DIGITALHOUSE

Patrones de diseño para automatización de la prueba

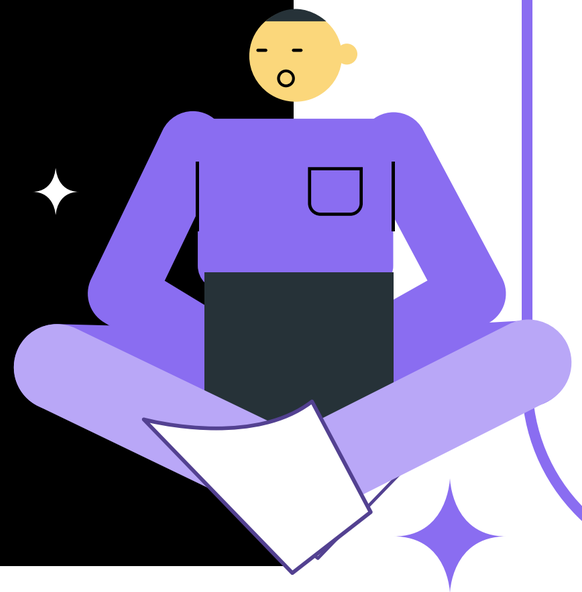
Certified Tech Developer
The Ultimate Degree

Índice

01

¿Qué es un patrón?

Una de las definiciones más destacada es la siguiente: “Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software.”



Patrones

Brindan una solución ya **probada y documentada** a problemas de desarrollo de software que están sujetos a contextos similares. Los siguientes elementos se deben tener en cuenta a la hora de elegir e implementar un patrón:

- Su nombre
- El problema: cuándo aplicar un patrón
- La solución: descripción abstracta del problema
- Las consecuencias: costos y beneficios

¿Dónde surgieron?

Antes de comenzar a detallar cada patrón deberìamos saber de dónde surgen los patrones y cuál es su utilidad.

El concepto de patrón de diseño lleva existiendo desde **finales de los 70**, pero su verdadera popularización surgió en los 90 con el **lanzamiento del libro de Design Pattern** de la Banda de los Cuatro (Gang of Four), nombre con el que se conoce a Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. En este libro explican 23 patrones de diseño, que desde entonces han sido considerados una referencia.

¿Por qué son útiles los patrones de diseño?

Te ahorran tiempo

Buscar siempre una nueva solución a los mismos problemas reduce tu eficacia como desarrollador. No hay que olvidar que **el desarrollo de software también es una ingeniería**, y que por tanto en muchas ocasiones habrá reglas comunes para solucionar problemas comunes.

Te ayudan a estar seguro de la validez de tu código

Los patrones de diseño son estructuras probadas por millones de desarrolladores a lo largo de muchos años, por lo que si eligen el patrón adecuado para modelar el problema adecuado, pueden estar seguros de que va a ser **una de las soluciones más válidas**.

Establecen un lenguaje común

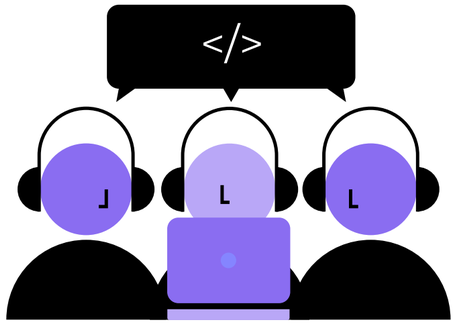
Modelar sus códigos mediante patrones les ayudará a explicar a otras personas, conozcan su código o no, a entender cómo han solucionado un problema. **Ayudan a todo el equipo a avanzar mucho más rápido**, con un código más fácil de entender y mucho más robusto.

02 Principios SOLID

Principios SOLID

Los patrones de diseño surgen debido a la aplicación de buenas prácticas, que están regidas por los principios SOLID de la programación Orientada a Objetos. Entonces: ¿qué son los principios SOLID?

“Son un conjunto de **principios aplicables a la Programación Orientada a Objetos** que, si los usas correctamente, **te ayudarán a escribir software de calidad** en cualquier lenguaje de programación orientada a objetos. Gracias a ellos, crearás código que será más fácil de leer, testear y mantener.”



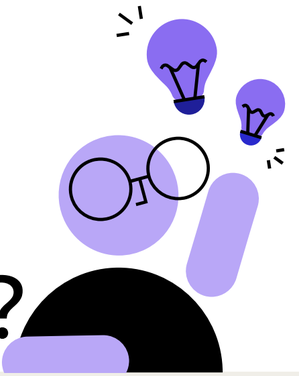
Principios SOLID

Los principios en los que se basa **SOLID** son los siguientes:

- ➔ Principio de Responsabilidad Única (**S**ingle Responsibility Principle)
- ➔ Principio Open/Closed (**O**pen/Closed Principle)
- ➔ Principio de Sustitución de Liskov (**L**iskov Substitution Principle)
- ➔ Principio de Segregación de Interfaces (**I**nterface Segregation Principle)
- ➔ Principio de Inversión de Dependencias (**D**ependency Inversion Principle)

Fueron publicados por primera vez por Robert C. Martin, también conocido como Uncle Bob, en su libro Agile Software Development: Principles, Patterns, and Practices.

¿Qué beneficios aportan los Principios SOLID?



Las **ventajas** de utilizar los **Principios SOLID** son innumerables, ya que nos aportan todas esas características que siempre queremos ver en un software de calidad.

- Software más flexible: mejoran la cohesión disminuyendo el acoplamiento: a grandes rasgos lo que buscamos de un buen código es que sus clases puedan trabajar de forma independiente y que el cambio de uno afecte lo menos posible al resto.
- Les van a hacer entender mucho mejor las arquitecturas.
- Simplifican la creación de tests.

Al final piensa que todo es como una cadena: si se aplican bien los principios, se organiza mejor el código. Esto permite definir una arquitectura que hará que los tests sean más sencillos.

Sabiendo de dónde surgen los patrones y cuáles son sus beneficios, ahora sí, vamos a detallar los patrones utilizados en el diseño de código correspondiente a pruebas automatizadas desde la perspectiva de los elementos que lo componen.

03 Patrón Page Object

Patrón Page Object

Este es el patrón de diseño más implementado para mejorar el mantenimiento de las pruebas y reducir el código duplicado. El concepto detrás de este patrón es el de **representar cada una de las pantallas que componen el sitio web o la aplicación que nos interesa probar**, como una serie de objetos que encapsulan las características y funcionalidades representadas en la página.

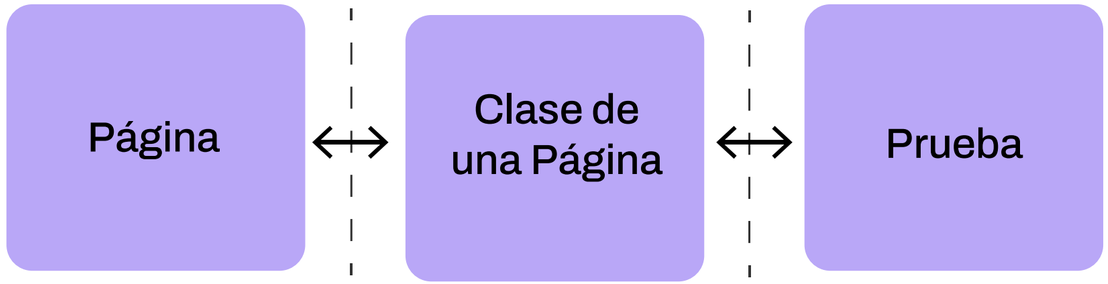
Al implementar este patrón, lo que estamos consiguiendo es crear una capa de abstracción entre el "¿qué podemos hacer/ver en la página?", y el "¿cómo se realiza esta acción?", simplificando enormemente la creación de nuestras pruebas y reutilizando el código con el que interactuamos con la página en concreto.

Patrón Page Object

Al implementar PageObject agrupamos aquellos métodos comunes a todas las páginas y luego vamos creando un PageObject para cada grupo de elementos que se relacionen de alguna manera —ya sea una barra de menú que se repite en varias páginas, una página con una funcionalidad especifica, un popup que aparece en varias páginas, etc.—. Lo mismo va a aplicar para los tests, ya que podemos centralizar en un TestPage object todo lo relacionado a la configuración de los tests y métodos de validación/generación de datos.

Por lo tanto, cualquier cambio que se produzca en la UI únicamente afectará al PageObject en cuestión, no a los test ya implementados.

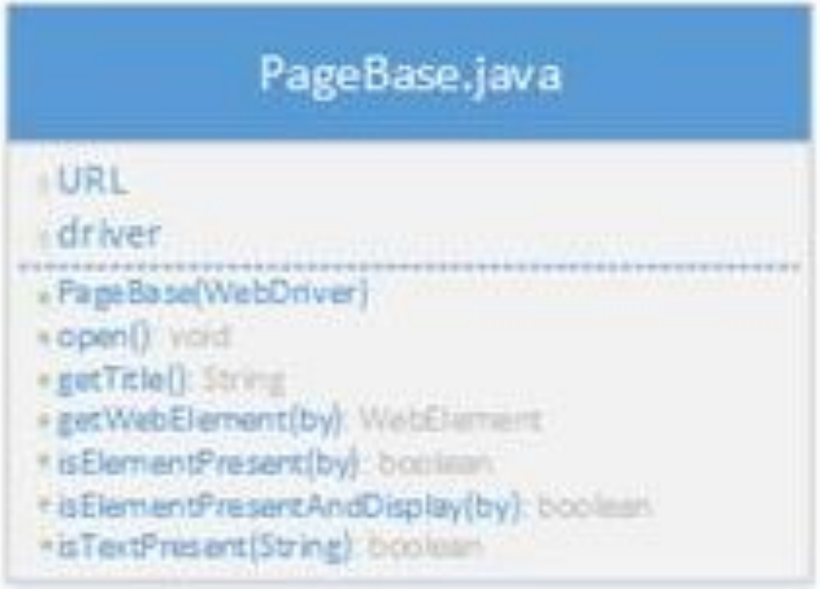
De esta manera, el PageObject se convierte en una API con la que fácilmente podemos encontrar y manipular los datos de la página.



Se puede dar el caso en el que una página corresponda a más de un PageObject, cuando algunas áreas de la página son lo suficientemente significativas. Por ejemplo, en una página web, podemos tener un PageObject para el header y otro para el body.

Patrón Page Object

Una opción válida al momento de implementación es crear, en primer lugar, una clase “básica”, que posteriormente será la que extenderán cada uno de los distintos PageObjects que vayamos implementando. Esta PageBase nos aporta la estructura básica y las propiedades generales que utilizaremos:



Patrón Page Object

Y una vez creada esta página, crear los PageObjects necesarios.

Un ejemplo de una página de “Login” podría ser:

03 Patrón Screenplay

Patrón Screenplay

Screenplay surge al escribir nuestras pruebas automatizadas basadas en los principios de ingeniería de software, SOLID, tal como son el Principio de Responsabilidad Única (SRP) y en el Principio Open/Closed (OCP).

El patrón Screenplay ha estado desde el 2007, algunos de los precursores son Antonio Marcano, Andy Palmer, Jan Molak y Jason Ferguson.

Screenplay **promueve los buenos hábitos de testing** y suites de pruebas bien organizadas las cuales son fáciles de seguir, fáciles de mantener y fáciles de extender; permitiendo que los equipos escriban pruebas automatizadas más robustas y entendibles.

Con este patrón las pruebas describen como un usuario puede interactuar con la aplicación para alcanzar una meta, es decir, está orientado al comportamiento del usuario, al enfoque de desarrollo encaminado por comportamiento o Behaviour Driven Development (BDD).

Patrón Screenplay

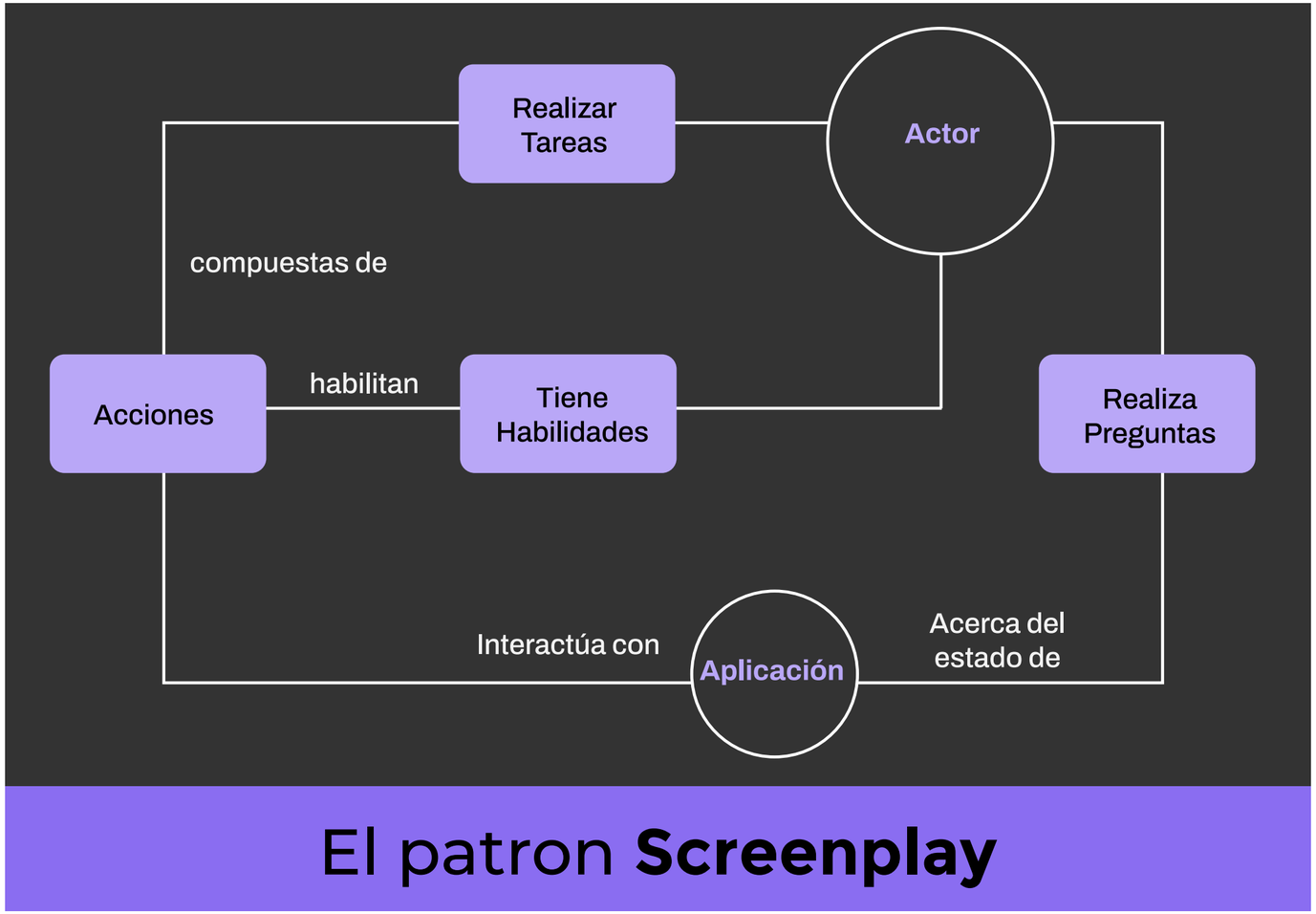
Un usuario interactuando con el sistema es llamado "Actor". **Los Actores son la parte central del patrón Screenplay.** Como Screenplay (Guión) los actores tienen uno o más "habilidades", como son la habilidad de navegar por la web o realizar una consulta un web service, el "Actor" también puede realizar "Tareas", como pueden ser agregar o remover objetos o cosas.

Para interactuar con la aplicaciones, como pueden ser introducir valores en los campos o dar clic en botones, los actores necesitan estas interacciones, que son denominadas "Acciones". Los "Actores" pueden hacer "Preguntas" acerca del estado del sistema, como por ejemplo, leer un valor de un campo en la pantalla o realizando una consulta a algún web service.

Patrón Screenplay

Es decir, el enfoque está basado en los siguientes elementos:

- Actores
- Habilidades
- Tareas
- Acciones
- Preguntas



04

Behavior Driven Development (BDD)

¿Qué es BDD (Behavior Driven Development)?

Este se refiere a desarrollo dirigido por comportamientos, BDD se trata de una estrategia de desarrollo, que se enfoca en prevenir defectos en lugar de encontrarlos en un ambiente controlado.

BDD es una **estrategia muy utilizada en las metodologías ágiles**, ya que en estas generalmente los requerimientos son entregados como User Stories (Historias de usuario). Lo que se plantea a través del desarrollo dirigido por comportamientos es la definición de un lenguaje común para el negocio y equipo solucionador.

Con BDD las pruebas de aceptación podrían ser escritas directamente en lenguaje Gherkin sin ningún problema, ya que este es un lenguaje común que puede escribir alguien sin conocimientos en programación.

Un ejemplo para el Login de una aplicación en lenguaje Gherkin:

Feature File:

```
1  Feature: login
2  Scenario: login with valid credentials
3    Given I am on the login page
4    When I enter a valid email
5    And I enter a valid password
6    And I press "Login"
7    Then I should be on the users home page
8    And I should see "Login successful"
```

BDD

El lenguaje **Gherkin** no es más que un texto con una estructura de 5 palabras claves con las que construimos sentencias y con estas describimos las funcionalidades. Este texto se guarda en un archivo con la extensión “.feature”, las palabras claves que normalmente se utilizan son:

- **Feature:** nombre de la funcionalidad que vamos a probar, el título de la prueba (HU).
- **Scenario:** por cada prueba que se ejecutara se especifica la funcionalidad a probar.
- **Given:** este sería el contexto del escenario de prueba o las precondiciones de los datos.
- **When:** se especifican el conjunto de las acciones que se van a ejecutar en el test.
- **Then:** aquí se especifica el resultado esperado del test y/o las validaciones que deseamos realizar.

BDD

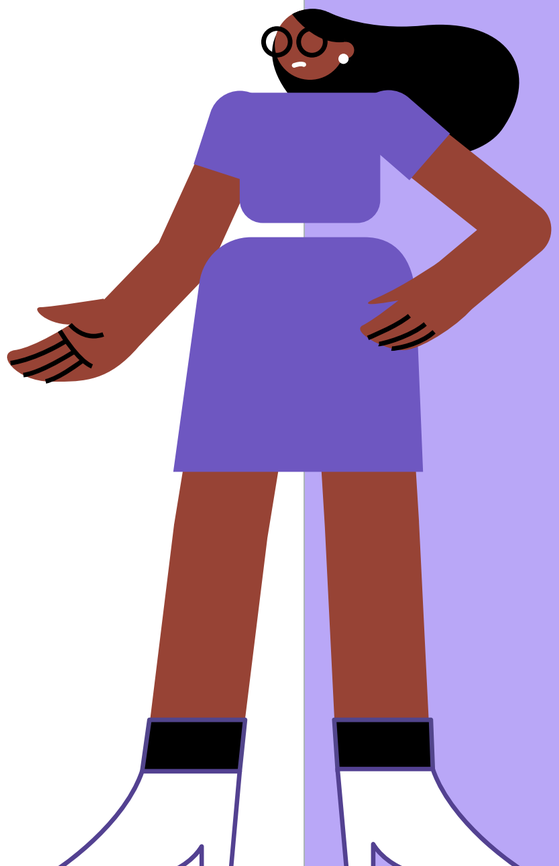
Las herramientas como **Cucumber**, **JBehave**, **Specflow** y muchas otras, permiten la implementación de una capa de conexión entre lenguaje Gherkin y la interfaz de usuario que se desea probar, pudiendo así utilizar eso como los pasos de una prueba automatizada.

Conclusiones

En general, cuando escribimos código debemos seguir buenas prácticas para evitar tener un código poco flexible, de difícil mantenimiento y poco extensible. Esto aplica tanto para el código de desarrollo de la aplicación como el código que es parte de pruebas automatizadas.

Los patrones son parte de éstas buenas prácticas que debemos aplicar para evitar esos problemas comunes que se generan cuando manejamos grandes volúmenes de código.

Recordemos que no conviene reinventar la rueda, mejor usemos soluciones ya probadas.



DIGITALHOUSE

¡Muchas gracias!