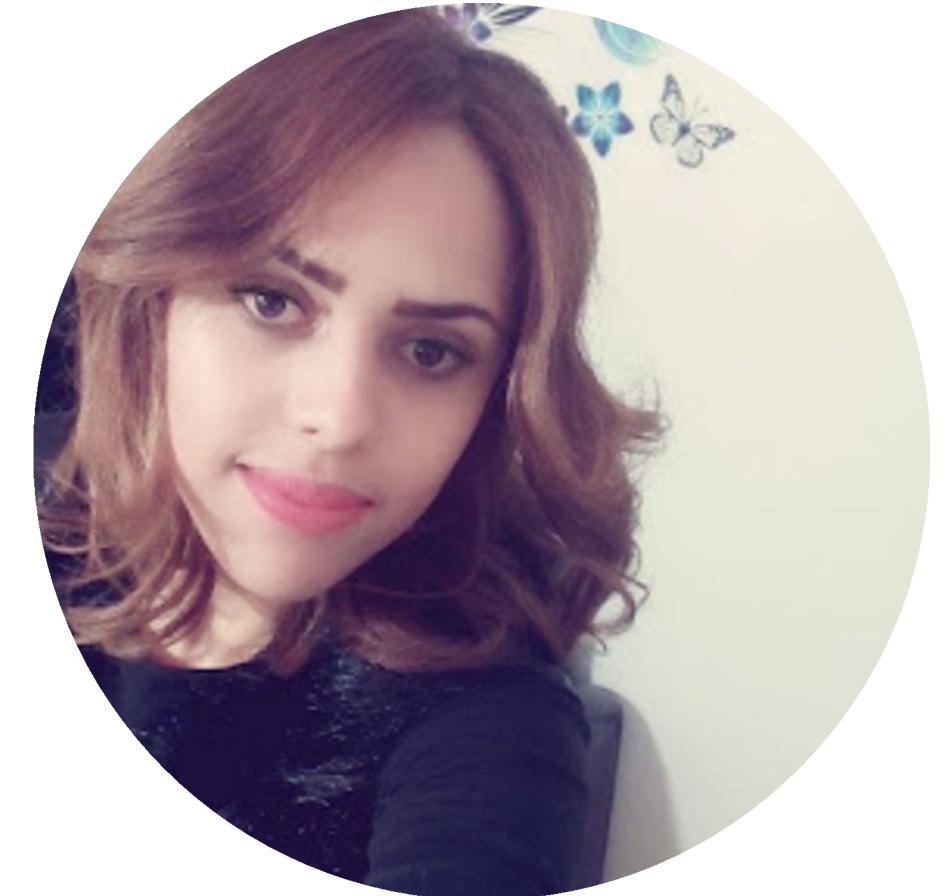


QA

with Marwa

A4Q Selenium Tester Foundation

01 août 2025



Présenté par
Marwa Bettaieb

Agenda

QA

With Marwa

01

Base de l'automatisation des tests

02

Technologies Internet pour l'automatisation
des tests d'applications Web

03

Utiliser Selenium WebDriver Termes

04

Préparer des scripts de test maintenables

Préparer des scripts de test maintenables

📌 Cas de test manuel vs automatisation : l'importance des fonctions wrapper

Un cas de test manuel consiste en une série d'instructions précises, incluant des actions à exécuter, des données d'entrée spécifiques et des résultats attendus. Ce type de test repose sur l'interprétation du testeur humain, capable de gérer intuitivement les imprévus et de s'adapter à des contextes complexes.

À l'inverse, l'automatisation des tests nécessite d'anticiper ces situations en intégrant de l'intelligence directement dans les scripts, ce qui augmente leur complexité et le risque d'échec si les scénarios ne sont pas suffisamment robustes.

Pour limiter ces risques, il est recommandé de concevoir une architecture d'automatisation intelligente et maintenable. L'une des meilleures pratiques est de créer des fonctions **wrapper**. Ces fonctions encapsulent des actions ou des validations fréquentes, tout en intégrant une logique additionnelle pour gérer les anomalies ou les conditions spécifiques.

Préparer des scripts de test maintenables

Avantages des fonctions wrapper :

- **Réutilisabilité** : centralisent le code commun (clic, saisie, vérification, attente, etc.).
- **Lisibilité** : simplifient les scripts de test en masquant les détails techniques.
- **Robustesse** : peuvent inclure des mécanismes de gestion d'erreurs et de waits conditionnels.
- **Maintenabilité** : toute modification (par ex. changement de sélecteur) s'effectue en un seul endroit.

Par exemple, une fonction **clickElement(WebElement element)** peut inclure :

- un test de visibilité,
- un try-catch pour intercepter les exceptions,
- et une stratégie de retry si nécessaire.

 **Conclusion** : Intégrer des fonctions wrapper dans vos frameworks permet de produire des scripts scalables, maintenables et plus intelligents, en comblant le fossé entre la flexibilité du test manuel et la rigueur de l'automatisation.

Préparer des scripts de test maintenables

1. Appel de la fonction

La fonction est invoquée pour vérifier l'état d'une case à cocher et la modifier si nécessaire (cocher/décocher selon l'état attendu).

2. Vérification de l'existence

La fonction commence par s'assurer que l'élément est présent dans le DOM.

- Si l'élément est absent, un timer démarre (avec des waits ou des retry loops).
- Si le délai maximal est atteint sans que l'élément n'apparaisse, le test échoue.

3. Vérification de la visibilité

Si l'élément existe, la fonction vérifie s'il est visible à l'écran.

- Si ce n'est pas le cas, un autre timer est lancé.
- À l'expiration de ce délai, le test échoue.

Préparer des scripts de test maintenables

4. Vérification de l'état activé (enabled)

La fonction s'assure que la case est activée et cliquable.

- Sinon, une attente conditionnelle est appliquée.
- Si elle reste désactivée au-delà du délai, le test échoue.

5. Action : cocher ou décocher

Lorsque la case est prête, la fonction compare l'état actuel de la case avec l'état attendu (checked ou unchecked).

- Si une action est nécessaire, elle est exécutée (clic).

6. Vérification de l'état final

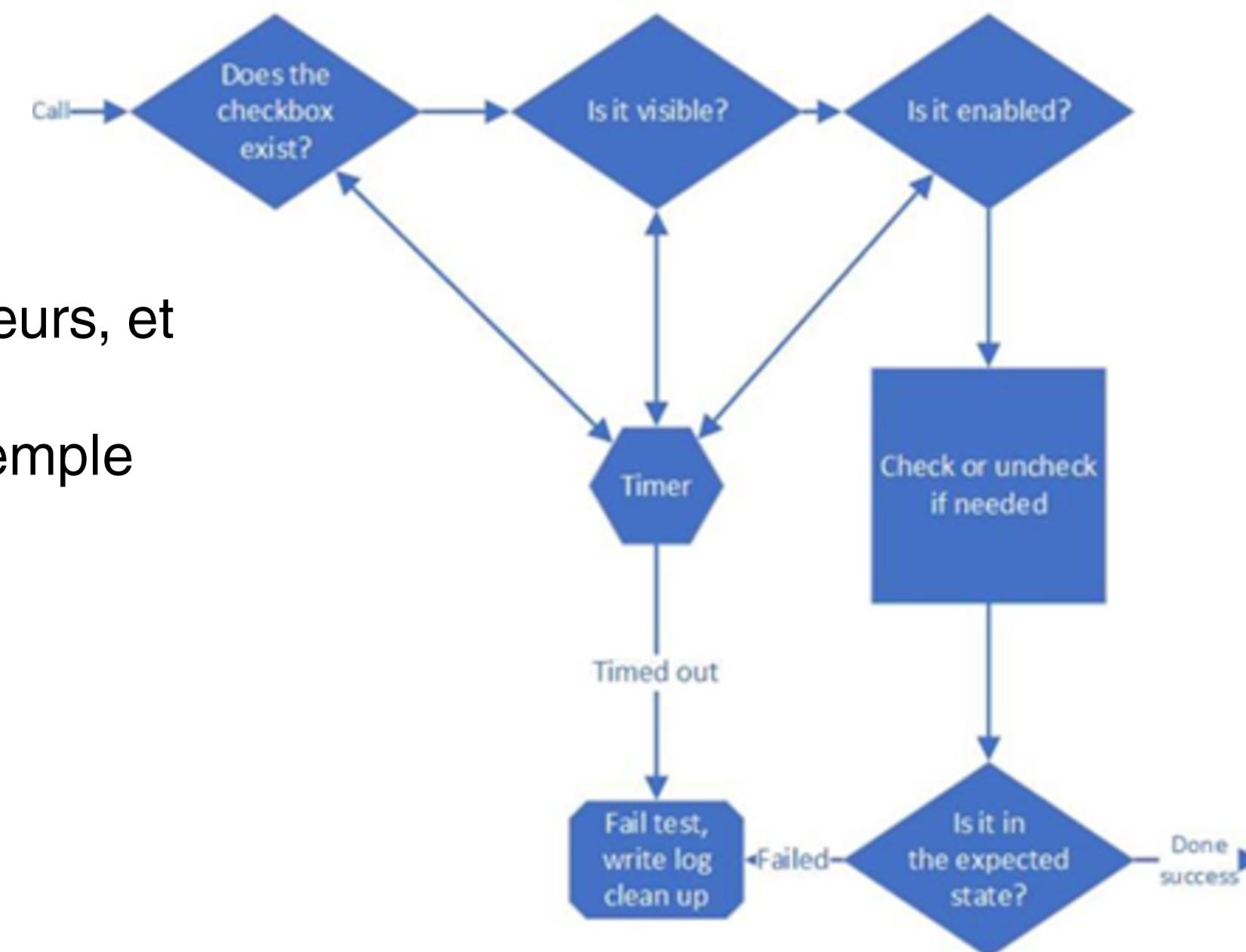
Une fois l'interaction effectuée, la fonction valide que la case se trouve dans l'état attendu.

- Si oui, l'opération est un succès.
- Sinon, un message d'erreur est enregistré, et le test échoue proprement.

Préparer des scripts de test maintenables

💡 Avantage

Ce type de fonction encapsule à la fois la logique métier, la robustesse face aux erreurs, et une meilleure gestion des attentes (waits intelligents), ce qui en fait un excellent exemple de wrapper pour des tests fiables et maintenables.



04-Préparer des scripts de test maintenables

QA

with Marwa

```
from selenium.common.exceptions import NoSuchElementException, ElementNotVisibleException
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
import logging

def set_checkbox_state(driver, checkbox_locator, expected_state, timeout=10):
    """
    Gère l'état d'une case à cocher en fonction du diagramme logique fourni.

    Arguments :
    - driver : WebDriver de Selenium.
    - checkbox_locator : Localisateur de la case à cocher (ex. (By.ID, 'checkbox_id')).
    - expected_state : Etat attendu de la case à cocher (True pour cochée, False pour décochée).
    - timeout : Temps maximum d'attente (en secondes).
    """

    try:
        # Vérifier si la case à cocher existe
        checkbox = WebDriverWait(driver, timeout).until(EC.presence_of_element_located(checkbox_locator))
    except NoSuchElementException:
        logging.error("La case à cocher n'existe pas.")
        return False # Échec du test

    try:
        # Vérifier si la case à cocher est visible
        WebDriverWait(driver, timeout).until(EC.visibility_of(checkbox))
    except ElementNotVisibleException:
        logging.error("La case à cocher n'est pas visible dans le délai imparti.")
        return False # Échec du test

    try:
        # Vérifier si la case à cocher est activée
        WebDriverWait(driver, timeout).until(EC.element_to_be_clickable(checkbox_locator))
    except ElementNotInteractableException:
        logging.error("La case à cocher n'est pas activée dans le délai imparti.")
        return False # Échec du test

    # Exemple d'utilisation
    # driver = webdriver.Chrome() # Assurez-vous que WebDriver est initialisé
    # set_checkbox_state(driver, (By.ID, 'checkbox_id'), expected_state=True)
```

```
try:
    # Vérifier si la case à cocher est activée
    WebDriverWait(driver, timeout).until(EC.element_to_be_clickable(checkbox_locator))
except ElementNotInteractableException:
    logging.error("La case à cocher n'est pas activée dans le délai imparti.")
    return False # Échec du test

# Vérifier l'état actuel de la case à cocher
is_checked = checkbox.is_selected()

# Cocher ou décocher si nécessaire
if is_checked != expected_state:
    checkbox.click()

# Vérifier l'état final de la case à cocher
if checkbox.is_selected() == expected_state:
    logging.info("La case à cocher est dans l'état attendu.")
    return True # Succès du test
else:
    logging.error("Échec : La case à cocher n'est pas dans l'état attendu.")
    return False # Échec du test
```

Mécanismes d'attente

Dans les tests manuels, le testeur utilise son jugement contextuel pour évaluer si une action prend un temps raisonnable. Par exemple, il peut tolérer qu'un petit fichier s'ouvre en quelques secondes, tandis qu'un fichier de 2 Go peut justifier un délai de 30 secondes.

En revanche, les scripts d'automatisation sont rigides : toute action doit respecter un délai défini à l'avance. Ainsi, un léger dépassement, même d'une milliseconde, peut provoquer l'échec du test. Pour gérer ces délais, **Selenium WebDriver** propose **trois mécanismes** d'attente.

1. Attentes implicites : elles demandent au WebDriver de rechercher un élément jusqu'à un délai défini (ex. : 10 secondes). Cette méthode est générale et reste active pour chaque recherche d'élément pendant toute la session de WebDriver.

Mécanismes d'attente

2. Attentes explicites : utilisées pour un élément ou une condition spécifique, elles offrent plus de flexibilité en permettant d'attendre jusqu'à ce que certaines conditions spécifiques soient remplies (ex., un élément devient cliquable ou visible)

3. Utilisation de time.sleep() :

`time.sleep(5)` : force une pause fixe de 5 secondes, sans tenir compte de l'état du DOM. Il est surtout utilisé pour déboguer ou introduire des pauses fixes dans le script. Ce type d'attente peut entraîner un ralentissement du script si l'élément est déjà disponible avant l'expiration des 5 secondes

Attente Implicite

```
from selenium import webdriver
driver = webdriver.Chrome()
driver.implicitly_wait(10) # attend jusqu'à 10s pour tout élément
checkbox = driver.find_element("id", "checkbox")
checkbox.click()
```

Attente Explicite

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
driver = webdriver.Chrome()
wait = WebDriverWait(driver, 10) # délai max = 10s
checkbox = wait.until(EC.element_to_be_clickable((By.ID, "checkbox")))
checkbox.click()
```

Mécanismes d'attente

Pause Fixe (sleep)

```
import time
from selenium import webdriver
driver = webdriver.Chrome()
time.sleep(5) # Pause forcée de 5 secondes (même si l'élément est prêt)
checkbox = driver.find_element("id", "checkbox")
checkbox.click()
```

Page Objects

Plus tôt dans le chapitre, il est recommandé de simplifier les scripts d'automatisation en déplaçant la complexité vers l'architecture ou le framework d'automatisation. L'objectif est de rendre les scripts plus lisibles et maintenables. Le concept du **Page Object Pattern** est introduit pour illustrer cette approche. Ce modèle consiste à créer des objets représentant les différentes pages de l'application, ce qui permet de centraliser la logique d'interaction avec l'interface utilisateur dans des classes spécifiques, plutôt que de disperser cette logique directement dans les scripts de test. Ainsi, on améliore la réutilisabilité et la maintenance du code d'automatisation. Le **Page Object Pattern** désigne donc l'application systématique du modèle **Page Object** dans l'architecture des tests d'automatisation. Ces deux termes sont souvent utilisés de manière interchangeable, car ils désignent essentiellement la même approche structurée pour gérer les interactions avec l'interface utilisateur dans les tests.

Page Objects

Les 6 avantages clés du Page Object:

- 1. Réutilisabilité du code :** Cela permet de créer du code réutilisable qui peut être partagé par plusieurs scripts de test.
- 2. Réduction du code dupliqué :** En centralisant la logique d'interaction avec l'interface, on évite de répéter cette logique dans chaque script de test.
- 3. Réduction des coûts et efforts de maintenance :** Comme toute l'interaction avec une page est encapsulée dans un objet, les modifications sont limitées à cet objet, ce qui simplifie la maintenance.
- 4. Encapsulation des opérations sur l'interface :** Cela permet d'encapsuler toutes les opérations sur l'interface graphique du SUT en une seule couche.

Page Objects

5. Séparation claire des responsabilités : L'architecture distingue clairement la partie métier des tests (ce que l'utilisateur final doit pouvoir faire « quoi ») et la partie technique (les interactions avec l'interface graphique « comment »).

6. Facilité d'adaptation aux changements : En cas de modifications de l'interface utilisateur (comme un changement de structure d'une page), il suffit de mettre à jour le Page Object correspondant, et tous les tests qui l'utilisent seront automatiquement ajustés

Page Objects

L'idée de diviser l'architecture d'automatisation des tests en **couches** est essentielle pour garantir la **maintenabilité** et la **lisibilité** du code sur le long terme. Une des couches clés mentionnées par l'**ISTQB** dans le syllabus de niveau avancé est la **couche d'abstraction des tests**. Cette couche fait le lien entre la **logique métier** (les actions à tester) et les besoins techniques spécifiques au pilotage de l'interface utilisateur du SUT (System Under Test)

Les **Page Objects** font partie intégrante de cette couche d'abstraction, car ils permettent de séparer la logique des tests des détails techniques de l'interface utilisateur. En d'autres termes, les **Page Objects abstraient** l'interface graphique en cachant les détails sur la manière dont les éléments sont localisés et manipulés, ce qui simplifie le script de test et le rend plus lisible et maintenable. Cela permet aux tests de se concentrer sur les scénarios métiers sans se soucier des spécificités de l'interface.

Page Objects

Ce code est fonctionnel mais a plusieurs problèmes :

- Il mélange la logique des tests avec l'interface utilisateur (UI).
- Il rend le code difficile à maintenir et à réutiliser

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time

def test_login():
    driver = webdriver.Chrome()

    # Étape 1 : Ouvrir la page de connexion
    driver.get("http://exemple.com/login")

    # Étape 2 : Trouver les éléments et interagir directement
    username_field = driver.find_element(By.ID, "username")
    password_field = driver.find_element(By.ID, "password")
    login_button = driver.find_element(By.ID, "login_button")

    # Étape 3 : Remplir le formulaire de connexion
    username_field.send_keys("user1")
    password_field.send_keys("pass123")
    login_button.click()

    # Attendre que la page se charge
    time.sleep(2) # Ce n'est pas une bonne pratique d'utiliser des pauses statiques

    # Étape 4 : Vérifier la présence du mot "Welcome" dans la source de la page
    assert "Welcome" in driver.page_source

    # Fermer le navigateur
    driver.quit()

# Appel de la fonction de test
test_login()
```

Page Objects

```
# login_page.py (Page Object)
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

class LoginPage:
    def __init__(self, driver):
        self.driver = driver
        self.username_field = driver.find_element(By.ID, "username")
        self.password_field = driver.find_element(By.ID, "password")
        self.login_button = driver.find_element(By.ID, "login_button")

    def login(self, username, password):
        self.username_field.send_keys(username)
        self.password_field.send_keys(password)
        self.login_button.click()

    def is_welcome_message_displayed(self):
        return "Welcome" in self.driver.page_source
```

```
# test_login.py (Test utilisant Page Object)
from selenium import webdriver
from login_page import LoginPage
import time

def test_login():
    # Initialiser le navigateur
    driver = webdriver.Chrome()

    # Ouvrir la page de Login
    driver.get("http://exemple.com/login")

    # Créer un objet LoginPage
    login_page = LoginPage(driver)

    # Effectuer la connexion
    login_page.login("user1", "pass123")

    # Attendre que la page se charge (meilleure approche : utiliser WebDriverWait)
    time.sleep(2)

    # Vérifier la présence du message "Welcome"
    assert login_page.is_welcome_message_displayed()

    # Fermer le navigateur
    driver.quit()

# Exécuter le test
#-----
```

Page Objects

Structure du code avec Page Object :

- **Page Object** : Représente une page web avec ses éléments et ses actions.
- **Test** : Contient les assertions, mais utilise le Page Object pour interagir avec l'interface.

Lorsque vous découpez votre architecture d'automatisation en couches et que vous concevez les Page Objects, vous devez respecter plusieurs règles générales :

- Les Page Objects ne doivent pas contenir d'assertions sur la logique métier ni de points de vérification.
- Toutes les assertions et vérifications techniques concernant l'interface graphique (par exemple, vérifier si une page a fini de se charger) doivent être réalisées dans les Page Objects.
- Toutes les attentes doivent être encapsulées dans les Page Objects.
- Seul le Page Object doit contenir les appels aux fonctions Selenium.
- Un Page Object n'a pas besoin de couvrir toute la page ou le formulaire. Il peut contrôler une section ou une autre partie spécifique de celle-ci.

Tests dirigés par mots-clé (Keyword Driven Testing)

Les tests dirigés par mots-clés, également connus sous le nom de **Keyword Driven Testing** (KDT), sont une technique de test automatisé où les tests sont définis en utilisant des mots-clés ou des instructions spécifiques qui décrivent les actions à effectuer et les vérifications à effectuer dans le système sous test (SUT). Ces mots-clés sont généralement regroupés dans des fichiers ou des feuilles de calcul, et un moteur d'exécution de tests lit ces mots-clés pour exécuter les tests.

Le concept de tâche abstraite dans le contexte du Keyword Driven Testing (KDT) permet de définir des actions utilisateur dans des termes abstraits et stables, indépendamment de la plateforme ou de l'interface utilisée. Par exemple, l'action "ouvrir un fichier" dans un traitement de texte est une tâche abstraite, peu importe qu'il s'agisse de DOS, Windows, MacOS ou Unix. L'important est que le "quoi" (ouvrir un fichier) reste constant, tandis que le "comment" (les étapes spécifiques pour y parvenir) varie selon l'implémentation de l'application.

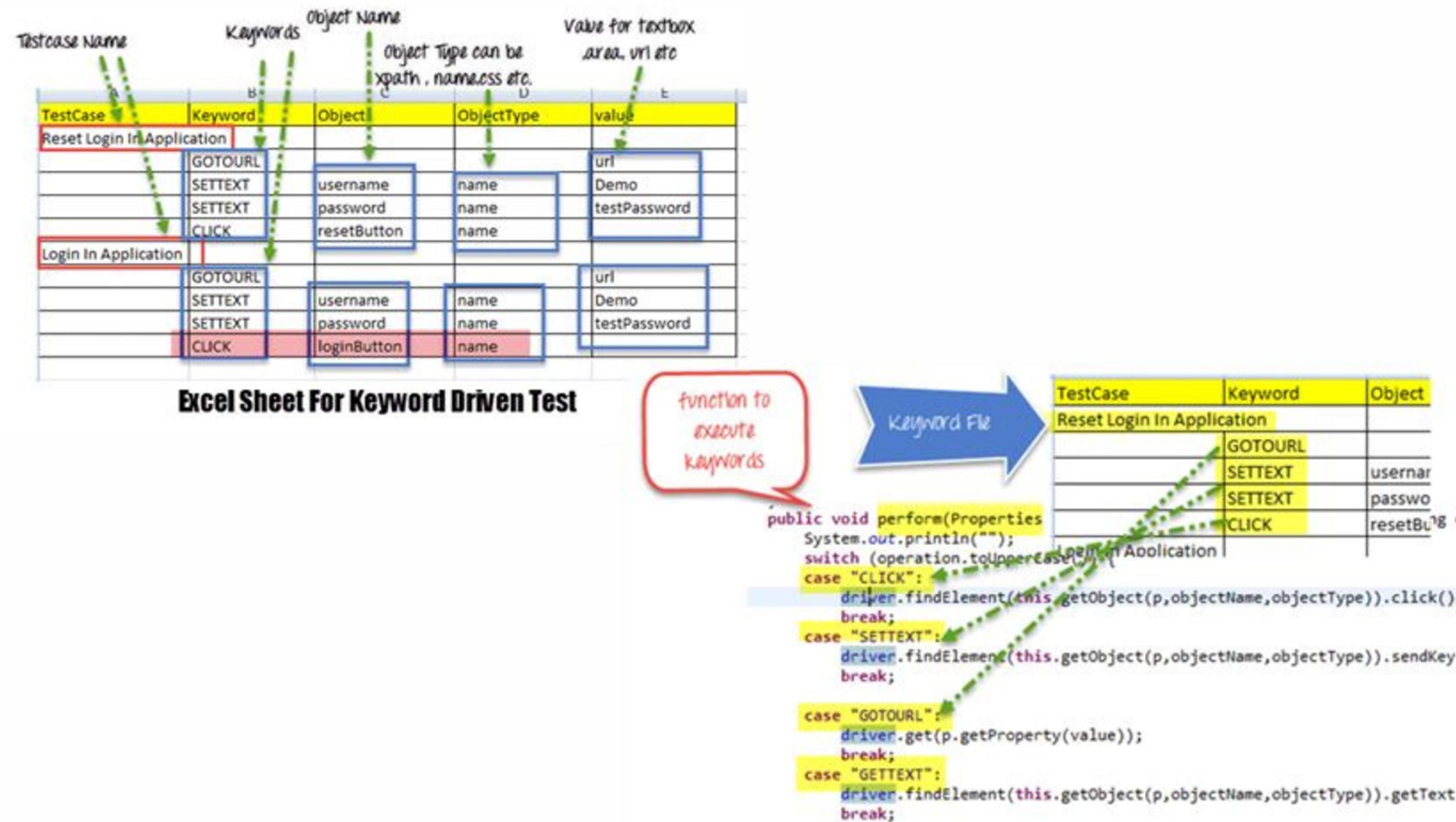
Tests dirigés par mots-clé (Keyword Driven Testing)

Dans le Keyword Driven Testing, chaque action ou tâche à tester est identifiée par un mot-clé, qui représente une tâche fonctionnelle (par exemple, ouvrir un fichier, sauvegarder un fichier, etc.). Ces tâches restent généralement les mêmes à travers les différentes versions d'une application, ce qui permet de créer des scripts de test réutilisables et maintenables. Les tests manuels bénéficient de cette abstraction car les concepts fondamentaux des tâches ne changent pas souvent, même si l'interface de l'application peut évoluer.

04-Préparer des scripts de test maintenables

QA

with Marwa



Tests dirigés par mots-clé (Keyword Driven Testing)

Une bonne pratique pour des tests maintenables (manuels ou automatisés) est d'assurer une séparation claire des couches dans l'automatisation des tests :

- la couche de définition de test contient des scénarios à un niveau abstrait, en se concentrant sur les actions métier, comme "se connecter à un système", "effectuer un paiement" ou "trouver un produit".
- Ces actions ne contiennent pas de détails sur la façon de les exécuter (le 'comment'), qui sont plutôt gérés par la couche d'adaptation.
- Cette dernière fait le lien entre les actions abstraites et l'interface du SUT (Système Sous Test), cachant les détails spécifiques de l'implémentation dans des mots-clés ou des objets tels que les Page Objects. Enfin, la couche d'exécution est responsable de l'exécution effective des tests

Tests dirigés par mots-clé (Keyword Driven Testing)

- Cette approche présente plusieurs avantages :
- La conception du cas de test est dissociée de l'implémentation du SUT.
- Une distinction claire est faite entre les couches d'exécution des tests, d'abstraction des tests et de définition des tests.
- Une répartition presque parfaite du travail :
- ✓ Les analystes de test conçoivent des scénarios de test et rédigent des scripts à l'aide de mots-clés, de données et de résultats attendus.
- ✓ Les analystes techniques de test (c'est-à-dire les automatiques de tests) implémentent les mots-clés et le framework d'exécution nécessaires à l'exécution des tests.
- La réutilisation de mots-clés dans différents cas de test.
- Une meilleure lisibilité des cas de test (ils ressemblent à des cas de test manuels).
- Moins de redondance.
- Une réduction des coûts et des efforts de maintenance.

Tests dirigés par mots-clé (Keyword Driven Testing)

- Si l'automatisation ne fonctionne pas, un testeur manuel peut exécuter le test manuellement directement à partir du script automatisé.
- Parce que les tests par mots-clés sont abstraits, les scripts utilisant des mots-clés peuvent être écrits bien avant que le SUT soit disponible pour le test (comme pour les tests manuels).
- Le point précédent indique que l'automatisation peut être prête plus tôt et utilisée pour les tests fonctionnels et pas seulement pour les tests de régression. Un petit nombre d'automaticien de tests peuvent travailler avec un nombre non limité d'analystes de tests, ce qui facilite l'extension de l'automatisation.
- Grâce à l'abstraction des tests, différents outils peuvent être utilisés de manière interchangeable pour l'exécution des tests

Tests dirigés par mots-clé (Keyword Driven Testing)

- Dans l'implémentation du Keyword Driven Testing (KDT), la structure des mots-clés peut devenir complexe, car ils peuvent varier en fonction de leur niveau d'abstraction. Certains mots-clés sont de bas niveau, représentant des actions concrètes sur l'interface utilisateur (par exemple, "Cliquer sur le bouton >>Cancel<<"), tandis que d'autres sont de haut niveau, représentant des scénarios plus abstraits qui regroupent plusieurs actions (par exemple, "Ajouter un produit X à la facture avec le prix Y et la quantité Z").

Organisation des mots-clés dans l'architecture des tests :

- **Mots-clés de bas niveau :**
 1. Implémentés dans la couche d'adaptation des tests, souvent sous forme de Page Objects, qui réalisent les actions concrètes sur le SUT.
 2. Ces mots-clés peuvent être utilisés dans des mots-clés de niveau supérieur dans la bibliothèque de la couche d'exécution des tests.

Tests dirigés par mots-clé (Keyword Driven Testing)

- **Mots-clés de haut niveau :**

1. Implémentés dans la bibliothèque des mots d'action et représentent des actions ou scénarios abstraits.
2. Utilisés comme étapes de test dans les procédures de test dans la couche d'exécution des tests.

Mise en œuvre du KDT :

Le KDT peut être mis en œuvre de deux manières :

- **Approche descendante (Top-down) :**

1. C'est la méthode "classique", telle que décrite par **Dorothy Graham**. Elle commence **par concevoir des cas de test manuels**, puis les étapes de ces tests sont **abstraites** pour devenir des mots-clés de haut niveau. Ces mots-clés peuvent être décomposés en mots-clés de bas niveau ou implémentés à l'aide d'outils ou de langages de programmation.

2. Cette méthode est plus efficace lorsque la **bibliothèque de tests** contient déjà de nombreuses fonctions ou méthodes qui réalisent des actions sur le SUT. Elle est particulièrement utile lorsque des tests manuels existants sont utilisés comme base pour l'automatisation.

Tests dirigés par mots-clé (Keyword Driven Testing)

- **Approche ascendante (Bottom-up) :**

1. Dans cette approche, les tests sont d'abord enregistrés **automatiquement** à l'aide d'un outil (par exemple, **Selenium IDE**). Ensuite, ces tests sont **restructurés** et organisés dans une architecture appropriée KDT.
2. Cette méthode permet de générer rapidement des scripts de test qui peuvent être exécutés sur le SUT, ce qui est particulièrement utile dans des situations où des tests rapides doivent être créés pour un produit en développement.

Avantages des deux approches :

- Approche descendante : Plus structurée et adaptée lorsqu'un cadre d'automatisation existe déjà, elle permet une intégration plus fluide des tests manuels existants dans l'automatisation.
- Approche ascendante : Plus rapide pour générer des scripts de test, idéale pour les environnements où une rapide rétroaction est nécessaire, mais elle peut nécessiter plus de réorganisation et de maintenance à long terme.

QA

with marwa



Fin du Chapitre 4
Merci pour votre attention !

bettaiebmarwa@gmail.com