()
```

```
clickable = driver.find_element(By.ID, "clickable")
ActionChains(driver)\
    .click_and_hold(clickable)\
    .perform()
```

Действия с мышью

Щелчок правой кнопкой.

```
clickable = driver.find_element(By.ID, "clickable")
ActionChains(driver)\
    .context_click(clickable)\
     .perform()
```

Двойной щелчок

```
clickable = driver.find_element(By.ID, "clickable")
ActionChains(driver)\
     .double_click(clickable)\
     .perform()
```

Действия с мышью

Перейти к элементу.

```
hoverable = driver.find_element(By.ID, "hover")
ActionChains(driver)\
    .move_to_element(hoverable)\
    .perform()
```

Перетащить элемент

```
draggable = driver.find_element(By.ID, "draggable")
droppable = driver.find_element(By.ID, "droppable")
ActionChains(driver)\
    .drag_and_drop(draggable, droppable)\
    .perform()
```

Настройка ожиданий

Ожидания - непременный атрибут любых UI тестов для динамических приложений. Нужны они для синхронизации работы UI и тестового скрипта.

Скрипт выполняется намного быстрее реакции приложения на команды, поэтому часто в скриптах необходимо дожидаться определенного состояния приложения для дальнейшего с ним взаимодействия.

Самый простой пример - переход по ссылке и нажатие кнопки:

```
driver.get("http://google.com")
driver. find_element(By.ID("element_id")).click();
```

В данном случае необходимо дождаться пока не появится кнопка с id = element_id и только потом совершать действия над ней. Для этого и существуют ожидания.

Для ожиданий можно использовать библиотеку **time** в Python.

```python
from selenium import webdriver
from selenium.webdriver.common.by import By
import time

browser = webdriver.Chrome()
browser.get("http://suninjuly.github.io/wait1.html")

time.sleep(1)
button = browser.find_element(By.ID, "verify")
button.click()
message = browser.find_element(By.ID, "verify_message")

assert "successful" in message.text
```

Вывод: решение с time.sleep() плохое:

оно не масштабируемое и трудно поддерживаемое.

- А если другие элементы тоже будут появляться с задержкой?
- А если изменится время задержки?
- А еще на разных машинах с разной скоростью интернета элементы могут появляться через разные промежутки времени.

Selenium Waits (Implicit Waits)

Ожидания бывают:

Неявные ожидания - Implicit Waits - конфигурируют экземпляр WebDriver делать многократные попытки найти элемент (элементы) на странице в течение заданного периода времени, если элемент не найден сразу.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

browser = webdriver.Chrome()
# говорим WebDriver искать каждый элемент в течение 5 секунд
browser.implicitly_wait(5)

browser.get("http://suninjuly.github.io/wait1.html")

button = browser.find_element(By.ID, "verify")
button.click()
message = browser.find_element(By.ID, "verify_message")

assert "successful" in message.text
```

В данном примере наличие элемента будет проверяться каждые 500 мс. Как только элемент будет найден, мы сразу перейдем к следующему шагу в тесте.

Selenium Waits (Explicit Waits)

Методы find_element проверяют только то, что элемент появился на странице. В то же время элемент может иметь дополнительные свойства, которые могут быть важны для наших тестов. Рассмотрим пример с кнопкой, которая отправляет данные:

- ✓ Кнопка может быть неактивной, то есть её нельзя кликнуть;
- ✓ Кнопка может содержать текст, который меняется в зависимости от действий пользователя. Например, текст "Отправить" после нажатия поменяется на "Отправлено";
- ✓ Кнопка может быть перекрыта каким-то другим элементом или быть невидимой.

> ***Пример**: Если мы хотим в тесте кликнуть на кнопку, а она в этот момент неактивна, то WebDriver все равно проэмулирует действие нажатия на кнопку, но данные не будут отправлены. Чтобы тест был надежным, нужно не только найти кнопку на странице, но и дождаться, когда кнопка станет кликабельной.*

Для реализации подобных ожиданий в Selenium WebDriver существует понятие **явных ожиданий (Explicit Waits)**, которые позволяют задать специальное ожидание для конкретного элемента.

Задание явных ожиданий реализуется с помощью инструментов WebDriverWait и expected_conditions.

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium import webdriver

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/wait2.html")

# говорим Selenium проверять в течение 5 секунд, пока кнопка не станет кликабельной
button = WebDriverWait(browser, 5).until(
        EC.element_to_be_clickable((By.ID, "verify"))
    )
button.click()
message = browser.find_element(By.ID, "verify_message")

assert "successful" in message.text
```

```python
# говорим Selenium проверять в течение 5 секунд, пока кнопка не станет кликабельной
button = WebDriverWait(browser, 5).until(
        EC.element_to_be_clickable((By.ID, "verify"))
    )
button.click()
```

Правило ожидания

В модуле expected_conditions есть много других правил, которые позволяют реализовать необходимые ожидания:

- title_is
- title_contains
- visibility_of_element_located
- visibility_of
- presence_of_all_elements_located
- text_to_be_present_in_element
- element_to_be_clickable
- element_to_be_selected
- element_located_to_be_selected
- element_selection_state_to_be
- element_located_selection_state_to_be
- alert_is_present . . .

Описание каждого правила можно найти на сайте.

Exceptions

Если при поиске элемента произойдет ошибка,
то WebDriver выбросит одно из следующих исключений:

- ✓ **NoSuchElementException** - если элемент не был найден за отведенное время.
- ✓ **StaleElementReferenceException** - если элемент был найден в момент поиска, но при последующем обращении к элементу DOM изменился. Например, мы нашли элемент Кнопка и через какое-то время решили выполнить с ним уже известный нам метод click. Если кнопка за это время была скрыта скриптом, то метод применять уже бесполезно — **элемент "устарел" (stale)** и мы увидим исключение.
- ✓ **ElementNotVisibleException** - если элемент был найден в момент поиска, но сам элемент невидим (например, имеет нулевые размеры), и реальный пользователь не смог бы с ним взаимодействовать.

Знание причин появления исключений помогает отлаживать тесты и понимать, где находится баг в случае его возникновения.