«Talento Tech»

Automation Testing

Clase 9

Buenos Aires Ciudad

Clase N° 9: Page Object Model

Temario

- ¿Por qué necesitamos un patrón de diseño para nuestras pruebas?
- Concepto y beneficios del Page Object Model
- Anatomía de una página (clase)
- Estructura de carpetas recomendada
- Implementación paso a paso con Selenium + Pytest
- Definición de locators y métodos de acción
- Separación de lógica de prueba y lógica de página
- Reutilización de fixtures
- Mejores prácticas y trampas habituales
- Ejercicio guiado: refactorizar la pre-entrega usando POM

Objetivos de la clase

En esta clase exploraremos el patrón de diseño Page Object Model (POM) y entenderemos por qué es una estrategia clave para mejorar la robustez y el mantenimiento de pruebas automatizadas de interfaz gráfica. Aprenderás a diseñar clases de página limpias y autocontenidas, con localizadores bien definidos y métodos claros que encapsulan las interacciones con cada vista. A través de la refactorización de los tests para el sitio saucedemo.com, aplicaremos el modelo POM de forma que cualquier cambio en la interfaz se vea reflejado en un solo lugar del código. Finalmente, reforzaremos buenas prácticas de nomenclatura, documentación y organización de carpetas, esenciales para mantener proyectos escalables y colaborativos.

¿Qué es un patrón y por qué lo necesitamos?

Un **patrón de diseño** es una solución reusable a un problema recurrente en desarrollo de software. No es código "copiar-pegar", sino una receta que describe cómo organizar clases y responsabilidades para resolver un tipo de situación.

Patrones conocidos:

- **Singleton**: una sola instancia global (útil para configuraciones)
- **Observer**: un objeto notifica a muchos cuando cambia (eventos)
- **MVC**: separa la vista, la lógica y los datos en apps web
- **Factory**: crea objetos sin exponer su lógica de construcción

En testing UI el problema recurrente es **duplicar selectores y acciones** a lo largo de múltiples tests. Si el front cambia, nuestros casos fallan masivamente. Ahí entra el Page Object Model: un patrón específico para automatización que encapsula cada pantalla en una clase y expone métodos de alto nivel.

Meta-lección: aplicar un patrón no es lujo arquitectónico; es invertir tiempo hoy para ahorrar diez veces ese esfuerzo cuando el proyecto crezca.

Page Object Model: Concepto y beneficios

El **Page Object Model (POM)** propone representar cada página o vista de tu aplicación como un objeto (clase) que:

- Conoce sus elementos (locators)
- Sabe actuar sobre ellos mediante métodos significativos
- Oculta detalles de UI al resto del código

Beneficios detallados:

✅**Mantenibilidad extrema**: si cambian 20 botones "Add to cart" de `.btn_primary` a `.btn`, se modifica un solo archivo

✅**Legibilidad**: los nombres de los tests cuentan la historia de negocio: `login_page.ingresar_credenciales()`, `inventario_page.agregar_producto('Sauce Labs Bike')`

✅**Reutilización**: otros testers o suites (API, mobile) pueden usar las mismas clases

✅**Abstracción de esperas**: las esperas explícitas pueden vivir dentro de la página, manteniendo los tests limpios

✅**Escalabilidad**: añadir una página nueva = añadir una clase, sin tocar las existentes

Anatomía de una clase de página

Para empezar a usar Page Object Model (POM), primero necesitamos definir **qué aplicación web vamos a automatizar** y luego organizar nuestro proyecto de manera que sea escalable y mantenible.

¿Qué aplicación web usaremos?

A lo largo de este curso utilizaremos **SauceDemo** (https://www.saucedemo.com) como nuestra aplicación de prueba. Esta es una aplicación web demo específicamente diseñada para practicar automatización de pruebas, que simula un e-commerce con las siguientes características:

- **Página de login** con diferentes tipos de usuarios
- **Catálogo de productos** con funcionalidad de carrito
- **Flujo de compra** completo
- **Elementos estables** para automatización (IDs, clases CSS consistentes)

¿Por qué SauceDemo?

- Es gratuita y está siempre disponible
- Tiene diferentes escenarios (usuarios válidos, bloqueados, con problemas)
- Simula comportamientos reales de una aplicación de producción
- No requiere registración ni configuración previa

Estructura de carpetas recomendada

Ahora que sabemos qué vamos a testear, organizamos nuestro proyecto de esta manera:

```
proyecto/
├── pages/
│   ├── __init__.py
│   ├── login_page.py
│   ├── inventory_page.py
│   └── cart_page.py
├── tests/
│   ├── __init__.py
│   ├── test_login.py
│   ├── test_catalog.py
│   └── test_cart.py
├── conftest.py
└── README.md
```

¿Por qué esta estructura?

- **pages/**: Cada archivo representa una página específica de SauceDemo. Por ejemplo, `login_page.py` encapsula todo lo relacionado con el login de `https://www.saucedemo.com/`
- **tests/**: Los tests importan y usan las clases de `pages/` pero no conocen los selectores internos
- **conftest.py**: Centraliza la configuración del WebDriver para que todos los tests la compartan
- **Separación clara**: Si SauceDemo cambia su diseño, solo modificamos la clase correspondiente en `pages/`, no todos los tests

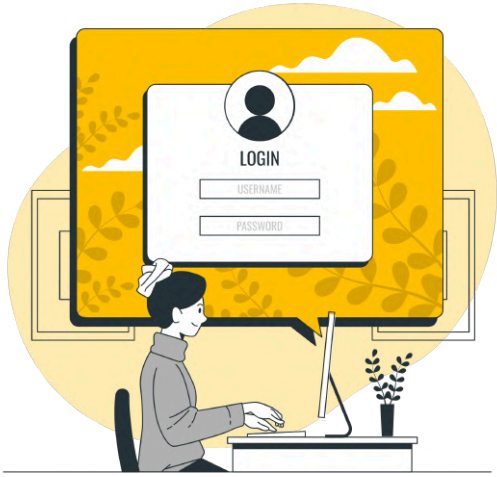
Flujo de trabajo con POM

1. **Análisis**: Identificamos las páginas principales de SauceDemo que queremos automatizar
2. **Creación**: Escribimos una clase Python por cada página (ej: `LoginPage` para la pantalla de login)
3. **Encapsulación**: Cada clase contiene los selectores y métodos específicos de esa página
4. **Uso**: Los tests importan estas clases y las usan sin conocer detalles internos de HTML

Con esta estructura, agregar una nueva pantalla de SauceDemo implica solo añadir `pages/nueva_page.py` y `tests/test_nueva_page.py`, sin tocar el código existente.

Implementación paso a paso

1) Crear el archivo pages/login_page.py



Para comenzar a aplicar el patrón Page Object Model, vamos a crear nuestra primera clase de página: la correspondiente a la pantalla de login. Esta clase vivirá en el archivo `pages/login_page.py`, siguiendo la convención de mantener una clase por archivo y reflejar en el nombre su propósito funcional.

Este archivo encapsulará toda la lógica asociada a la página de inicio de sesión: desde su URL, pasando por los elementos que la componen, hasta los métodos que permiten interactuar con ella. De esta forma, cualquier cambio futuro en la interfaz de login (por ejemplo, un nuevo ID para el botón de ingreso) podrá manejarse modificando una sola clase, sin afectar directamente los tests.

A continuación, se presenta el contenido básico de esta clase `LoginPage`, con comentarios que explican el rol de cada sección:

pages/login_page.py

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

class LoginPage:
    # 1. IDENTIDAD DE LA PÁGINA
    # URL absoluta de la página de login; si cambia, solo toca este valor.
    URL = "https://www.saucedemo.com/"

    # 2. LOCATORS (privados y centralizados)
    # Define aquí todos los selectores; si el front cambia,
    # solo modificas estas tres tuplas.
    _USER_INPUT   = (By.ID,   "user-name")
    _PASS_INPUT   = (By.ID,   "password")
    _LOGIN_BUTTON = (By.ID,   "login-button")
    _ERROR_MESSAGE = (By.CSS_SELECTOR, "[data-test='error']")

    def __init__(self, driver):
        """
        3. INYECCIÓN DE DEPENDENCIAS
        Recibe la instancia de WebDriver (Chrome, Firefox…)
        desde tu fixture en conftest.py.
        """
```

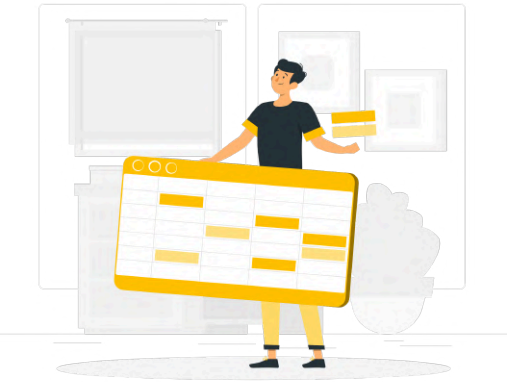
```
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    # 4. MÉTODOS DE NAVEGACIÓN Y ACCIÓN
    def abrir(self):
        """Carga la URL de login en el navegador."""
        self.driver.get(self.URL)
        return self
    def completar_usuario(self, usuario: str):
        """Escribe el nombre de usuario."""
        campo = self.wait.until(EC.visibility_of_element_located(self._USER_INPUT))
        campo.clear()               # Limpia antes de escribir
        campo.send_keys(usuario)
        return self
    def completar_clave(self, clave: str):
        """Escribe la contraseña."""
        campo = self.driver.find_element(*self._PASS_INPUT)
        campo.clear()
        campo.send_keys(clave)
        return self
    def hacer_clic_login(self):
        """Hace clic en el botón Login."""
        self.driver.find_element(*self._LOGIN_BUTTON).click()
        return self
    def login_completo(self, usuario, clave):
        """Método de conveniencia para hacer login completo."""
        self.completar_usuario(usuario)
        self.completar_clave(clave)
        self.hacer_clic_login()
        return self

    # 5. MÉTODOS DE VERIFICACIÓN (opcionales)
    def esta_error_visible(self) -> bool:
        """
        Comprueba si aparece un mensaje de error en la página
        tras un intento de login fallido.
        """
        try:
            self.wait.until(EC.visibility_of_element_located(self._ERROR_MESSAGE))
            return True
        except:
            return False

    def obtener_mensaje_error(self) -> str:
        """Obtiene el texto del mensaje de error."""
        if self.esta_error_visible():
            return self.driver.find_element(*self._ERROR_MESSAGE).text
        return ""
```

Esta clase permite que los tests se enfoquen únicamente en describir flujos de negocio (por ejemplo: “hacer login exitoso”), sin preocuparse por detalles técnicos como localizadores o tiempos de espera.

¿Por qué cada sección?

- **URL**: centraliza el punto de entrada
- **Locators**: en variables privadas (_MAYÚSCULAS), único lugar a editar si cambian atributos
- **Constructor**: inyecta driver desde el fixture, permitiendo usar cualquier navegador
- **Acciones de alto nivel**: describen "qué" hace el usuario, no "cómo" (olvídate de selectores en tus tests)
- **Helpers de aserción**: pequeños métodos para consultas, pero sin asserts dentro de la clase

2) InventoryPage: acciones después de loguearte



Una vez que el usuario accede exitosamente a la aplicación, es redirigido a la página de inventario, donde puede ver los productos disponibles y realizar acciones como agregarlos al carrito o cerrar sesión. Esta pantalla también debe encapsularse en una clase, que llamaremos `InventoryPage`, ubicada en el archivo `pages/inventory_page.py`.

Esta clase tiene como objetivo abstraer todas las interacciones posibles en esa vista: desde obtener información visible (como el título o los productos), hasta realizar acciones (como agregar productos o navegar al carrito). Siguiendo el enfoque del Page Object Model, centralizamos los selectores y exponemos métodos expresivos que describen las acciones del usuario sin acoplar al test con la lógica interna de la interfaz.

Aquí tienes su implementación con el código comentado para mejor comprensión.

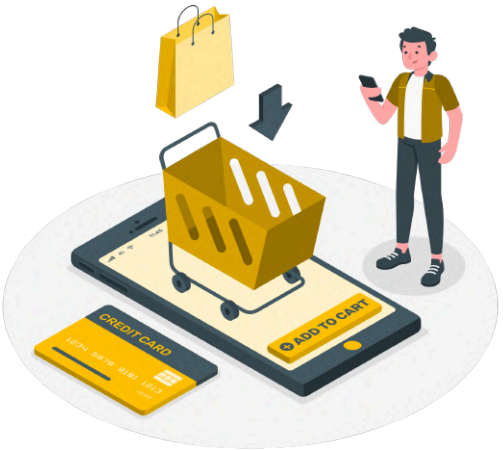
pages/inventory_page.py

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

class InventoryPage:
    _TITLE = (By.CLASS_NAME, "title")
    _PRODUCTS = (By.CLASS_NAME, "inventory_item")
```

```
    _ADD_BUTTONS = (By.CSS_SELECTOR, "button[data-test*='add-to-cart']")
    _CART_BADGE = (By.CLASS_NAME, "shopping_cart_badge")
    _CART_LINK = (By.CLASS_NAME, "shopping_cart_link")
    _MENU_BUTTON = (By.ID, "react-burger-menu-btn")
    _LOGOUT_LINK = (By.ID, "logout_sidebar_link")

    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    def obtener_titulo(self):
        """Obtiene el título de la página de inventario."""
        return self.driver.find_element(*self._TITLE).text

    def obtener_productos(self):
        """Obtiene la lista de productos disponibles."""
        return self.driver.find_elements(*self._PRODUCTS)

    def agregar_primer_producto(self):
        """Añade el primer producto disponible al carrito."""
        primer_boton = self.driver.find_elements(*self._ADD_BUTTONS)[0]
        primer_boton.click()
        return self

    def obtener_contador_carrito(self):
        """Obtiene el número de productos en el carrito."""
        try:
            badge = self.driver.find_element(*self._CART_BADGE)
            return int(badge.text)
        except:
            return 0

    def ir_al_carrito(self):
        """Navega a la página del carrito."""
        self.driver.find_element(*self._CART_LINK).click()
        # Importación lazy para evitar dependencias circulares
        from pages.cart_page import CartPage
        return CartPage(self.driver)

    def hacer_logout(self):
        """Cierra la sesión del usuario."""
        self.driver.find_element(*self._MENU_BUTTON).click()
        logout_link = self.wait.until(
            EC.element_to_be_clickable(self._LOGOUT_LINK)
        )
        logout_link.click()
        from pages.login_page import LoginPage
        return LoginPage(self.driver)
```

Este diseño no solo mejora la legibilidad del código de pruebas, sino que también permite una fácil extensión futura: si se agregan filtros, ordenamientos u otras funcionalidades en esta página, bastará con sumar métodos aquí sin tocar los tests existentes.

3) CartPage: verificación del carrito



La clase `CartPage` representa la pantalla del carrito de compras dentro del flujo de navegación de SauceDemo. Esta vista permite al usuario revisar qué productos ha agregado antes de continuar con el proceso de compra, por lo que cumple un rol clave en la validación del flujo de negocio.

Desde el enfoque del Page Object Model, encapsulamos en esta clase tanto la lógica de lectura (como obtener la lista de productos o sus nombres) como las posibles acciones disponibles (volver al inventario o iniciar el checkout). Esta separación permite que los tests solo describan lo que se espera validar ("ver que el carrito contiene el producto correcto"), sin necesidad de repetir selectores ni preocuparse por detalles de navegación.

Además, esta clase ayuda a reforzar la cohesión del código: cada método representa una acción o consulta específica y bien delimitada, lo cual facilita el mantenimiento y favorece la reutilización en múltiples escenarios de prueba.

pages/cart_page.py

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

class CartPage:
    _CART_ITEMS = (By.CLASS_NAME, "cart_item")
    _ITEM_NAMES = (By.CLASS_NAME, "inventory_item_name")
    _CHECKOUT_BUTTON = (By.ID, "checkout")
    _CONTINUE_SHOPPING = (By.ID, "continue-shopping")

    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)
        # Verificar que estamos en la página correcta
        self.wait.until(EC.url_contains("cart.html"))

    def obtener_productos_en_carrito(self):
        """Obtiene la lista de productos en el carrito."""
        return self.driver.find_elements(*self._CART_ITEMS)
```

```
    def obtener_nombres_productos(self):
        """Obtiene los nombres de todos los productos en el carrito."""
        elementos_nombre = self.driver.find_elements(*self._ITEM_NAMES)
        return [elemento.text for elemento in elementos_nombre]

    def continuar_comprando(self):
        """Regresa a la página de inventario."""
        self.driver.find_element(*self._CONTINUE_SHOPPING).click()
        from pages.inventory_page import InventoryPage
        return InventoryPage(self.driver)

    def proceder_checkout(self):
        """Inicia el proceso de checkout."""
        self.driver.find_element(*self._CHECKOUT_BUTTON).click()
        # Aquí podrías devolver CheckoutPage cuando la implementes
        return self
```

Fixture driver compartido

Para aplicar el Page Object Model de manera efectiva y evitar código duplicado en nuestros tests, es fundamental tener un punto centralizado desde donde se configure y provea el navegador. En este curso utilizamos una fixture de Pytest, llamada `driver`, que vive en el archivo `conftest.py`. Esta fixture se encarga de inicializar el WebDriver, aplicar configuraciones necesarias (como maximizar la ventana o activar modo sin cabeza para CI/CD) y cerrarlo al finalizar cada test.

Gracias a este enfoque, todos nuestros tests pueden acceder al navegador simplemente incluyendo `driver` como parámetro en sus funciones, sin preocuparse por detalles técnicos de su creación o destrucción. Además, al vivir en `conftest.py`, la fixture es accesible desde cualquier subdirectorio del proyecto, lo que fomenta la escalabilidad del framework.

conftest.py

```
import pytest
import time
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options

@pytest.fixture(scope="function")
def driver():
```

```
    """Fixture que proporciona un WebDriver configurado."""
    chrome_options = Options()
    # chrome_options.add_argument("--headless")  # Para CI/CD
    chrome_options.add_argument("--no-sandbox")
    chrome_options.add_argument("--disable-dev-shm-usage")

    service = Service()
    driver = webdriver.Chrome(service=service, options=chrome_options)
    driver.maximize_window()
    driver.implicitly_wait(5)

    yield driver

    time.sleep(1)  # Para ver el resultado final
    driver.quit()
```

Mejores prácticas y trampas

✅ Buenas prácticas:

- **Una página, una clase**: si tu app usa tabs con URL distinta, es otra página
- **Métodos claros**: `agregar_al_carrito()` mejor que `click_boton_rojo()`
- **No devuelvas elementos crudos**: expón datos (texto, contador) o acciones
- **Usa method chaining**: permite escribir `login_page.abrir().login_completo("user", "pass")`

❌ Trampas comunes:

- **Evita lógicas condicionales complejas** dentro de la página; eso es tarea del test
- **No mezcles asserts** dentro de métodos de página — mantenlos en tests
- **No hagas page objects demasiado genéricos** — cada página debe ser específica

🗨️**Storytelling**

Silvia anuncia:

“El cliente revisará la suite mañana. Queremos refactorizar la pre-entrega al patrón POM para demostrar escalabilidad.”

Matías aclara por qué es **clave para tu proyecto final**:

“Todo lo que hagas hoy quedará como base del framework definitivo que entregarás al final del curso. Si centralizas locators y acciones ahora, en las próximas clases solo tendrás que añadir nuevas páginas y tests sin reescribir código. Piensa en esto como colocar los cimientos de tu casa de automatización.”

Ejercicio práctico

Tarea paso a paso

1. Crea rama **feature/pom-refactor**.
2. Estructura carpetas `/pages` y mueve selectores allí.
3. Refactoriza los tests de login, inventario y carrito a la nueva API.
4. Ejecuta `pytest -m smoke --html=preentrega_pom.html` y adjunta el reporte al PR.

> **Conexión con la entrega:** cuando termines, tu repositorio cumplirá el criterio “Aplicar un enfoque básico de Page Object Model”

Checklist de entrega:

- `login_page.py`, `inventory_page.py` con locators.
- Fixture `driver()` en `conftest.py`.
- Mínimo **tres** tests (`test_login.py`, `test_catalogo.py`, `test_carrito.py`).
- README actualizado con nueva ejecución: `pytest -v`.

Este proceso de refactorización es un poco delicado, es por eso que desde Talento Tech creamos esta guía paso a paso para que tengas a mano para refactorizar tu aplicación:

Guía de Refactorización a Page Object Model - Paso a Paso

Próximos pasos

En la **Clase 10** entraremos a **Manejo de Datos de Prueba**: parametrización con `pytest.mark.parametrize`, lectura de CSV/JSON y generación de datos aleatorios con **Faker** para ampliar la cobertura sin duplicar tests.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad