Шпаргалка: Page Object

Можно писать код автотеста по особым правилам: так, чтобы с ним было удобно работать. Эти правила называют **паттернами проектирования**.

Page Object Model, или **POM** — это один из паттернов проектирования.

Он работает так: каждой странице приложения соответствует класс. Веб-элементы становятся атрибутами этого класса, а действия с ними — методами.

Класс, в который выносят локаторы элементов и действия с ними, называют **page object**.

Зачем использовать POM

Паттерн используют, чтобы упростить работу с кодом тестов. Он помогает отделить технические детали тест-кода от его логики. И поэтому:

- код становится более читаемым.
- редактировать код удобнее.
- описания локаторов можно переиспользовать.

Как создать page object

Чтобы это сделать, нужно:

1. Создать класс страницы авторизации.
2. Описать локаторы нужных элементов как атрибуты класса.
3. Добавить конструктор класса — метод `__init__`.
4. Описать действия с элементами страницы как методы класса.

Пример. Автотест должен заполнить поле Email и поле «Пароль» на странице авторизации. Page object страницы авторизации будет таким:

```
# импортировали webdriver и класс для поиска локаторов By
from selenium import webdriver
from selenium.webdriver.common.by import By

# инициализировали драйвер
```

```
driver = webdriver.Chrome()

# Шаг 1. Объявили класс страницы — page object
class LoginPageMesto:
    # Шаг 2. Описали локатор поля «Email»
    email_field = [By.ID, 'email']
    # Описали локатор поля «Пароль»
    password_field = [By.ID, 'password']

    # Шаг 3. Добавили конструктор класса
    def __init__(self, driver):
        self.driver = driver # Инициализировали его атрибуты

    # Шаг 4. Добавили действия с элементами как методы
    # метод заполняет поле «Email»
    def set_email(email):
        driver.find_element(*email_field).send_keys(email)

    # метод заполненяет поле «Пароль»
    def set_password(password):
        driver.find_element(*password_field).send_keys(password)
```

Метод `__init__` в page object обычно используют в начале кода, до описания методов. Первым аргументом в метод передают атрибут `self`, вторым — драйвер.

Локаторы в POM

📌 В page object описывают только элементы, которые участвуют в тестах.

Чтобы описать локатор как атрибут класса:

1. Объяви атрибут класса. Дай ему имя, которое укажет на описанный элемент.
2. Присвой переменной список или кортеж.
3. Первым значением списка укажи способ, которым программа будет искать локатор на странице. Вторым значением — сам локатор.

Последовательность значений в списке важна. Их нельзя менять местами.

Пример. Page object с тремя локаторами.

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

```
class LoginPageMesto:
    # локатор поля «Email»
    email_field = [By.ID, 'email']
    # локатор поля «Пароль»
    password_field = [By.ID, 'password']
    # локатор кнопки входа в приложение
    sign_in_button = [By.CLASS_NAME, 'auth-form__button']
```

В page object необязательно хранить целую страницу. В класс можно выделить несколько отдельных элементов. Например, шапку, футер и боковое меню, если они одинаковые на каждой странице. Так делают, чтобы собрать схожие локаторы в одном классе.

Методы в POM

В POM каждый элемент выносится в атрибут класса. Именно поэтому его можно использовать в коде несколько раз. Чтобы так получилось сделать, **нужно распаковывать** переменную перед использованием в методе.

Значения атрибута распаковывают с помощью оператора звёздочки — `*`.

Пример. Метод для клика кнопки «Войти».

```
    # локатор кнопки входа в приложение
    sign_in_button = [By.CLASS_NAME, 'auth-form__button']

    # метод кликает по кнопке «Войти»
    def click_sign_in_button(self):
        self.driver.find_element(*self.sign_in_button).click()
```

Что происходит в коде. Когда метод обращается к переменной `sign_in_button`, оператор берёт её значения и передаёт их на вход методу. Так метод `find_element()` понимает, что должен искать по имени класса, и ищет он локатор `sign_in_button`. Это и есть распаковка.

Виды методов

Все методы в POM делятся на два типа:

- действия,
- проверки.

К действиям относятся методы, которые имитируют действия пользователя на странице. Например, клик по кнопке и выбор элемента из списка.

Проверки — это методы, которые проверяют свойства элементов. Например, видимость элемента или соответствие отображаемого на нём текста ожидаемому.

Объединение методов: шаг

Внутри page object можно группировать методы и объединять их в шаги.

Шаг — это метод, который хранит последовательность действий или проверок. В шаг объединяют методы, которые вместе приводят к какому-то результату.

Как создать шаг

Чтобы объединить методы в шаг, нужно:

1. Создать новый метод. Дать ему имя, которое отразит результат шага.

2. Внутри метода-шага вызвать методы с действиями или проверками в нужном порядке.

Например, объединили в шаг `login` методы, которые нужны для авторизации в приложении:

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


class LoginPageMesto:
    # локатор поля «Email»
    email_field = [By.ID, 'email']
    # локатор поля «Пароль»
    password_field = [By.ID, 'password']
    # локатор кнопки входа в приложение
    sign_in_button = [By.CLASS_NAME, 'auth-form__button']

    def __init__(self, driver):
        self.driver = driver

    # метод заполняет поле «Email»
    def set_email(self, email):
        self.driver.find_element(*self.email_field).send_keys(email)

    # метод заполняет поля «Пароль»
    def set_password(self, password):
        self.driver.find_element(*self.password_field).send_keys(password)

    # метод кликает по кнопке «Войти»
```

```
    def click_sign_in_button(self):
        self.driver.find_element(*self.sign_in_button).click()

    # метод авторизации в приложении: объединяет ввод email, пароля и клик по кнопке
    # это и есть шаг
    def login(self, email, password):
        self.set_email(email)
        self.set_password(password)
        self.click_sign_in_button()
```

📎 **О чём стоит помнить:**

1. Лучше давать шагам осмысленные имена;
2. Когда объединяешь методы, важно соблюдать баланс. Объединять все методы в шаги не нужно;
3. Лучше объединяй в шаги методы, которые часто идут друг за другом. Так их можно будет использовать в разных тестах.

Пример автотеста с POM

Программа проверяет страницу авторизации Mesto: вводит почту и пароль, затем кликает по кнопке входа в приложение.

```
# импорт драйвера
from selenium import webdriver
# импорт класса страницы авторизации
from page_object import LoginPageMesto


class TestPraktikum:

    driver = None

    @classmethod
    def setup_class(cls):
        # создали драйвер для браузера Chrome
        cls.driver = webdriver.Chrome()

    def test_login(self):
        self.driver.get('https://qa-mesto.praktikum-services.ru/')

        # создать объект класса страницы авторизации
        login_page = LoginPageMesto(self.driver)
        # выполнить авторизацию
```

```
        login_page.login('email учётной записи', 'пароль учётной записи')

    @classmethod
    def teardown_class(cls):
        # закрыли браузер
        cls.driver.quit()
```

Метод `setup_class()`

В методе `setup_class()` инициализируется `driver` для работы с `webdriver.Chrome()`.

```
    @classmethod
    def setup_class(cls):
        # создали драйвер для браузера Chrome
        cls.driver = webdriver.Chrome()
```

Этот метод используется в классе один раз, перед всеми тестами. Он нужен, чтобы не инициализировать драйвер в каждом тесте.

После метода, в тестах к `driver` обращаются через `self.driver`. Например, вот так:

```
# объект класса страницы авторизации
login_page = LoginPageMesto(self.driver)
```

Перед объявлением метода стоит `@classmethod`. Это **декоратор**.

Декораторы расширяют возможности методов и функций, к которым их применяют. Конкретно `@classmethod` помогает отделить методы класса от обычных методов.

Поэтому метод `setup_class` — это метод класса. И его первый параметр `cls`, а не `self`.

Метод `teardown_class()`

Метод `teardown_class()` закрывает браузер и завершает работу `driver`. Это делается с помощью конструкции `cls.driver.quit()`.

```
    @classmethod
    def teardown_class(cls):
        # закрыли браузер
        cls.driver.quit()
```

Как и `setup_class()`, метод `teardown_class()` исполняется один раз. Только его используют после всех тестов.