

Testeur Selenium Certifié

Syllabus Niveau Fondation

Version 2018 FR



© A4Q Copyright 2018 - Copyright notice

Tous les contenus de ce document, en particulier les textes et les graphiques, sont protégés par le droit d'auteur. L'utilisation et l'exploitation de ce matériel relèvent exclusivement de la responsabilité de l'A4Q. En particulier, la copie ou la duplication de ce document, mais aussi de parties de ce document est interdite. L'A4Q pourra procéder à des poursuites civiles et pénales en cas d'infraction.

## Historique des révisions

Version	Date	Remarques
V 1.0F	20 septembre 2018	Version 2018 en français
V 1.0	5 August 2018	Version 2018 en anglais

## Table des matières

0 Introduction .....	5
0.1 Objet du présent Syllabus.....	5
0.2 Objectifs d'apprentissage et niveaux cognitifs de connaissances .....	5
0.3 L'examen de certification Testeur Selenium au niveau fondation.....	5
0.4 Accréditation.....	6
0.5 Niveau de détail.....	6
0.6 Organisation du syllabus .....	6
0.7 Compétences visées .....	7
0.8 Acronymes .....	7
Chapitre 1 - Bases de l'automatisation des tests.....	8
1.1 Aperçu de l'automatisation des tests .....	8
1.2 Les tests manuels par rapport aux tests automatisés .....	11
1.3 Facteurs de succès .....	14
1.4 Risques et avantages de Selenium WebDriver.....	15
1.5 Selenium WebDriver dans l'architecture d'automatisation des tests.....	16
1.6 Métriques pour l'automatisation .....	18
1.7 La boîte à outils Selenium .....	20
Chapitre 2 - Technologies Internet pour l'automatisation des tests d'applications Web.....	22
2.1 Comprendre HTML et XML.....	22
2.1.1 Comprendre HTML .....	22
2.1.2 Comprendre XML .....	30
2.2 XPath et recherche dans les documents HTML.....	32
2.3 Localisateur CSS.....	35
Chapitre 3 - Utiliser Selenium WebDriver .....	38
3.1 Mécanismes de logs et de reporting .....	39
3.2 Naviguer dans différentes URLs .....	43
3.2.1 Démarrer une session d'automatisation des tests .....	43
3.2.2 Navigation et rafraîchissement des pages.....	44
3.2.3 Fermeture du navigateur .....	45
3.3 Changer le contexte de la fenêtre.....	46
3.4 Capturer des captures d'écran de pages Web .....	48
3.5 Localiser les éléments de l'interface graphique.....	50
3.5.1 Introduction.....	50

3.5.2	HTML Methods .....	52
3.5.3	Méthodes XPath .....	54
3.5.4	Méthodes de sélection CSS .....	56
3.5.5	Localisation via des conditions prédéfinies .....	56
3.6	Obtenir l'état des éléments de l'interface graphique.....	57
3.7	Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver .....	58
3.7.1	Introduction.....	58
3.7.2	Manipulation des champs de texte.....	59
3.7.3	Cliquez sur des éléments web .....	60
3.7.4	Manipulation des cases à cocher.....	60
3.7.5	Manipulation des menus déroulants.....	61
3.7.6	Travailler avec les boîtes de dialogue modal.....	62
3.8	Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver .....	65
Chapitre 4	- Préparer des scripts de test maintenables .....	66
4.1	Maintenabilité des scripts de test.....	66
4.2	Mécanismes d'attente.....	72
4.3	Page Objects .....	75
4.4	Tests dirigés par mots-clé (Keyword Driven Testing) .....	78
Annexe	- Glossaire des termes Selenium .....	82

# 0 Introduction

## 0.1 Objet du présent Syllabus

Ce syllabus présente les compétences visées, les objectifs d'apprentissage et les concepts qui sous-tendent la formation et la certification Testeur Selenium au niveau fondation.

## 0.2 Objectifs d'apprentissage et niveaux cognitifs de connaissances

Les objectifs d'apprentissage permettent l'acquisition des compétences visées et sont utilisés pour créer les examens de la certification Testeur Selenium au niveau fondation.

De manière générale, tous les contenus de ce syllabus peuvent faire l'objet de questions d'examen au niveau K1, à l'exception de l'introduction et des annexes. C'est-à-dire qu'on peut demander au candidat de reconnaître ou de se souvenir d'un mot-clé ou d'un concept mentionné dans l'un des six chapitres du présent document. Les niveaux de connaissance des objectifs d'apprentissage sont fournis au début de chaque chapitre et classés de la façon suivante :

- K1 : se souvenir
- K2 : comprendre
- K3 : appliquer
- K4 : analyser

De plus amples détails et des exemples d'objectifs d'apprentissage sont donnés en Annexe B.

Tous les termes listés sous la rubrique "termes" après les titres de chapitre doivent être retenus (K1), même s'ils ne sont pas explicitement mentionnés dans les objectifs d'apprentissage.

## 0.3 L'examen de certification Testeur Selenium au niveau fondation

L'examen pour l'obtention du certificat Testeur Selenium au niveau fondation sera basé sur ce syllabus et sur le cours de formation correspondant accrédité par A4Q. Les réponses aux questions d'examen peuvent requérir l'utilisation d'informations contenues dans plus d'une section de ce syllabus. Toutes les sections de ce syllabus peuvent donner lieu à des questions d'examen à l'exception de cette introduction et des annexes. Des normes, livres et autres syllabus de l'ISTQB® sont indiqués comme références, mais leur contenu ne fait pas

l'objet de questions d'examen, au-delà de ce qui est résumé dans ce syllabus lui-même comme les normes, livres et autres syllabus de l'ISTQB®.

Le format de l'examen est un questionnaire à choix multiples. L'examen est structuré en 40 questions. Pour réussir cet examen, il faut obtenir au moins 65% de réponses justes (c'est-à-dire au moins 26 réponses justes). Le temps alloué pour passer l'examen est de 60 minutes. Si la langue maternelle du candidat n'est pas la langue de l'examen, le candidat peut bénéficier d'un délai supplémentaire de 25 % (15 minutes).

Les examens ne peuvent être passés qu'après avoir suivi la formation A4Q Testeur Selenium au niveau fondation, puisque l'évaluation par l'instructeur de la compétence du candidat dans les exercices fait partie de l'obtention de la certification.

## 0.4 Accréditation

La formation de l'A4Q Testeur Selenium au niveau fondation est le seul cours de formation accrédité.

## 0.5 Niveau de détail

Le niveau de détail dans ce syllabus permet un enseignement et des examens compatibles internationalement. Pour atteindre cet objectif, le syllabus contient :

- Des objectifs généraux d'instruction, décrivant les intentions du niveau fondation
- Une liste de termes dont les étudiants doivent se souvenir
- Des objectifs d'apprentissage pour chaque domaine de connaissance, décrivant les résultats cognitifs d'enseignements à acquérir
- Une description des concepts clé, incluant des références à des sources comme des normes ou de la littérature reconnue

Le contenu du syllabus n'est pas une description de l'ensemble du domaine de connaissance de l'automatisation des tests logiciels avec Selenium ; il reflète le niveau de détail devant être couvert par les formations du niveau fondation. Il se concentre sur les concepts et les techniques de test qui peuvent s'appliquer à tous les projets logiciels, y compris les projets Agiles. Ce syllabus ne contient pas d'objectifs d'apprentissage spécifiques liés à un cycle de développement logiciel ou à une méthode particulière, mais il discute de la façon dont ces concepts peuvent s'appliquer à divers cycles de développement de logiciels.

## 0.6 Organisation du syllabus

Ce syllabus comprend quatre chapitres sur lesquels porte l'examen. Le titre principal de chaque chapitre spécifie la durée prévue pour traiter ce chapitre, cette durée n'est pas détaillée au niveau des sections du chapitre. Pour la formation de l'A4Q Testeur Selenium au niveau fondation, le programme exige un minimum de 16,75 heures d'enseignement, réparties entre les quatre chapitres suivants :

Chapitre 1 : Bases de l'automatisation des tests - 105 minutes

Chapitre 2 : Technologies Internet pour l'automatisation des tests d'applications Web - 195 minutes

Chapitre 3 : Utilisation de Selenium WebDriver - 495 minutes

Chapitre 4 : Préparer des scénarios de test maintenables - 225 minutes

## 0.7 Compétences visées

SF-BO-1 Appliquer correctement les principes de l'automatisation des tests pour construire une solution d'automatisation des tests maintenable.

SF-BO-2 Etre capable de choisir et d'implémenter les bons outils d'automatisation des tests.

SF-BO-3 Pouvoir mettre en œuvre des scripts Selenium WebDriver qui exécutent des tests fonctionnels d'applications Web.

SF-BO-4 Pouvoir mettre en œuvre des scripts maintenables.

## 0.8 Acronymes

Pour chaque acronyme en anglais, nous donnons la traduction en français lorsque celle-ci est employée.

AKA : Also known as / Aussi connu sous le nom de

API : Application Programming Interface / Interface de programmation d'application

AuT : Automaticien de Tests

CERN : Conseil Européen pour la Recherche Nucléaire

CI : Continuous Integration / Intégration continue

CSS : Cascading Style Sheets / Feuilles de style en cascade

DOM : Document Object Model / Modèle Objet de Document

GUI : Graphical User Interface / IHM : Interface Homme Machine

HTTP : Protocole de Transfert HyperTexte

ISTQB : International Software Testing Qualification Board

KDT : Keyword Driven Testing / Tests dirigés par mots-clés

REST : Representational State Transfer

ROI : Return of Investment / Retour sur Investissement

SDLC : Software Development Lifecycle / Cycle de vie du développement logiciel

SOAP : Simple Object Access Protocol

SUT : System Under Test / Système sous test

TAA : Test Automation Architecture / Architecture d'automatisation des tests

TAE : Test Automation Engineer / Ingénieur en automatisation de tests

TAS : Test Automation Solution / Solution d'automatisation des tests

TCP : Transmission Control Protocol

UI : User Interface / Interface utilisateur

W3C : World Wide Web Consortium

# Chapitre 1 - Bases de l'automatisation des tests

## Termes

architecture, capture/replay, comparateur, tests exploratoires, injection de fautes, framework, hook, paradoxe du pesticide, dette technique, testabilité, harnais de test, oracle de test, testware

## Objectifs d'apprentissage pour les bases de l'automatisation des tests

- STF-1.1 (K2) Expliquer les objectifs, les avantages, les inconvénients et les limites de l'automatisation des tests
- STF-1.2 (K2) Comprendre la relation entre les tests manuels et les tests automatisés
- STF-1.3 (K2) Identifier les facteurs de réussite technique d'un projet d'automatisation des tests
- STF-1.4 (K2) Comprendre les risques et les avantages de l'utilisation de Selenium WebDriver
- STF-1.5 (K2) Expliquer la place de Selenium WebDriver dans une AAT
- STF-1.6 (K2) Expliquer la raison et le but de la collecte de métriques de l'automatisation
- STF-1.7 (K2) Comprendre et comparer les objectifs de l'utilisation des outils Selenium (WebDriver, Selenium Server, Selenium Grid)

### 1.1 Aperçu de l'automatisation des tests

L'automatisation des tests recouvre différents aspects. Nous limiterons notre vision de l'automatisation dans ce syllabus comme étant l'exécution automatique de tests fonctionnels, conçus au moins d'une certaine manière pour simuler un être humain exécutant des tests manuels. Il existe de nombreuses définitions différentes (voir le syllabus ISTQB de niveau avancé Automaticien de Tests - AuT) ; Cette vision de l'automatisation est celle qui est la plus adaptée à ce syllabus.

Alors que l'exécution des tests est largement automatisée, l'analyse des tests, la conception des tests et leur implémentation sont généralement effectuées manuellement. La création et le déploiement des données utilisées lors des tests peuvent être partiellement automatisés mais se font souvent manuellement. L'évaluation de la réussite ou de l'échec d'un test peut faire partie de l'automatisation (via un comparateur intégré à l'automatisation), mais pas toujours.

L'automatisation nécessite la conception, la création et la maintenance de différents niveaux de testware, y compris l'environnement dans lequel les tests seront exécutés, les outils utilisés, les bibliothèques de code qui fournissent les fonctionnalités, les scripts de test et les harnais de test, ainsi que les structures de log et de reporting pour évaluer les résultats des

tests. Selon les outils utilisés, la surveillance et le contrôle de l'exécution des tests peuvent être une combinaison de processus manuels et automatisés.

Il peut y avoir de différents objectifs poursuivis avec l'automatisation des tests fonctionnels, tels que :

- Améliorer l'efficacité des tests en réduisant le coût de chaque test.
- Tester plus et d'autres aspects que ceux que nous pourrions tester manuellement.
- Réduire le temps nécessaire à l'exécution des tests.
- Pouvoir pousser plus de tests plus tôt dans le cycle de vie du développement logiciel pour trouver et supprimer les défauts du code plus tôt (ce qui est appelé "shift left" pour le test).
- Augmenter la fréquence d'exécution des tests.

Comme toute technologie, il y a des avantages et des inconvénients à l'utilisation de l'automatisation des tests.

Tous les avantages ne peuvent pas être obtenus dans tous les projets, et tous les inconvénients ne se produisent pas dans tous les projets. En portant une attention particulière aux détails et en utilisant des processus d'ingénierie de qualité, il est possible d'augmenter les bons et de diminuer les mauvais résultats qui peuvent résulter d'un projet d'automatisation. Une constatation s'impose : une organisation n'a jamais construit **par hasard** un projet d'automatisation réussi.

Les avantages de l'automatisation peuvent inclure :

- Exécuter des tests automatisés peut être plus efficace que de les exécuter manuellement.
- Effectuer certains tests qui ne peuvent pas être effectués du tout (ou facilement) manuellement (comme les tests de fiabilité ou de performance).
- Réduire le temps nécessaire à l'exécution des tests, ce qui nous permet d'exécuter plus de tests par version construite de l'application (c'est-à-dire par build).
- Augmenter la fréquence à laquelle certains tests peuvent être effectués.
- Libérer les testeurs manuels pour exécuter des tests manuels plus intéressants et plus complexes (p. ex. tests exploratoires).
- Réduire les erreurs commises par des testeurs manuels lassés ou distraits, en particulier lors de la répétition des tests de régression.
- Exécuter des tests plus tôt dans le processus (par exemple en intégration continue pour exécuter automatiquement des tests unitaires, de composants et d'intégration), fournissant un retour d'information plus rapide sur la qualité du système et éliminant les défauts du logiciel plus tôt dans le processus.
- Exécuter des tests en dehors des heures de travail normales.
- Augmenter la confiance dans le build.

Les inconvénients peuvent comprendre :

- L'augmentation des coûts (y compris les coûts de mise en place élevés)
- Des retards, des coûts et des erreurs inhérents à l'apprentissage de nouvelles technologies par les testeurs.

- Dans le pire des cas, la complexité des tests automatisés peut devenir très importante.
- Une croissance inacceptable de la taille des tests automatisés, dépassant éventuellement la taille du système testé (SUT).
- Les compétences en développement de logiciels sont nécessaires au sein de l'équipe de test ou pour fournir un service à l'équipe de test.
- Une maintenance importante des outils, des environnements et des actifs de test peut s'avérer nécessaire.
- Il est facile d'ajouter de la dette technique, surtout lorsqu'une programmation supplémentaire est faite pour améliorer le contexte et le caractère réaliste des tests automatisés, mais qui peut être difficile à réduire (comme pour tous les logiciels).
- L'automatisation des tests fait appel à tous les processus et disciplines du développement logiciel.
- En se concentrant sur l'automatisation, les testeurs peuvent perdre de vue la gestion des risques pour le projet.
- Le paradoxe des pesticides augmente lorsque l'automatisation est utilisée, car le même test est exécuté à chaque exécution.
- Les faux positifs se produisent lorsque les défaillances d'automatisation ne sont pas des défaillances de SUT, mais sont dues à des défauts de l'automatisation elle-même.
- Sans une programmation intelligente dans les tests automatisés, les outils restent purement et simplement stupides ; les testeurs ne le sont pas.
- Les outils ont tendance à n'avoir qu'un seul fil conducteur - c'est-à-dire qu'ils ne cherchent qu'un seul résultat pour un événement - les humains peuvent comprendre ce qui s'est passé et déterminer à la volée si c'était correct.

Il y a beaucoup de contraintes sur un projet d'automatisation ; certaines d'entre elles peuvent être atténuées par une programmation intelligente et de bonnes pratiques d'ingénierie, d'autres non. Certaines de ces limitations sont d'ordre technique, d'autres concernent des aspects de gestion. Ces limitations comprennent :

- Des attentes irréalistes de la hiérarchie qui peuvent pousser l'automatisation dans une mauvaise direction.
- Une réflexion à court terme qui peut nuire à un projet d'automatisation ; ce n'est qu'en pensant à long terme que le projet d'automatisation peut réussir.
- La maturité organisationnelle est nécessaire pour réussir ; l'automatisation basée sur des processus de test médiocres n'offre que de mauvais tests plus rapidement (parfois même plus lentement).
- Certains testeurs sont parfaitement satisfaits des tests manuels et ne veulent pas automatiser les tests.
- Les oracles de test automatisés peuvent être différents des oracles de test manuels, ce qui nécessite qu'ils soient identifiés.
- Tous les tests ne peuvent ou ne doivent pas être automatisés.
- Les tests manuels seront toujours nécessaires (tests exploratoires, certaines injections de fautes, etc.).
- L'analyse, la conception et l'implémentation des tests restent encore probablement manuelles.

- Les êtres humains trouvent la plupart des bogues ; l'automatisation ne peut trouver que ce qu'elle est programmée pour trouver et est limitée par le paradoxe des pesticides.
- Un faux sentiment de sécurité dû au grand nombre de tests automatisés qui s'exécutent sans trouver beaucoup de bogues.
- Des problèmes techniques sur le projet lors de l'utilisation d'outils ou de technologies de pointe pour l'automatisation.
- Une coopération avec le développement est nécessaire, ce qui peut créer des problèmes organisationnels.

L'automatisation peut réussir et réussit souvent. Mais ce succès dépend avant tout de l'attention portée aux détails, des bonnes pratiques d'ingénierie et d'un travail important et constant sur le long terme.

## 1.2 Les tests manuels par rapport aux tests automatisés

Les premières versions des outils d'automatisation des tests ont été un échec total. Ils se sont très bien vendus, mais ont rarement fonctionné comme annoncé. Cela s'explique en partie par le fait qu'ils n'ont pas réussi à réaliser des tests logiciels pertinents et qu'ils n'ont tout simplement pas compris le rôle du testeur dans le processus de test.

Par exemple, un outil d'enregistrement/rejeu (appelé outil de capture/replay) était vendu avec le type d'instructions suivantes :

***Connectez l'outil à votre système pour le tester. Allumez l'interrupteur pour commencer l'enregistrement. Demandez au testeur d'effectuer le test. Après avoir terminé, éteindre l'outil. Il générera un script (dans un langage de programmation) qui fera exactement ce que le testeur a fait. Vous pouvez jouer ce script à chaque fois que vous voulez exécuter le test.***

Souvent, ces scripts n'ont même pas fonctionné la première fois qu'ils ont été exécutés. Tout changement dans le contexte de l'écran, le timing, les propriétés de l'interface graphique ou des centaines d'autres facteurs peuvent faire échouer le script enregistré.

Pour comprendre l'automatisation des tests, vous devez comprendre ce qu'est un script de test manuel et comment il est utilisé.

Au minimum, un script de test manuel tend à contenir des informations en trois colonnes :

La première colonne contiendra un résumé de la tâche à accomplir. Abstraite, elle n'a donc pas besoin d'être modifiée lorsque le logiciel change. Par exemple, la tâche "Ajouter un enregistrement à la base de données" est une tâche abstraite. Quelle que soit la version de la base de données, cette tâche abstraite peut être exécutée - en supposant qu'un testeur manuel possède les connaissances du domaine pour la traduire en actions concrètes.

La deuxième colonne indique au testeur quelles données utiliser pour effectuer la tâche.

La troisième colonne indique au testeur à quel comportement s'attendre.

Table 1: Fragment de test manuel (rédigé en anglais)

1	Add a record to the database	First name: Gerry Last name: Franklin SSN: 234-34-5678	Record created, Record # returned
2	Search for the name	Franklin, Gerry	Expect to find it
3	Edit the record	Occupation: Lawyer Income: \$125,000	Expect dialog verifying change
4	Check record ordering	Record # from step 1	Expect valid ordering
5	Etc.		

Le test manuel a très bien fonctionné pendant de nombreuses années parce que nous avons su créer ce type de scripts de test manuel. Cependant, le script en lui-même n'est pas ce qui permet de faire le test. Ce n'est que lorsque le script est utilisé par un testeur manuel bien informé que nous pouvons en obtenir un résultat optimal.

Que rajoute le testeur au script pour permettre une exécution correcte du test ? **Le contexte et le caractère vraisemblable.** Chaque tâche est filtrée par les connaissances du testeur : "Quelle base de données ? Quelle table ? Comment puis-je effectuer cette tâche avec cette version de cette base de données ?" Le contexte mène à certaines réponses, le caractère vraisemblable mène à d'autres pour permettre au testeur de correctement exécuter les tâches du script de test.

Un test automatisé à l'aide d'un outil d'automatisation a un contexte et un caractère vraisemblable limités. Pour exécuter une tâche particulière (p. ex. "Ajouter un enregistrement"), l'outil doit placer le SUT à l'endroit approprié pour exécuter la tâche. L'hypothèse est donc que la dernière étape du cas de test automatisé a positionné le SUT à l'endroit exact où il est possible de réaliser "Add Record".

Et si ce n'était pas le cas ? Dommage !

Non seulement l'automatisation échouera, mais le message d'erreur réel est susceptible d'être dénué de sens, puisque l'échec véritable a probablement eu lieu au cours de l'étape précédente. Un testeur qui exécute ce test n'aura jamais ce problème.

De même, lors de l'exécution d'un test manuel, le résultat d'une action est vérifié par le testeur manuel. Par exemple, si le test demandait au testeur d'ouvrir un fichier (i.e., tâche = "Ouvrir fichier"), il y a une variété de situations qui peuvent se produire. Le dossier peut s'ouvrir, ce qui est correct. Ou bien un message d'erreur, un message d'avertissement, un message d'information, un message de synchronisation et une douzaine d'autres événements peuvent en résulter. Dans chaque cas, le testeur manuel lit simplement le message, applique le caractère vraisemblable et le contexte à la situation et fait ce qu'il faut.

C'est là toute la différence entre un script automatisé et un script manuel. Le script de test manuel ne donne de la valeur ajoutée que dans les mains d'un testeur manuel. On pourrait ajouter des lignes dans le script manuel pour aider le testeur à comprendre comment traiter un point, mais le plus souvent on ne le fait pas parce que le testeur manuel a un cerveau humain, une bonne intelligence et de l'expérience, ce qui lui permet de résoudre le problème par lui-même. Et, dans le pire des cas, il peut prendre son téléphone et demander à un expert.

Examinons quelques-unes des questions auxquelles le testeur manuel peut répondre :

- Combien de temps dois-je attendre que quelque chose se produise ?
- Que faire lorsqu'on obtient un résultat de retour inattendu ? (par exemple, on a essayé d'ouvrir un fichier et le message d'erreur reçu est "Lecteur non associé").
- Que faire si un avertissement s'affiche ?
- Que faire si je suis censé attendre 5 secondes, mais que cela a pris 5,1 secondes ?

Aucune de ces questions ne peut être traitée directement par un outil d'automatisation. Un outil d'automatisation n'a pas d'intelligence ; ce n'est qu'un outil. Un script automatisé, s'il est enregistré, a une compréhension limitée de ce qu'il peut faire si quelque chose d'autre que ce qui est attendu se produit. Une seule solution est apportée par l'outil d'automatisation pour tous ces cas : remonter une erreur.

Cependant, un script automatisé peut avoir une intelligence intégrée par l'automatisme de tests via la programmation. Dès lors que les automatismes ont perçu que l'intelligence pouvait être ajoutée au script d'automatisation par la programmation, il a été possible de rendre l'automatisation plus efficace. En programmant, nous pouvons saisir les processus de pensée d'un testeur manuel, le contexte et le caractère vraisemblable du test, et commencer à faire en sorte que l'automatisation ajoute plus de valeur en complétant plus souvent le test plutôt que de le faire échouer rapidement. Notez, cependant, qu'il n'y a rien qui soit gratuit : une programmation supplémentaire pourra également nécessiter plus de maintenance ultérieure.

Tous les tests manuels ne devraient pas être automatisés. Parce qu'un script automatisé nécessite plus d'analyse, plus de conception, plus d'ingénierie et plus de maintenance qu'un script manuel, nous devons calculer le coût de sa création. Et, une fois que nous avons créé le script automatisé, nous devons le maintenir en permanence, et cela doit aussi être pris en compte. Lorsque le SUT est modifié, les scripts automatisés devront généralement être modifiés conjointement.

Inévitablement, nous trouverons de nouvelles façons dont un script automatisé pourrait échouer : le simple fait d'obtenir une valeur de retour que nous n'avons jamais vue auparavant fera échouer l'automatisation. Lorsque cela se produit, nous devons modifier, tester à nouveau et redéployer le script. Ces problèmes ne concernent généralement pas les tests manuels.

Une étude de rentabilité doit être réalisée avant et pendant que nous automatisons les tests. Est-ce que le coût global de l'automatisation de ce test sera rentabilisé en étant capable de l'exécuter N fois au cours des prochains M mois ? Souvent, nous pouvons déterminer que la réponse est non. Certains tests manuels subsisteront parce qu'il n'y a pas de retour sur investissement (ROI) positif à leur automatisation.

Certains tests manuels ne peuvent tout simplement pas être automatisés parce que le processus de raisonnement du testeur manuel est essentiel à la bonne réalisation du test. Les tests exploratoires, les tests par injection de fautes et d'autres types de tests manuels restent nécessaires au succès des tests.

## 1.3 Facteurs de succès

L'automatisation ne réussit pas par accident. Pour réussir, il faut en général :

- Un plan à long terme aligné sur les besoins de l'entreprise.
- Une gestion solide et intelligente.
- Une attention toute particulière aux détails.
- La maturité du processus.
- Une architecture et un cadre formalisés.
- Une formation adaptée.
- Des niveaux matures de documentation.

La capacité d'automatisation repose souvent sur la testabilité du système sous test (SUT). Il arrive souvent que les interfaces du SUT ne conviennent tout simplement pas aux tests. Obtenir des interfaces spécifiques ou privées (souvent appelées hooks) de la part des développeurs du système peut le plus souvent permettre de faire la différence entre réussir ou échouer à l'automatisation.

Il y a plusieurs niveaux différents d'interface que nous pouvons automatiser pour un SUT donné :

- Au niveau de l'interface graphique – IH/M : Interface Homme/Machine – qui est souvent le niveau le plus fragile et le plus sujet à l'échec.
- Au niveau API (à l'aide d'interfaces de programmation d'applications que les développeurs mettent à disposition du public ou pour un usage privé).
- Au niveau de hooks privés (APIs qui sont spécifiques aux tests)
- Au niveau du protocole (HTTP, TCP, etc.)
- Au niveau de la couche des services (SOAP, REST, etc.)

Il est à noter que Selenium fonctionne au niveau de l'IH/M. Ce syllabus contiendra des conseils et des techniques pour réduire la fragilité et améliorer la facilité d'utilisation tout en testant à ce niveau.

Les facteurs de succès de l'automatisation sont les suivants :

- Une hiérarchie formée pour comprendre ce qui est possible et ce qui ne l'est pas (les attentes irréalistes de la hiérarchie sont l'une des principales raisons de l'échec des projets d'automatisation des tests logiciels).
- Une équipe de développement pour le(s) SUT qui comprend l'automatisation et est prête à travailler avec les testeurs en cas de besoin.
- Des systèmes sous test qui sont conçus pour être testables.
- La réalisation d'une analyse de rentabilité à court, moyen et long terme.
- Le fait de disposer des bons outils pour travailler dans l'environnement et avec le(s) SUT(s).
- La bonne formation, y compris les techniques de test et de développement.
- Une architecture et un framework d'automatisation bien conçus et bien documentés pour permettre le traitement de chaque script (le syllabus ISTQB de niveau avancé Automaticien de tests – AuT - appelle cela la solution d'automatisation de test).
- Une stratégie d'automatisation des tests bien documentée, financée et soutenue par la hiérarchie.

- Un plan formel et bien documenté pour la maintenance de l'automatisation.
- Une automatisation au bon niveau d'interface en fonction du contexte des tests requis.

## 1.4 Risques et avantages de Selenium WebDriver

WebDriver est une interface de programmation pour le développement de scripts Selenium avancés utilisant les langages de programmation suivants :

- C#
- Haskell
- Java
- JavaScript
- Objective C
- Perl
- PHP
- Python
- R
- Ruby

La plupart de ces langages possèdent également des frameworks de test open-source disponibles.

Les navigateurs supportés par Selenium (et le composant nécessaire pour tester avec) sont les suivants :

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

Selenium WebDriver fonctionne en utilisant ses APIs (Application Programming Interfaces) pour pouvoir réaliser des appels directs vers un navigateur en utilisant le support natif de chaque navigateur pour l'automatisation. Chaque navigateur supporté a un fonctionnement légèrement différent.

Comme tout outil, l'utilisation de Selenium WebDriver comporte à la fois des avantages et des risques.

Plusieurs des avantages dont une organisation peut bénéficier sont les mêmes que n'importe quel autre outil d'automatisation, notamment :

- L'exécution des tests peut être uniforme et reproductible.
- Bien adapté aux tests de régression
- Parce qu'il teste au niveau de l'interface utilisateur, il peut détecter des défauts non détectés par des tests au niveau de l'API.
- Un investissement initial moins élevé parce qu'il s'agit d'un logiciel libre.

- La compatibilité avec différents navigateurs permet d'effectuer des tests de compatibilité.
- Le support de différents langages de programmation pour être utilisé par un plus grand nombre de personnes.
- Parce qu'il implique une compréhension profonde du code, ce qui est utile pour les équipes agiles.

Avec les avantages, vous avez aussi les risques. Voici les risques associés à l'utilisation de Selenium WebDriver :

- Les organisations sont souvent tellement préoccupées par les tests via l'interface graphique qu'elles oublient que la pyramide des tests suggère que davantage de tests unitaires/composants soient effectués.
- Lors du test d'un workflow dans un contexte d'intégration continue, cette automatisation peut rendre le build beaucoup plus long que prévu.
- Les modifications apportées à l'interface utilisateur ont un impact sur les tests au niveau du navigateur plus important que les tests au niveau unitaire ou de l'API.
- Les testeurs manuels sont plus efficaces pour trouver des bogues que l'automatisation.
- Les tests difficiles à automatiser peuvent être ignorés.
- L'automatisation doit être exécutée souvent pour obtenir un ROI (retour sur investissement) positif. Si l'application Web est plus ou moins stable, l'automatisation peut ne pas s'amortir.

## 1.5 Selenium WebDriver dans l'architecture d'automatisation des tests

L'Architecture d'Automatisation des Tests (AAT), tel que défini par l'ISTQB dans le syllabus de niveau avancé Automaticien de tests, est un ensemble de couches, de services et d'interfaces d'une Solution d'Automatisation des Tests (SAT).

L'AAT se compose de quatre couches (voir l'illustration ci-dessous) :

- Couche de génération des tests : prend en charge la conception manuelle ou automatisée des cas de test.
- Couche de définition des tests : prend en charge la définition et l'implémentation des cas de test et/ou des suites de tests.
- Couche d'exécution des tests : supporte à la fois l'exécution des tests automatisés et l'enregistrement des résultats.
- Couche d'adaptation des tests : fournit les objets et le code nécessaires à l'interface avec le SUT (système testé) à différents niveaux.

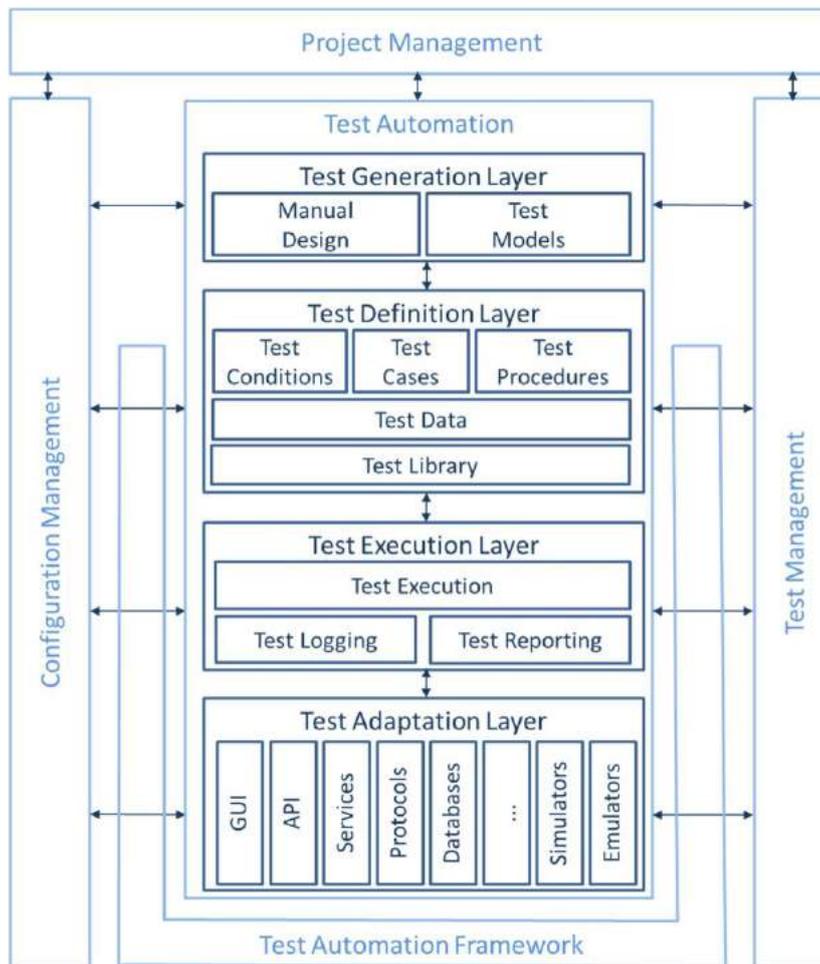


Figure 1: Une AAT générique (extrait du syllabus AuT)

Selenium WebDriver s'insère dans la couche d'adaptation des tests, fournissant un moyen par programmation d'accéder aux SUT via l'interface du navigateur.

La couche d'adaptation des tests facilite la séparation entre le SUT (et ses interfaces) et les tests et suites de tests que nous voulons exécuter sur le SUT.

En principe, cette séparation permet aux cas de test d'être plus abstraits et déconnectés du système testé. Lorsque le système testé change, les cas de test eux-mêmes peuvent ne pas avoir besoin de changer, en supposant que la même fonctionnalité est fournie - mais juste d'une manière légèrement différente.

Lorsque le SUT change, la couche d'adaptation des tests, dans ce cas, WebDriver, permet de modifier le test automatisé, ce qui lui permet d'exécuter le cas de test concret réel sur l'interface modifiée du SUT.

Le script automatisé avec WebDriver utilise efficacement l'API pour communiquer entre le test et le SUT comme suit :

- Le cas de test requiert l'exécution d'une tâche.
- Le script appelle une API dans WebDriver.
- L'API se connecte à l'objet approprié au sein du SUT.
- Le SUT répond comme demandé (ou non, en cas d'échec du pas de test).

- Le succès ou l'échec est communiqué au scénario.
- En cas de succès, l'étape suivante du scénario de test est appelée dans le script.

## 1.6 Métriques pour l'automatisation

Il y a un principe ancien en management qui dit : "Ce qui est mesuré est fait". Les mesures sont une de ces pratiques que beaucoup d'automatiseurs de tests aiment ignorer ou occulter, parce qu'elles sont souvent difficiles à quantifier et à prouver.

Le problème est que l'automatisation tend à être assez coûteuse en effort humain et en coût d'outillage. Il y a peu ou pas de retour sur investissement positif à très court terme ; la valeur vient à long terme. Des années plutôt que des mois. Demander à la Direction un investissement important en temps et en ressources et ne pas faire une analyse de rentabilité prouvable (y compris au travers de la collecte de métriques pour fournir des preuves), c'est leur demander de nous croire sur parole, tel un acte de foi. Pour la Direction, cette approche n'est pas acceptable.

Nous devons établir des métriques pertinentes montrant la valeur de l'automatisation si nous voulons que le projet d'automatisation survive à l'analyse de la rentabilité financière. Le syllabus AuT mentionne plusieurs domaines dans lesquels les métriques peuvent être utiles, notamment au niveau :

- utilisabilité,
- maintenabilité,
- performance,
- et fiabilité.

Beaucoup des métriques qui sont mentionnées dans le syllabus AuT ciblent les grands projets d'automatisation plutôt que les petits projets d'automatisation comme pourraient l'être ceux utilisant Selenium WebDriver. Nous allons donc plutôt nous concentrer sur des projets plus petits dans cette section. Si vous travaillez sur un projet d'automatisation complexe et de grande envergure, veuillez-vous référer plutôt au syllabus AuT.

La collecte de données inadaptées est susceptible d'être une source de confusion pour une petite équipe qui essaie simplement de lancer un projet pour la première fois. L'un des problèmes liés à l'identification de métriques significatives à collecter est que, alors que le temps d'exécution de l'automatisation des tests tend à être inférieur au temps de test manuel équivalent, l'analyse, la conception, le développement, le dépannage et la maintenance des tests automatisés prennent beaucoup plus de temps que le même travail pour les tests manuels.

Essayer de recueillir des métriques pertinentes pour déterminer le ROI revient souvent à comparer des choux et des carottes. Par exemple, la création d'un cas de test manuel spécifique peut prendre 1 heure. Trente minutes pour l'exécuter. Supposons que nous prévoyons d'exécuter le test trois fois pour cette version. Il nous faut 2,5 heures pour ce test. Ce sont des mesures simples à réaliser pour les tests manuels.

Mais combien coûtera l'automatisation de ce même test ? La première chose à faire est de déterminer s'il est utile d'automatiser ce test. Ensuite, en supposant que nous décidons qu'il doit être automatisé, comment l'automatiser, écrire le code nécessaire, déboguer le code et s'assurer qu'il fait ce à quoi nous nous attendons peut prendre plusieurs heures. Chaque test est susceptible de présenter des difficultés particulières ; comment pouvons-nous évaluer cela ?

Le test manuel, étant plus abstrait, n'a probablement pas besoin d'être modifié lorsque l'interface graphique du SUT change. Le script automatisé aura sûrement besoin d'être adapté, même pour des changements mineurs de l'interface graphique. Nous pouvons minimiser cela par une bonne architecture et une bonne conception du framework, mais les changements prendront néanmoins du temps. Mais alors, comment estimer le nombre de changements prévus ?

En cas d'échec d'un test manuel, le diagnostic du problème est généralement simple. L'analyste de test peut généralement analyser facilement ce qui s'est mal passé afin de pouvoir rédiger le rapport d'incident.

Une défaillance dans le contexte de l'automatisation peut prendre plus de temps à diagnostiquer (voir la section sur les logs ci-dessous).

Comment calculer combien de fois le test pourrait échouer, surtout si ces échecs proviennent souvent de l'automatisation elle-même, et non d'une défaillance du SUT ? Pour rendre encore plus difficile le calcul du ROI, il faut prendre en compte le fait que l'automatisation nécessite un investissement initial important qui n'est pas nécessaire pour les tests manuels. Cet investissement comprend l'achat d'un ou plusieurs outils, les environnements appropriés pour faire fonctionner les outils, la création de l'architecture et du framework pour faciliter la mise en œuvre de l'automatisation, la formation (ou l'embauche) des automatiseurs de tests, et d'autres coûts fixes.

Cet investissement doit être ajouté à tout calcul de retour sur investissement que nous pourrions faire. Parce qu'il s'agit de coûts fixes, l'obtention d'un ROI positif implique généralement que nous devons automatiser et exécuter un grand nombre de tests. Cela exige un niveau de scalabilité plus élevé, ce qui augmentera le coût de l'automatisation.

Un autre problème avec de nombreuses métriques d'automatisation est qu'elles doivent souvent être estimées plutôt que mesurées de manière directe ; cela peut introduire des biais lors de la preuve de la rentabilité de l'automatisation.

Pour un petit projet de démarrage, voici quelques mesures qui pourraient être utiles :

- Coûts fixes pour mettre en place et faire fonctionner l'automatisation
- Effort de test de régression qui a été économisé par l'automatisation.
- Effort déployé par l'équipe d'automatisation en support à l'automatisation.
- Couverture
  - Au niveau des tests unitaires, par le biais de la couverture des instructions et des décisions.

- Au niveau des tests d'intégration, de l'interface ou de la couverture des flux de données.
- Au niveau des tests système, par la couverture des exigences, des caractéristiques ou des risques identifiés.
- Des configurations testées.
- Des User stories couvertes (en Agile).
- Des cas d'utilisation couverts.
- Configurations couvertes testées.
- Nombre de passages réussis entre les échecs.
- Patterns fréquents d'échecs de l'automatisation (recherche de problèmes communs par le suivi des causes racines des échecs).
- Nombre de défaillances constatées des tests automatisés eux-mêmes, par rapport aux défaillances effectives du SUT trouvées avec ces tests automatisés.

Pour conclure, voici un dernier point sur les métriques. Essayez de trouver des métriques significatives à collecter sur votre projet qui aident à expliquer pourquoi le projet est pertinent et important. Négocier avec la direction, en s'assurant qu'ils comprennent qu'il est difficile de prouver la valeur de l'automatisation au début et qu'il faut souvent beaucoup de temps pour faire des progrès raisonnables vers des métriques significatives. Il peut être tentant d'utiliser des scénarios idylliques et de faire preuve d'imagination pour prouver la valeur de l'automatisation. Ce n'est cependant pas la meilleure façon de convaincre.

Lorsqu'il est bien fait, un projet d'automatisation a de sérieuses chances de donner de la valeur ajoutée à votre organisation, même si cela est difficile à prouver.

## 1.7 La boîte à outils Selenium

Selenium WebDriver n'est pas le seul outil de l'écosystème open-source Selenium. Cet écosystème se compose de quatre outils, dont chacun joue un rôle différent dans l'automatisation des tests :

- Selenium IDE
- Selenium WebDriver
- Selenium Grid
- Selenium Standalone Server

Au tout début de la longue histoire des outils Selenium, il existait l'outil appelé Selenium RC, qui implémentait la version 1 de Selenium. Cet outil n'est plus utilisé.

Selenium IDE est un add-on aux navigateurs web Chrome et Firefox. Selenium IDE ne fonctionne pas en tant qu'application autonome. Sa fonction principale est d'enregistrer et de rejouer les actions de l'utilisateur sur les pages Web. Selenium IDE permet également à un automaticien de tests d'insérer des points de vérification pendant l'enregistrement. Les scripts enregistrés peuvent être sauvegardés sur disque sous forme de tableaux HTML ou exportés vers différents langages de programmation.

Les principaux avantages de Selenium IDE sont sa simplicité et ses localisateurs d'éléments de bonne qualité. Son principal inconvénient est le manque de variables, de procédures et

d'instructions de flux de contrôle ; en tant que tel, il n'est pas véritablement utilisable pour créer des tests robustes et automatisés. Selenium IDE est principalement utilisé pour enregistrer des scripts provisoires (par exemple, à des fins de débogage) ou simplement pour identifier des localisateurs.

Selenium WebDriver est principalement un framework permettant aux scripts de test de contrôler les navigateurs web. Il est basé sur HTTP et a été normalisé par l'organisme W3C. Ce syllabus présente les caractéristiques de base de ce protocole au Chapitre 3. Selenium WebDriver a des connecteurs pour de nombreux langages de programmation différents, par exemple Java, Python, Ruby, Ruby, C#. Dans ce syllabus, Python est utilisé pour montrer des exemples d'utilisation de différents objets WebDriver et de leurs méthodes.

L'utilisation de bibliothèques qui implémentent l'API WebDriver pour divers langages de programmation permet aux automaticiens de tests de combiner la capacité de WebDriver à contrôler les navigateurs Web avec la puissance des langages de programmation généraux. Cela permet aux automaticiens d'utiliser les bibliothèques de ces langages pour construire des frameworks d'automatisation de tests complexes comprenant la capture des logs, le traitement des assertions, le multithreading et d'autres choses encore.

Pour obtenir un résultat à long terme, les frameworks d'automatisation des tests doivent être réalisés avec soin et selon de bons principes de conception, comme décrit dans la section 1.5.

Certains navigateurs Web ont besoin de processus WebDriver supplémentaires pour démarrer de nouvelles instances de test et les contrôler. Dans ce cas, un script appelle des commandes de la bibliothèque et la bibliothèque à travers le WebDriver transmet ces commandes au navigateur Web. Cette question sera abordée au chapitre trois.

Un autre outil qui peut être utile dans un environnement de test est Selenium Grid. Il permet d'exécuter des scripts de test sur plusieurs machines avec des configurations différentes. Il permet l'exécution distribuée et simultanée de cas de test. L'architecture de Selenium Grid est très flexible. Il peut être configuré pour utiliser de nombreuses machines physiques ou virtuelles avec différentes combinaisons de systèmes d'exploitation et de versions de navigateurs Web.

Au centre de Selenium Grid, il y a un hub qui contrôle les autres nœuds et agit en tant que point de contact unique pour les scripts de test. Les scripts de test qui exécutent les commandes WebDriver n'ont pas besoin (dans la plupart des cas) d'être modifiés pour fonctionner sur différents systèmes d'exploitation ou navigateurs Web.

Le dernier outil de l'écosystème Selenium que nous mentionnons dans ce syllabus est Selenium Standalone Server. Cet outil est écrit en Java et est livré sous la forme d'un fichier .jar qui implémente les fonctions des hubs et des nœuds pour Selenium Grid. Cet outil doit être démarré séparément (en dehors des scripts de test) et configuré correctement pour jouer son rôle dans l'environnement de test.

Une description plus détaillée de Selenium Grid et Selenium Standalone Server est en dehors du champ du présent syllabus.

# Chapter 2 - Technologies Internet pour l'automatisation des tests d'applications Web

## Termes

sélecteur CSS, HTML, tag, XML, XPath

## Objectifs d'apprentissage pour les technologies Internet pour l'automatisation des tests d'applications Web

STF-2.1 (K3) Comprendre et écrire des documents HTML et XML

STF-2.2 (K3) Appliquer les XPath pour rechercher des documents XML

STF-2.3 (K3) Appliquer les localisateurs CSS pour trouver des éléments de documents HTML

## 2.1 Comprendre HTML et XML

### 2.1.1 Comprendre HTML

Il ne serait pas exagéré de dire que HTML (HyperText Markup Language) a permis la diffusion très importante du Web que nous connaissons. Un document HTML est un fichier texte simple qui contient des éléments qui spécifient certaines significations contextuelles lorsque le document est analysé. Les éléments se combinent pour déterminer comment un navigateur doit afficher ces parties du document. Essentiellement, HTML décrit sémantiquement la structure d'une page Web.

L'avantage le plus important de l'utilisation de HTML pour spécifier des pages Web est sans doute l'applicabilité universelle du langage. Lorsqu'il est écrit correctement, n'importe quel navigateur sur n'importe quel système informatique peut afficher correctement la page.

L'apparence exacte de la page peut changer à certains niveaux, selon le type d'ordinateur (p. ex., PC ou téléphone intelligent), l'écran, la vitesse de connexion Internet et le navigateur.

Le mérite de l'invention du Web est attribué à Timothy Berners Lee. Alors qu'il travaillait pour le CERN (Conseil Européen pour la Recherche Nucléaire) en 1980, il a proposé d'utiliser l'hypertexte pour permettre le partage et la mise à jour de l'information entre chercheurs. Puis, en 1989, il a mis en place la première communication efficace entre un poste client et un serveur HTTP (HyperText Transfer Protocol), inaugurant le Web et transformant le monde tel que nous le connaissons aujourd'hui.

Les éléments HTML sont insérés en étant souvent entourés de balises définies par des chevrons, comme on peut le voir ci-dessous :

```

<!DOCTYPE html>
<html>
  <head>
    <title> Page Title</title>
  </head>
  <body>
    <h1>This is a Heading</h1>
    <p>This is a paragraph</p>
  </body>
</html>

```

Figure 2: Example HTML

Certaines balises introduisent directement le contenu dans la page en cours de visualisation (par exemple, **<img .../>** se traduira par une image placée dans la page.) D'autres balises entourent et fournissent des informations sémantiques sur la façon dont l'élément doit être restitué (comme vu ci-dessus pour l'élément d'en-tête **<h1>...</h1>**). Nous abordons les balises dans les paragraphes suivants.

Longtemps, la version HTML utilisée a été stable avec la version 4.01. Mais en 2014, l'organe directeur qui contrôle le HTML a publié la version actuelle, HTML 5.

HTML est un langage flexible, permettant une certaine variation dans la façon dont les balises sont utilisées (par exemple, certaines balises peuvent ne pas avoir de balise de fermeture). XML, qui sera discuté ci-dessous, est un langage plus restrictif que HTML, exigeant que chaque page soit "bien formée", chaque balise d'ouverture étant équilibrée par une balise de fermeture. Quand il est écrit d'une manière bien formée (c'est-à-dire que toutes les balises ouvertes sont associées à des balises fermées), HTML constitue un sous-ensemble de XML.

L'automatisation avec Selenium nécessite une compréhension des balises HTML. Pour automatiser une interface graphique, l'automaticien de tests doit être capable d'identifier chaque commande particulière dans l'écran manipulé. La recherche dans la page HTML permet à l'automaticien de détecter distinctement les contrôles qui seront placés sur la page affichée par le navigateur. Pour identifier les contrôles, l'automaticien doit comprendre la disposition et la logique de la page HTML ; cela se fait en connaissant et en analysant les balises.

Les éléments HTML comportent généralement une balise de début et une balise de fin. La balise de fin est la même que la balise de début sauf que la balise de fin est précédée d'une barre oblique comme indiqué ci-dessous :

**<p>Paragraphe de texte </p>**

Certains éléments peuvent être fermés dans la balise d'ouverture ; par exemple, l'élément de saut de ligne vide comme suit :

`<br />`

Une implémentation moins stricte du saut de ligne, pouvant poser des problèmes pour certains navigateurs serait :

`<br>`

Voici les balises que tout automaticien de tests avec Selenium devrait connaître.

Table 2: Balises HTML de base

<u>Balise</u>	<u>Utilisation</u>
<code>&lt;html&gt; ... &lt;/html&gt;</code>	<u>Indique la racine du document HTML</u>
<code>&lt;!DOCTYPE&gt;</code>	<u>Définit le type de document (pas nécessaire dans HTML 5)</u>
<code>&lt;head&gt; ... &lt;/head&gt;</code>	<u>Définition et métadonnées pour le document</u>
<code>&lt;body&gt; ... &lt;/body&gt;</code>	<u>Définit le contenu principal du document</u>
<code>&lt;p&gt; ... &lt;/p&gt;</code>	<u>Définit un paragraphe</u>
<code>&lt;br /&gt;</code>	<u>Insère un saut de ligne simple</u>
<code>&lt;div&gt; ... &lt;/div&gt;</code>	<u>Définit une section dans le document</u>
<code>&lt;-- ... --&gt;</code>	<u>Définit un commentaire (peut être sur plusieurs lignes)</u>

Les balises d'en-tête définissent différents niveaux d'en-têtes. Le format réel du texte (taille, gras, police) peut être spécifié dans les feuilles de style CSS.

Table 3: Balises d'en-tête

<u>Balise</u>	<u>Affichage</u>
<code>&lt;h1&gt; En-tête 1 &lt;/h1&gt;</code>	<b>En-tête 1</b>
<code>&lt;h2&gt; En-tête 2 &lt;/h2&gt;</code>	<b>En-tête 2</b>
<code>&lt;h3&gt; En-tête 3 &lt;/h3&gt;</code>	<b>En-tête 3</b>
<code>&lt;h4&gt; En-tête 4 &lt;/h4&gt;</code>	<b>En-tête 4</b>
<code>&lt;h5&gt; En-tête 5 &lt;/h5&gt;</code>	<b>En-tête 5</b>
<code>&lt;h6&gt; En-tête 6 &lt;/h6&gt;</code>	<b>En-tête 6</b>

Les liens et les images sont essentiels pour créer des pages de navigation bien conçues. Ils sont faciles à réaliser avec HTML.

`<a href="URL">texte du lien</a>`

Cette combinaison de symboles commence par une balise `<a...>` ancre et se termine par un `</a>`. Ceux-ci définissent un hyperlien sur lequel on peut cliquer. La référence `href="URL"` est un attribut qui indique la destination du lien Web. Le texte du lien entre les balises représente le texte qui apparaîtra dans le lien sur lequel il faut cliquer pour emmener l'utilisateur vers l'URL cible.

``

La balise de base ici, `<img.../>`, définit une image qui sera placée à cet endroit dans le document. L'attribut `src="pulpitrock.jpg"` est l'adresse du lien de l'image réelle qui sera affichée. L'autre attribut, `alt="Mountain view"`, représente le texte qui sera affiché si l'image ne peut pas être trouvée ou affichée.

Table 4: Balises des listes et des tableaux

<b>Balise</b>	<b>Utilisation</b>
<code>&lt;ul&gt; ... &lt;/ul&gt;</code>	Définit une liste non ordonnée (à puces)
<code>&lt;ol&gt; ... &lt;/ol&gt;</code>	Définit une liste ordonnée (numérotée)
<code>&lt;li&gt; ... &lt;/li&gt;</code>	Définit un élément de liste (pour <code>&lt;ul&gt;</code> ou <code>&lt;ol&gt;</code> )
<code>&lt;table&gt; ... &lt;/table&gt;</code>	Définit un tableau HTML
<code>&lt;tr&gt; ... &lt;/tr&gt;</code>	Définit une ligne de tableau
<code>&lt;th&gt; ... &lt;/th&gt;</code>	Définit l'en-tête de colonne d'un tableau
<code>&lt;td&gt; ... &lt;/td&gt;</code>	Définit une cellule de données de table
<code>&lt;tbody&gt; ... &lt;/tbody&gt;</code>	Regroupe le contenu du texte dans un tableau HTML
<code>&lt;thead&gt; ... &lt;/thead&gt;</code>	Définit un en-tête de tableau HTML
<code>&lt;tfoot&gt; ... &lt;/tfoot&gt;</code>	Définit un pied de page de tableau HTML
<code>&lt;colgroup&gt; ... &lt;/colgroup&gt;</code>	Regroupe les colonnes du tableau pour le formatage

Les listes sont simples à définir et à restituer. Par exemple, le code HTML de la figure 3 restituera la liste présentée en figure 4.

```
<!DOCTYPE html>
<html>
<head>

<body>

<h4>An Unordered List containing an Ordered List</h4>
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <ol>
    <li>Oolong</li>
    <li>Black</li>
    <li>Earl Grey</li>
  </ol>
  <li>Milk</li>
</ul>
</body>
</html>
```

Figure 3: Exemple de liste HTML

#### **An Unordered List containing an Ordered List**

- Coffee
- Tea
  1. Oolong
  2. Black
  3. Earl Grey
- Milk

Figure 4: Une liste non ordonnée contenant une liste ordonnée

Les tableaux sont également simples à créer et à afficher. Les données obtenues lors des tests sont souvent renvoyées dans des tableaux, si bien que les automaticiens de tests les utilisent souvent. Le code présenté en figure 5 affichera le tableau juste après (Table 5).

```

<!DOCTYPE html>
<html>
  <head>
    <style>
      table, th, td {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>

    <table>
      <tr>
        <th>Year</th>
        <th>Car Payment</th>
      </tr>
      <tr>
        <td>2017</td>
        <td>$3,780</td>
      </tr>
      <tr>
        <td>2018</td>
        <td>$2,905</td>
      </tr>
      <tr>
        <td>2019</td>
        <td>$4,812</td>
      </tr>
      <tr>
        <td>2020</td>
        <td>$1,790</td>
      </tr>
    </table>
  </body>
</html>

```

Figure 5: Code de génération de table

Table 5: Table restituée

<b>Year</b>	<b>Car Payment</b>
2017	\$3,780
2018	\$2,905
2019	\$4,812
2020	\$1,790

Les formulaires HTML et les contrôles associés sont utilisés pour recueillir les commentaires des utilisateurs. Vous trouverez ci-dessous les balises nécessaires pour restituer les formulaires et les contrôles qui s'y trouvent. Les utilisateurs de Selenium WebDriver doivent souvent utiliser ces formulaires et contrôles pour automatiser leurs tests.

Les balises suivantes sont utilisées pour créer des contrôles à l'écran.

## **<form> ... </form>**

Définit un formulaire HTML pour la saisie utilisateur.

## **<input>**

Définit un contrôle d'entrée. Le type de contrôle est défini par le type d'attribut **type=**. Les types possibles comprennent le texte, le bouton radio, la case à cocher, la soumission du formulaire, etc. Par exemple, les lignes suivantes seront affichées comme suit :

```
<form action="/action_page.php">  
First name: <input type="text" name="FirstName" value="Mortey"><br>  
Last name: <input type="text" name="LastName" value="Moose"><br>  
<input type="submit" value="Submit">  
</form>
```

Figure 6: Exemple de formulaire en HTML

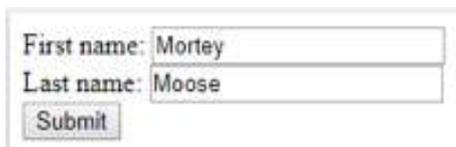
A screenshot of a web form. It contains two text input fields. The first field is labeled "First name:" and contains the text "Mortey". The second field is labeled "Last name:" and contains the text "Moose". Below the fields is a button labeled "Submit".

Figure 7: Champs d'entrée du formulaire

## **<textarea> ... </textarea>**

Définit une commande d'entrée de plusieurs lignes. La zone de texte peut contenir un nombre illimité de caractères. Par exemple, les lignes suivantes seront affichées comme suit :

```
<textarea rows="4" cols="50">  
Selenium allows you to automate browsers with  
maximum return and minimum effort.  
</textarea>
```

Figure 8: Code HTML pour le contrôle d'entrée multiligne

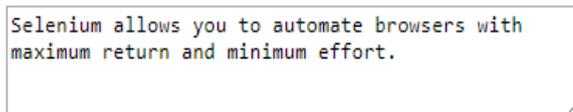
A screenshot of a multi-line text area. It contains the text "Selenium allows you to automate browsers with maximum return and minimum effort." The text area has a white background and a thin border.

Figure 9: Contrôle d'entrée multiligne

## **<button>**

Définit un bouton cliquable.

## **<select> ... </select>**

Définit une liste déroulante. Lorsqu'il est utilisé avec les balises `<option>....</option>`, l'auteur peut définir une liste déroulante et la remplir comme ci-dessous :

```
<select>
  <option value="volvo">Volvo</option>
  <option value="saab" >Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

Figure 10: Code pour le contrôle utilisant la balise Select

Le code ci-dessus affichera la liste déroulante suivante :

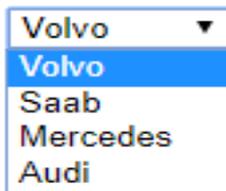


Figure 11: Une liste déroulante

### **<fieldset> ... </fieldset>**

Ces balises permettent à l'auteur de regrouper les éléments liés dans un formulaire. Lorsqu'il est utilisé avec la balise **<legend>**, il affiche une boîte nommée autour des contrôles qui sont considérés liés de la manière suivante :

```
<fieldset>
  <legend>Child 1:</legend>
  Name: <input type="text"><br>
  Email: <input type="text"><br>
  Date of birth: <input type="text">
</fieldset>
```

Figure 12: Code de création d'un groupe de champs

Le code ci-dessus générera le rendu suivant :

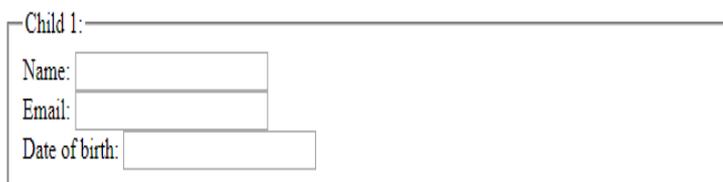


Figure 13: Éléments regroupés sur le formulaire

## 2.1.2 Comprendre XML

XML (eXtensible Markup Language) est un langage de balisage utilisé pour définir les règles de formatage des documents d'une manière qui est lisible par une machine mais qui est aussi très lisible par l'homme. XML a été conçu pour accentuer la simplicité et la facilité d'utilisation en Web. Bien qu'il soit utilisé dans les documents, XML permet la représentation de structures de données qui peuvent être créées à la volée.

HTML a été conçu pour afficher les données en mettant l'accent sur la présentation des données. XML a été conçu pour être un outil indépendant du logiciel et du matériel qui peut être utilisé pour transmettre et stocker des données dans un format lisible.

Les balises XML ne sont pas prédéfinies comme le sont les balises HTML. A la place, les balises sont spécifiées par le créateur du document XML selon les critères qu'il souhaite utiliser. Le format des balises est comparable au format HTML. Par exemple, voici un ensemble de champs en XML :

```
<?xml version="1.0" ?>
<note>
  <date>2018-06-12</date>
  <hour>10:30</hour>
  <to>Francis</to>
  <from>Morrow</from>
  <body>Please pick me up this weekend!</body>
</note>
```

*Figure 14: Exemple de code XML*

Notez que chaque balise d'ouverture, **<from>**, a une balise de fermeture correspondante, **</from>**. La construction complète s'appelle un élément.

Les sections peuvent être incorporées dans d'autres sections comme indiqué. Un document XML forme toujours une arborescence. Le premier élément de la figure ci-dessus indique qu'il s'agit d'un document XML.

En plus des balises, XML prend en charge les attributs qui fournissent des informations supplémentaires sur l'élément auquel ils sont associés. Un attribut consiste en une paire de termes séparés par un signe égal. Par exemple :

```
<person gender="female">
```

L'attribut est contenu dans les balises de l'élément. Plutôt que d'utiliser un attribut, la même information peut être utilisée comme élément. Ainsi, les deux exemples suivants contiennent exactement les mêmes informations.

```

<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

```

Figure 15: L'attribut par rapport à l'élément

Les attributs ne sont pas aussi flexibles que les éléments. Par exemple, les points suivants sont soulignés par le W3C, l'organisme qui contrôle la norme XML :

- Les attributs ne peuvent pas contenir plusieurs valeurs (les éléments le peuvent).
- Les attributs ne peuvent pas contenir de structures arborescentes (les éléments le peuvent).
- Les attributs ne sont pas facilement extensibles (pour des changements futurs).

Différents types de PC stockent les données de différentes manières, et de façon souvent incompatibles. XML permet à ces différents ordinateurs de partager des données parce que les données XML sont stockées en format texte brut. Il n'y a pas besoin de transferts complexes entre ordinateurs car ainsi ils peuvent communiquer à l'aide de fichiers texte.

XML sépare les données de la façon dont elles sont présentées. Ainsi, les mêmes données XML peuvent être présentées de la manière dont l'auteur le souhaite.

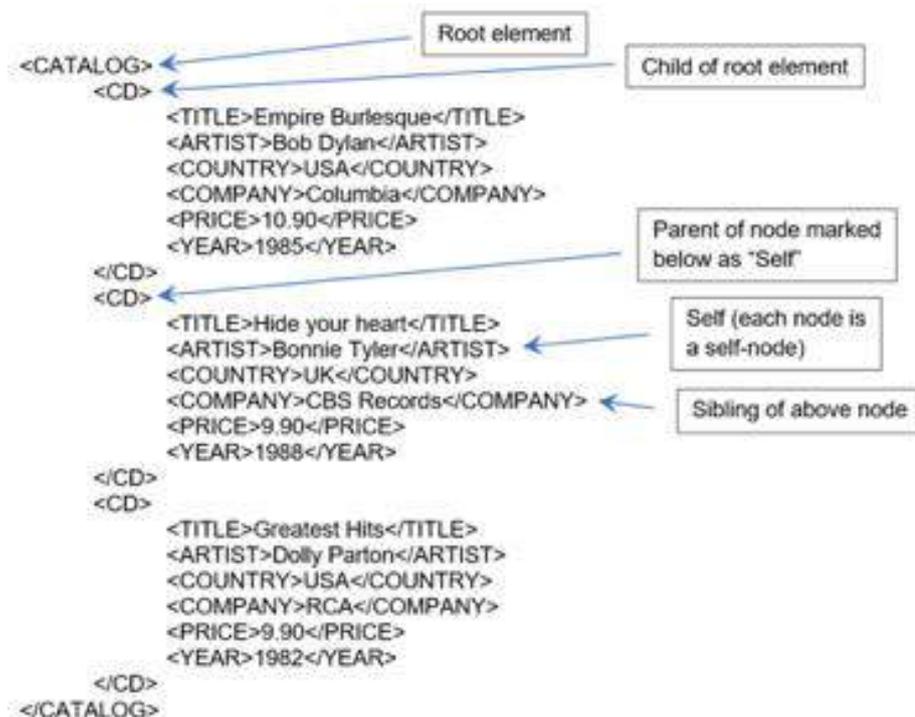


Figure 16: Base de données en XML (partielle)

Comme XML décrit toujours un format arborescent, nous pouvons définir des relations spécifiques entre les éléments. Ces relations peuvent être définies comme suit :

- Le nœud courant (« **current** ») est n'importe quel nœud arbitraire que nous choisissons. Toutes les autres relations dérivent du nœud courant. Dans la figure ci-dessus, nous avons choisi "Bonnie Tyler" comme nœud courant. Toutes les relations se réfèrent alors à ce nœud.
- Chaque nœud peut s'auto-identifier en tant que tel (« **self** »).
- Un nœud parent (« **parent** ») est toujours un niveau plus élevé dans la hiérarchie que le nœud sélectionné en tant que nœud courant. Chaque nœud et attribut d'élément a exactement un parent (à l'exception de l'élément racine).
- Un nœud enfant (« **child** ») est toujours un niveau en dessous de son parent dans la hiérarchie.
- Un nœud frère (« **sibling**») se trouve au même niveau que le nœud actuel, sous le même parent.
- Un nœud ancêtre (« **ancestor** ») est un nœud dans un chemin direct à partir du nœud courant jusqu'à son parent, grand-parent, arrière-grand-parent, arrière-grand-parent, etc.
- Un nœud descendant (« **descendent** ») est un nœud qui descend directement du nœud courant dans la hiérarchie (c.-à-d. un enfant, un enfant d'un enfant, etc.).

## 2.2 XPath et recherche dans les documents HTML

Comme mentionné ci-dessus, pour automatiser avec Selenium, un automaticien de tests doit être capable de localiser n'importe quel objet ou élément donné dans un document XML. L'un des moyens les plus puissants pour localiser des éléments spécifiques est d'utiliser XPath.

XPath utilise des expressions de chemin d'accès pour identifier et naviguer dans un document XML. Puisque HTML est un sous-ensemble de XML, XPath peut aussi être utilisé pour rechercher dans les documents HTML. Ces expressions de chemin sont en rapport avec les expressions de chemin qui peuvent être utilisées avec un système de fichiers informatique traditionnel qui permet à un utilisateur de parcourir une hiérarchie de dossiers.

Les expressions suivantes permettent de sélectionner les nœuds :

*Table 6: Expressions XPath*

Expression	Description
<b>LeNoeud</b>	Sélectionne tous les éléments avec le nom "LeNoeud"
/	Sélectionne à partir de l'élément racine
//	Retourne les descendants de l'élément courant
.	Retourne l'élément courant
..	Sélectionne le parent de l'élément courant
@	Sélectionne un attribut de l'élément courant

Vous trouverez ci-dessous un exemple de document XML ainsi que des exemples d'utilisation de XPath.

```

<?xml version="1.0" encoding="UTF-8"?>
<Magazines>
  <Magazine>
    <title lang="en">Time</title>
    <price>9.99</price>
  </Magazine>
  <Magazine>
    <title lang="en">Newsweek</title>
    <price>8.95</price>
  </Magazine>
</Magazines>

```

Figure 17: Exemple de base de données XML (partielle)

Ci-dessous se trouvent quelques expressions de chemin et les résultats qui en découlent.

Table 7: Path Expressions

Expression de chemins	Résultat
Magazines	Sélectionne tous les nœuds portant le nom "Magazines"
/Magazines	Sélectionne l'élément racine "Magazines"
Magazines/Magazine	Sélectionne tous les éléments "Magazine" de "Magazines"
//Magazines	Sélectionne tous les éléments "Magazine" dans le document
Magazines//Magazine	Sélectionne tous les éléments "Magazine" qui sont des descendants de "Magazines"
//@lang	Sélectionne tous les attributs nommés lang

Les prédicats sont utilisés pour trouver un élément spécifique ou un élément qui contient une valeur spécifique. Les prédicats sont toujours entourés de crochets et apparaissent directement après le nom de l'élément.

Table 8: Expressions de chemins à l'aide de prédicats

Expression de chemins	Résultat
/Magazines/Magazine[1]	Sélectionne le premier élément Magazine
/Magazines/Magazine[last()]	Sélectionne le dernier élément enfant Magazine
/Magazines/Magazine[last()-1]	Sélectionne l'avant-dernier élément Magazine
//title[@lang]	Sélectionne tous les éléments de titre avec l'attribut "lang"
//title[@lang='en']	Sélectionne tous les éléments de titre avec l'attribut lang=fr

XPath dispose d'expressions appelées wildcards permettant de sélectionner les nœuds XML en fonction de critères spécifiés.

Table 9: Wildcards en XPath

Wildcard	Description
----------	-------------

*	Correspond à n'importe quel élément du nœud
@*	Correspond à n'importe quel attribut du nœud
Node()	Correspond à n'importe quel nœud de n'importe quel type

Il y a une grande variété d'opérateurs qui peuvent être utilisés dans les expressions XPath.

Table 10: Opérateurs en XPath

Opérateur	Description	Exemple
	Sélectionne plusieurs chemins	//Magazine   //CD
+	Addition	2 + 2
-	Soustraction	5 - 3
*	Multiplication	8 * 8
div	Division	14 div 2
mod	Modulo (le reste après division)	7 mod 3
=	Egal	price=4.35
!=	Différent	price!=4.35
<	Inférieur à	price<4.35
<=	Inférieur ou égal à	price<=4.35
>	Supérieur à	price>4.35
>=	Supérieur ou égal à	price>=4.35
or	ou	price>3.00 or lang="en"
and	et	price>3.00 and lang="en"
not	Inverser	not lang="en"
	Concaténation de chaînes de caractères	"en"    "glish"

De même, il existe de nombreuses fonctions utiles de manipulation de chaînes de caractères disponibles dans XPath. Les fonctions peuvent être appelées avec le préfixe d'espace de nommage "fn :". Cependant, comme le préfixe par défaut de l'espace de nom est "fn :", les fonctions de chaîne de caractères n'ont généralement pas besoin du préfixe.

Table 11: Fonctions de chaînes de caractères dans XPath

Nom	Description
string(arg)	Retourne la valeur de la chaîne de caractères de l'argument
substring(str, start, len)	Retourne une sous-chaîne d'une chaîne de longueur "len" à partir de "start"
string-length(str)	Renvoie la longueur de la chaîne de caractères. S'il n'y a pas d'argument passé, retourne la longueur du nœud courant
compare(str1, str2)	Retourne -1 si str1 < str2, 0 si les chaînes sont égales, +1 si str1 > str2
concat(str1, str2, ...)	Retourne une concaténation de toutes les chaînes de caractères saisies
upper-case(str)	Convertit la chaîne de caractères en majuscules
lower-case(str)	Convertit la chaîne de caractères en minuscules
contains(str1, str2)	Retourne TRUE si str1 contient str2
starts-with(str1, str2)	Retourne TRUE si str1 commence par str2

ends-with(str1, str2)
-----------------------

Retourne TRUE si str1 se termine par str2
---

Un lien utile pour tester les expressions XPath peut être trouvé à l'adresse suivante (en anglais) :

<https://www.freeformatter.com/xpath-tester.html>

## 2.3 Localisateur CSS

Il y a des divergences d'opinions quant à savoir si CSS est un langage de programmation ou non. CSS signifie Cascading Style Sheets et sert principalement à déterminer comment les différents éléments HTML d'un ensemble de documents HTML doivent être rendus à l'écran, sur papier ou sur d'autres supports. Les feuilles de style CSS externes sont stockées dans des fichiers CSS.

HTML est un langage de balises et CSS est un langage de feuilles de style. Bien que HTML et CSS donnent à un utilisateur des outils très puissants pour l'affichage du contenu, la plupart des experts ne les considèrent pas comme de véritables langages de programmation.

Cependant, lorsqu'il est appliqué aux tests en Selenium, CSS est particulièrement utile pour trouver des éléments HTML, permettant d'automatiser les tests sur le navigateur.

CSS peut être utilisé sous trois formes différentes dans les documents HTML :

1. Une feuille de style externe : chaque page HTML doit inclure une référence au fichier de feuille de style externe à l'intérieur de l'élément **<link>** qui va dans la section **<head>**.
2. Une feuille de style interne : lorsqu'une page HTML doit avoir un style particulier ; les styles sont définis dans l'élément **<style>**, à l'intérieur de la section **<head>** du document.
3. Un style en ligne : s'applique à un élément spécifique et est ajouté directement à l'élément en tant qu'attribut.

Lorsque plusieurs styles CSS sont définis pour le même élément, la valeur de la dernière feuille de style lue sera utilisée. Par conséquent, l'ordre dans lequel un style sera utilisé est défini (en commençant par le haut) comme suit :

1. Style en ligne (en tant qu'attribut dans un élément HTML).
2. Feuilles de style externes et internes définies dans la section d'en-tête.
3. La valeur par défaut du navigateur.

Un ensemble de règles CSS se compose de sélecteur(s) et de bloc(s) de déclaration sous la forme suivante :

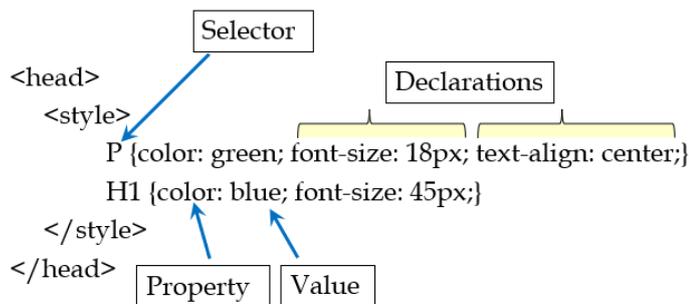


Figure 18: Règle CSS

Chaque sélecteur pointe vers un élément HTML sur lequel vous souhaitez appliquer un style. Le bloc de déclaration contient une ou plusieurs déclarations séparées par des points-virgules. Chaque déclaration comprend un nom de propriété CSS et une valeur séparée par deux points. Les blocs de déclaration sont entourés d'accolades.

Les sélecteurs CSS sont utilisés pour localiser les éléments du document HTML en fonction du nom de l'élément, de l'ID, de la classe, de l'attribut ou d'autres spécificateurs. Dans de nombreux cas, XPath peut être utilisé de la même manière pour trouver certains éléments, comme le montre le tableau ci-dessous :

Table 12: Sélecteurs CSS vs XPath

CSS	XPath	Résultats
div.even	//div[@class="even"]	Éléments div avec un attribut class="even"
#login	//*[@id="login"]	Un élément avec id="login"
*	//*	Tous les éléments
input	//input	Tous les éléments d'entrée
p,div		Tous les éléments p et tous les éléments div
div input	//div//input	Tous les éléments d'entrée à l'intérieur de tous les éléments div
div > input	//div/input	Tous les éléments d'entrée qui ont l'élément div comme parent
br + p		Sélectionne tous les éléments p qui sont placés immédiatement après l'élément br
p ~ br		Sélectionne tous les éléments p qui sont placés immédiatement avant l'élément br

Comme on pouvait le supposer, les sélecteurs CSS fonctionnent aussi avec des attributs.

Table 13: Sélecteurs CSS pour les attributs

CSS	Result
[lang]	Tous les éléments avec l'attribut lang
[lang=en]	Tous les éléments avec l'attribut lang prenant la valeur en
[lang^=en]	Tous les éléments avec l'attribut lang commençant par la chaîne de caractères en
[lang   =en]	Tous les éléments qui ont l'attribut lang égal à en ou commençant par la chaîne en suivie d'un trait d'union

[lang\$=en]	Tous les éléments qui ont l'attribut <i>lang</i> se terminant par la chaîne de caractères <i>en</i>
[lang~=en]	Tous les éléments qui ont l'attribut <i>lang</i> dont la valeur est une liste de mots séparés par des espaces, dont l'un d'entre eux est exactement la chaîne de caractères <i>en</i>
[lang*=en]	Tous les éléments qui ont l'attribut <i>lang</i> contenant la chaîne de caractères <i>en</i>

Pour les éléments de formulaire, il existe différents sélecteurs CSS :

Table 14: Sélecteurs CSS pour les éléments de formulaire

CSS	Résultat
:checked	Sélectionne tous les éléments cochés (pour les cases à cocher, les boutons radio et les options qui sont mises à l'état activé)
:default	Sélectionne n'importe quel élément de formulaire qui est par défaut parmi un groupe d'éléments liés
:defined	Sélectionne tous les éléments qui ont été définis
:disabled	Sélectionne tous les éléments qui ont été désactivés
:enabled	Sélectionne tous les éléments qui ont été activés
:focus	Sélectionne l'élément qui fait l'objet du focus
:invalid	Sélectionne les éléments de formulaire qui ne sont pas validés
:optional	Sélectionne les éléments de formulaire qui n'ont pas l'attribut 'required' activé
:out-of-range	Sélectionne tous les éléments d'entrée dont la valeur actuelle est hors de la zone de valeurs min à max
:read-only	Sélectionne les éléments qui ne sont pas modifiables par l'utilisateur
:read-write	Sélectionne les éléments qui peuvent être édités par l'utilisateur
:required	Sélectionne les éléments de formulaire ou l'attribut 'required' est activé
:valid	Sélectionne les éléments de formulaire qui sont validés avec succès
:visited	Sélectionne les liens qu'un utilisateur a déjà visités

Pour finir, il y a aussi d'autres sélecteurs CSS qui peuvent être utiles à l'automatisation des tests, notamment :

Table 15: Sélecteurs CSS Utiles pour l'automatisation

CSS	Résultat
:not(<selector>)	Sélectionne les éléments qui ne correspondent pas au sélecteur spécifié
:first-child	Sélectionne tous les éléments qui sont les premiers enfants de leurs éléments parents
:last-child	Sélectionne tous les éléments qui sont les derniers enfants de leurs éléments parents
:nth-child(<n>)	Sélectionne tous les éléments qui sont les nième enfants de leurs éléments parents

:nth-last-child(<n>)	Sélectionne tous les éléments qui sont des énièmes enfants de leurs éléments parents en comptant à partir du dernier enfant
div:nth-of-type(<n>)	Sélectionne tous les éléments qui sont énièmes enfants div de leurs éléments parents

Comprendre HTML, CSS, XML et XPath est essentiel pour réussir avec Selenium. Ce que nous avons présenté ici n'est toutefois qu'une introduction à ces formalismes.

W3C (World-Wide-Web Consortium) est une communauté internationale qui travaille avec de nombreuses organisations et le public pour définir et développer des standards Web, y compris XML et XPath.

Indépendamment de W3C, il existe un excellent ensemble de tutoriels gratuits (et en anglais) sur le site W3Schools. Ces tutoriels incluent HTML, XML, XPath, CSS, et couvrent de nombreuses autres technologies qui sont utiles lors des tests avec Selenium.

W3Schools et tous ses tutoriels sont la propriété de Refsnes Data. A4Q n'a aucune lien avec W3Schools de Refsnes Data. Le lien suivant donne accès aux tutoriels :

<https://www.w3schools.com/default.asp>

## Chapitre 3 - Utiliser Selenium WebDriver

### Termes

### Objectifs d'apprentissage pour l'utilisation de Selenium WebDriver

- STF-3.1 (K3) Utiliser des moyens appropriés pour enregistrer les logs et produire les rapports
- STF-3.2 (K3) Naviguer vers différentes URL en utilisant les fonctions WebDriver
- STF-3.3 (K3) Modifier le contexte de la fenêtre dans les navigateurs Web à l'aide des fonctions WebDriver
- STF-3.4 (K3) Réaliser des captures d'écran de pages Web à l'aide de fonctions WebDriver
- STF-3.5 (K4) Localiser des éléments de l'interface graphique avec diverses stratégies
- STF-3.6 (K3) Obtenir l'état des éléments de l'interface graphique via les fonctions WebDriver
- STF-3.7 (K3) Interagir avec les éléments de l'interface graphique à l'aide des fonctions WebDriver
- STF-3.8 (K3) Gérer les entrées utilisateur dans les navigateurs Web à l'aide des fonctions WebDriver

## 3.1 Mécanismes de logs et de reporting

Les scripts de test automatisés sont du code logiciel qui exécutent des commandes sur le SUT. Ainsi, ils simulent des opérateurs humains en train de réaliser des tests à l'aide du clavier et de la souris. La solution d'automatisation des tests doit fournir un mécanisme qui implémente la couche d'exécution des tests. Une façon d'implémenter un tel mécanisme est de coder entièrement les scripts d'automatisation de tests. De tels scripts peuvent alors être exécutés comme n'importe quel autre script, par exemple en langage Python.

Plutôt que de créer complètement de tels scripts, nous pouvons plutôt utiliser des bibliothèques de tests unitaires existantes, tels que par exemple, unittest ou pytest, qui facilitent l'exécution des tests et la gestion des résultats d'exécution. Dans ce syllabus, nous avons choisi le framework d'automatisation pytest comme bibliothèque d'exécution des tests.

Pytest est un framework d'automatisation de test qui facilite l'écriture de tests pour Python, mais aussi pour Selenium WebDriver avec Python. Pytest facilite l'écriture de tests simples, mais permet aussi de réaliser des tests automatisés plus complexes.

Lorsqu'il est appelé, Pytest exécute tous les tests dans le répertoire courant ou ses sous-répertoires. Il recherche spécifiquement tous les fichiers qui répondent aux patterns : "test\_\*.py" ou " \*\_\_test.py" et les exécute ensuite.

Lorsqu'il est exécuté sans option (**pytest**), pytest exécute tous les tests dans le répertoire courant et dans ses sous-répertoires. Il est également possible de définir les options suivantes lors de l'appel de la commande (**pytest**) :

- (**pytest -v**) Mode verbeux qui affiche les noms complets des tests plutôt qu'un point.
- (**pytest -q**) Mode silencieux qui affiche moins d'information en sortie.
- (**pytest --html=report.html**) Exécute le(s) test(s) avec un rapport dans un fichier HTML.

Les tests peuvent être annotés pour être traités d'une manière particulière. Par exemple, lorsque vous mettez (**@pytest.mark.skip**) avant une définition de test, pytest ne lancera pas le test. Si vous mettez (**@pytest.mark.xfail**) avant une définition de test, il informe le moteur d'exécution que le test devrait échouer. Ces tests, s'ils échouent lors de leur exécution, ne seront pas notifiés de la même manière que les tests qui échouent lorsqu'on s'attend à ce qu'ils réussissent.

Lorsqu'un testeur manuel exécute un cas de test et que ce cas de test échoue, le testeur a une assez bonne idée de ce qui s'est passé. Il connaît les étapes qui ont mené à l'échec, ce qui s'est passé lors de l'échec, quelles données ont été utilisées et quel est le résultat obtenu. Lors de l'exécution d'un test exploratoire, bien que le testeur n'ait pas de cas de test documenté prédéfini, il a une vision générale de l'exécution et la possibilité dans une certaine mesure de revenir en arrière et établir pourquoi un échec est survenu.

Dans le contexte de l'automatisation, pratiquement rien de cela n'est vrai.

Souvent, dans l'automatisation, le message d'erreur enregistré par l'outil est tout à fait insuffisant pour que la personne qui examine une exécution qui a échoué puisse comprendre ce qui est arrivé. Par exemple, le message d'erreur n'a pas de contexte et l'erreur enregistrée peut n'avoir rien à voir avec le dysfonctionnement réel qui a arrêté l'exécution du test.

Par exemple, supposons qu'une défaillance se produise à l'étape **N**. Selon la façon dont la défaillance s'est produite, le résultat peut signifier que l'interface du SUT ne se trouve pas dans l'état correct. Lorsque le script tente d'exécuter l'étape **N+1**, l'étape échoue parce que le SUT n'est pas dans l'état correct pour exécuter cette étape. Le rapport de l'exécution du test (le fichier de logs) indique alors que c'est l'étape **N+1** qui a échoué.

Dans ce cas, le diagnostic du problème commence mal. De plus, si l'analyste de test qui a exécuté le test automatisé (généralement en mode batch) n'a pas une connaissance détaillée du script qui a été exécuté, essayer de diagnostiquer l'échec pour déterminer si c'est une défaillance réelle du SUT ou une défaillance d'automatisation peut prendre un temps conséquent.

Il n'est pas nécessaire qu'il en soit ainsi dans le domaine de l'automatisation. L'enregistrement de l'exécution des tests automatisés, correctement réalisé, peut grandement aider un analyste de test à déterminer rapidement si la défaillance a été causée par le SUT ou non. Il est primordial qu'un automaticien de tests soit attentif à la précision des logs. Une bonne gestion des logs peut faire la différence entre un projet d'automatisation qui échoue et un projet qui réussit à apporter de la valeur lorsqu'elle est réalisée.

L'enregistrement des logs est un moyen de suivre les événements qui se produisent pendant l'exécution. L'automaticien de tests peut intégrer des instructions de log à un script pour enregistrer des informations afin de faciliter la compréhension de l'exécution. Cet enregistrement peut être effectué avant et après une étape et peut inclure les données qui ont été utilisées dans l'étape et le comportement qui s'est produit à la suite de l'étape. Plus cet enregistrement est détaillé, mieux il permettra de comprendre le résultat de l'exécution du test.

Dans certains cas, lorsque vous testez des logiciels critiques pour la sûreté ou pour le service rendu, un enregistrement détaillé des logs peut être nécessaire à des fins d'audit.

L'enregistrement des logs peut être conditionné. Ainsi, les données de logs peuvent n'être sauvegardées et restituées en tant que logs uniquement si le test est en échec ou si un rapport de diagnostic complet est nécessaire. L'enregistrement de chaque étape peut ne pas être recommandé lorsque le test se déroule comme attendu ; cependant, en cas d'échec, ces informations d'enregistrement peuvent économiser des heures de dépannage du fait de la sauvegarde, étape par étape, des données exactes qui ont été utilisées et des événements survenus pendant l'exécution.

Tous les projets d'automatisation n'ont pas besoin de logs aussi robustes. Souvent, un projet d'automatisation démarre avec un ou deux automaticiens de test qui vont créer et exécuter leurs propres scripts. Dans ce cas, l'enregistrement des logs tels que décrit ci-dessus peut être considéré comme excessif. Mais, il faut noter que lorsque de petits projets d'automatisation réussissent, il est à peu près garanti que le management en voudra plus : beaucoup plus d'automatisation. Plus il y a de succès, plus la demande est forte.

Cela signifie qu'à terme, le projet va atteindre une taille telle que la gestion des enregistrements d'exécution décrite précédemment **sera nécessaire**. Ce qui impliquera que les scripts automatisés existants devront être repris et améliorés pour des logs plus pertinents. Il est de ce fait plus efficace et plus efficient de réaliser un enregistrement des logs de manière robuste dès le début.

Python dispose d'un ensemble complet de fonctions pour l'enregistrement des logs qui peuvent être utilisées avec WebDriver. À tout moment dans un script automatisé,

l'automatisme de tests peut ajouter des appels d'enregistrement de logs pour rapporter toutes les informations souhaitées. Il peut s'agir d'informations générales pour le suivi ultérieur du test (par exemple, "Je suis sur le point de cliquer sur le bouton XYZ"), d'avertissements concernant un événement à noter même s'il ne provoque pas l'échec du test (par exemple, "Ouverture du fichier <ABC> a pris plus longtemps que prévu.") ou des erreurs réelles, qui provoquent une exception déclenchant les actions de terminaison du test et le passage au test suivant

La bibliothèque d'enregistrement des logs en Python possède cinq niveaux différents de messages qui peuvent être sauvegardés. Du plus bas au plus haut niveau :

- DEBUG: pour diagnostiquer les problèmes
- INFO: pour informer sur le déroulement du test
- WARNING: quelque chose d'inattendu s'est produit, un problème potentiel
- ERROR: un problème majeur s'est produit
- CRITICAL: un problème critique s'est produit

Lorsqu'un log est édité sur la console ou dans un fichier, le niveau de message peut être paramétré sur l'un de ces cinq niveaux (ou sur un niveau personnalisé pouvant être défini) permettant à l'utilisateur de voir uniquement les messages désirés. Par exemple, si l'édition des logs est paramétrée au niveau WARNING, l'utilisateur ne verra pas les messages DEBUG ou INFO, mais seulement les messages de logs de types WARNING, ERROR, et CRITICAL. Ainsi, supposons que le code suivant a été exécuté :

```
import logging
logging.basicConfig(level=logging.WARNING)
# default logging level is WARNING

logging.info("Hello world.")
logging.warning("Title: %d Dalmatians" % 101)
logging.debug("Title: %s" % "101 Dalmatians")
```

*Figure 19: Exemple d'enregistrement de logs*

Comme le niveau d'enregistrement par défaut est WARNING, les informations suivantes sont affichées sur la console :

```
WARNING:root: Title: 101 Dalmatians
```

Lors de l'exécution d'un cas de test, il est important de tester un comportement effectif du système. Chaque cas de test doit pour cela avoir des résultats attendus liés à des actions du SUT qui peuvent être vérifiées. Python dispose d'un mécanisme intégré pour vérifier si les données ou le comportement corrects se sont produits : l'assertion.

Une assertion est un énoncé dont on s'attend à ce qu'il soit vrai à un moment donné dans le script. Par exemple, si nous testons une calculatrice, nous pourrions ajouter deux plus deux, puis affirmer que la réponse devrait être égale à quatre. Si pour une raison quelconque le calcul est incorrect, le script lancera une exception.

La syntaxe est la suivante :

```
assert sumVariable==4, "sumVariable should equal 4."
```

Avec Selenium WebDriver, une étape d'un cas de test comprend généralement les actions suivantes.

1. Localiser un élément web affiché.
2. Agir sur cet élément web.
3. S'assurer que la situation s'est bien déroulée.

Pour la première action (1), si l'élément n'est pas trouvé, ou est trouvé dans un état incorrect, une exception est levée.

Pour l'action deux (2), si la tentative d'agir sur l'élément web échoue, une exception est levée.

Pour l'action trois (3), nous pouvons utiliser une assertion pour vérifier que le comportement attendu s'est produit, ou que la valeur attendue a été reçue.

Fondamentalement, cela suit la façon dont un testeur manuel effectuerait ces étapes et permet à la personne qui évalue un test échoué de comprendre ce qui s'est passé.

Le reporting s'appuie sur l'enregistrement des logs d'exécution, mais ils sont différents. L'enregistrement des logs d'exécution fournit des informations sur l'exécution des scripts automatisés à l'analyste de test qui a lancé la suite de test, et au(x) automaticien(s) responsable(s) de la correction des tests si nécessaire. Le reporting consiste à fournir cette information d'exécution ainsi que d'autres informations contextuelles aux divers intervenants, à l'extérieur comme à l'intérieur, qui en ont besoin ou qui les souhaitent.

Il est important que les automaticiens de test déterminent qui veut ou a besoin de rapports d'exécution des tests, et quelles informations les intéressent. Envoyer les enregistrements d'exécutions tels quels aux parties prenantes les submergerait certainement avec une masse d'informations sur l'exécution dont ils n'ont ni besoin ni envie.

Une façon de gérer les rapports est de les créer à partir des logs et d'autres informations et de les mettre à la disposition des parties prenantes pour qu'elles puissent les télécharger quand elles le souhaitent. L'autre moyen est de créer et d'envoyer les rapports dès qu'ils sont prêts aux parties prenantes qui le souhaitent.

Dans les deux cas, il faut chercher à automatiser la création et la diffusion des rapports pour supprimer une tâche manuelle à réaliser.

Les rapports doivent contenir un résumé avec un aperçu du système testé, de l'environnement ou des environnements sur lesquels les tests ont été réalisés et des résultats obtenus pendant l'exécution. Comme nous l'avons mentionné, chaque partie prenante peut vouloir avoir une vision différente des résultats et ces besoins doivent être satisfaits par l'équipe d'automatisation. Souvent, les parties prenantes peuvent vouloir voir les évolutions des résultats des tests, et pas seulement à un moment précis dans le temps. Comme chaque projet a probablement des intervenants différents ayant des besoins différents, plus les tâches de production de rapports sont automatisées, plus il sera facile d'y parvenir.

## 3.2 Naviguer dans différentes URLs

### 3.2.1 Démarrer une session d'automatisation des tests

Il existe de nombreux navigateurs différents que vous pourriez vouloir tester. Bien que la tâche de base d'un navigateur soit de permettre à l'utilisateur de visualiser et d'interagir avec diverses pages Web, il est probable que chaque navigateur ait un comportement un peu différent des autres navigateurs.

Au début d'Internet, il arrivait que certaines pages web ne fonctionnaient que dans Netscape, alors que d'autres ne fonctionnaient correctement que dans IE. Heureusement, la plupart de ces problèmes ont disparu depuis longtemps, bien qu'il y ait encore quelques incompatibilités entre les navigateurs. Lorsqu'une organisation veut mettre en place un site Web, des tests doivent donc être effectués sur différents navigateurs afin de vérifier la compatibilité.

Au chapitre 1.4, lorsque nous avons introduit Selenium WebDriver pour la première fois, nous avons mentionné que différents navigateurs nécessitaient différents pilotes pour s'assurer que les scripts automatisés que nous écrivons fonctionnent avec les différents navigateurs de la liste suivante :

- Chrome (chromedriver.exe)
- Internet Explorer (IEDriverServer.exe)
- Edge (MicrosoftWebDriver.msi)
- Firefox (geckodriver.exe)
- Safari (safaridriver)
- HtmlUnit (HtmlUnit driver)

Par exemple, si nous voulons tester le navigateur Chrome, nous devons d'abord télécharger le pilote Chrome (commodément nommé **chromedriver.exe**) et installer ce fichier à un endroit bien défini du poste de travail du testeur. Ceci est généralement réalisé en éditant la variable d'environnement **Path** pour que votre système d'exploitation la trouve quand il en a besoin.

Veuillez noter que, comme c'est souvent le cas avec les aspects techniques, ces fichiers de pilotes et les informations nécessaires à leur utilisation peuvent changer. L'information contenue dans ce syllabus a été vérifiée et est valide au moment où elle a été rédigée. Mais cela peut évoluer. La meilleure chose qu'un automaticien puisse faire est de se familiariser avec la documentation accessible sur les différents sites web de support pour les différents navigateurs, et le site de support Selenium WebDriver.

Pour travailler avec des pages Web, vous devez d'abord ouvrir un navigateur Web. Ceci peut être réalisé en créant un **objet WebDriver**. L'instanciation de l'**objet WebDriver** crée l'interface de programmation entre un script et le navigateur Web. Il exécutera également un processus WebDriver si besoin est pour un navigateur Web particulier. En général, il lancera également le navigateur Web lui-même.

```
from selenium import webdriver
driver = webdriver.Chrome()
```

Figure 20: Code pour créer un objet WebDriver

Ce code ne fonctionnera qu'après l'installation du fichier **chromedriver.exe** comme indiqué précédemment.

La création d'un objet WebDriver démarre un navigateur Web avec une page vide. Pour naviguer à la page de site désirée, la fonction **get()** doit être utilisée. Notez que la fonctionnalité de l'objet pilote est accessible en utilisant la notation "." car Python est un langage orienté objet.

```
driver.get('https://www.python.org')
```

Figure 21: Se positionner sur l'URL

Il est également possible d'attacher un objet WebDriver à un processus Webdriver existant ou de créer un processus WebDriver et de le rattacher à un navigateur déjà lancé via Selenium RemoteWebDriver, mais ces aspects ne sont pas couverts par ce syllabus.

Après avoir ouvert une page ou navigué vers une page différente, il est conseillé de vérifier si la page correcte a été ouverte. L'objet WebDriver contient deux attributs utiles pour cela : **current\_url** et **title**. La vérification des valeurs de ces champs permet au script de garder la trace de sa page courante.

```
assert driver.current_url == 'https://www.python.org/', ErrMsg
assert driver.title == 'Welcome to Python.org', ErrMsg
```

Figure 22: Assertion de la page correcte

### 3.2.2 Navigation et rafraîchissement des pages

Lorsque vous avez besoin de simuler la navigation avant et arrière dans le navigateur Web, vous devez utiliser les méthodes **back()** et **forward()** de l'objet WebDriver. Ils enverront les commandes appropriées au WebDriver. Ces méthodes n'ont pas d'arguments.

```
driver.back()
driver.forward()
```

Figure 23: Navigation dans l'historique du navigateur

Il est également possible de rafraîchir la page en cours avec le navigateur Web. Ceci peut être fait en appelant la méthode **refresh()** de WebDriver.

```
driver.refresh()
```

Figure 24: Rafraîchissement du navigateur

### 3.2.3 Fermeture du navigateur

À la fin du test, vous devez fermer le processus du navigateur Web et tous les autres processus du pilote qui ont été exécutés. Si vous ne fermez pas le navigateur, il restera ouvert même une fois le test terminé. Il est conseillé de replacer l'environnement de test dans le même état que celui dans lequel le(s) test(s) a (ont) commencé. Cela permet d'exécuter un nombre indéfini de tests sans intervention humaine.

Pour fermer le navigateur contrôlé par webdriver, appelez la méthode **quit()** de l'objet webdriver.

```
driver.quit()
```

*Figure 25: Fermeture du navigateur complet*

Vous devez placer la fonction **quit()** dans la partie du script de test qui est exécutée quel que soit le résultat du test. Une erreur courante consiste à considérer la fermeture du navigateur comme l'une des étapes de test ordinaires. Dans ce cas, lorsque le test échoue, la bibliothèque d'exécution de test arrête l'exécution des étapes de test et l'étape qui est responsable de la fermeture du navigateur n'est jamais exécutée. Généralement, les bibliothèques de test ont leurs propres mécanismes de définition et d'exécution du code de fin d'exécution du test – appelé « tear down » en anglais -, par exemple, les méthodes **tearDown()** du module Python **unittest**. Il faut vous référer à la documentation de la bibliothèque que vous utilisez pour obtenir plus de détails à ce sujet.

Si plusieurs onglets sont ouverts dans le navigateur testé, vous pouvez déterminer lequel des onglets est ouvert en vérifiant le titre de la fenêtre en cours en utilisant la commande suivante :

```
cur_win = driver.title
```

*Figure 26: Obtenir le titre de la fenêtre active*

Si vous ne voulez pas quitter tout le navigateur et seulement fermer un onglet dans le navigateur, utilisez la fonction ci-dessus et déplacez-vous dans les onglets jusqu'à ce que vous arriviez dans la fenêtre à fermer. Puis fermez l'onglet en utilisant la méthode **close()**. Lorsque le dernier onglet ouvert est fermé, le processus du navigateur se termine automatiquement.

```
driver.close()
```

*Figure 27: Fermeture de l'onglet actif*

Cette méthode ne prend aucun paramètre et ferme l'onglet actif. N'oubliez pas de changer le contexte sur l'onglet désiré pour pouvoir fermer cet onglet. Une fois la fermeture effectuée, l'appel de toute autre commande WebDriver autre que **driver.switch\_to.window()** lèvera alors une exception **NoSuchWindowException** puisque la référence objet pointe toujours sur la fenêtre qui n'existe plus. Vous devez passer de manière proactive à une autre fenêtre avant d'appeler une méthode WebDriver. Après avoir changé d'onglet, vous pouvez vérifier

le titre pour déterminer quel onglet est maintenant actif. Nous abordons ci-dessous le contrôle des onglets multiples.

### 3.3 Changer le contexte de la fenêtre

Parfois, lorsque vous testez des applications ou des scénarios plus complexes, vous devrez changer le contexte courant de l'interface graphique que vous testez. Cela peut être dû à la nécessité de vérifier le résultat du test dans un autre système ou d'exécuter une étape de test dans une autre application.

Il peut aussi arriver que l'interface graphique de l'application que vous testez soit si complexe que vous aurez besoin de passer d'un cadre ou d'une fenêtre à l'autre.

Un navigateur Web n'a pas besoin d'avoir le focus pour pouvoir exécuter des commandes Selenium WebDriver, car le protocole WebDriver est basé sur la communication HTTP. Ceci permet à WebDriver d'exécuter plusieurs tests simultanément ou de contrôler plusieurs navigateurs dans un seul script.

Changer le contexte du script peut se faire de trois façons :

- changer d'instance de navigateur
- changer de fenêtre ou d'onglet au sein d'un même navigateur
- changer de cadre dans une page

Pour ouvrir deux navigateurs, vous devez créer deux objets WebDriver. Chaque objet WebDriver contrôle un navigateur. Chaque objet WebDriver est ensuite utilisé dans le script de test conformément aux étapes du scénario de test automatisé. Les objets WebDriver peuvent contrôler le même type de navigateur Web (par exemple, Chrome, Firefox) ou des types différents (par exemple, un Chrome et l'autre Firefox). Vous pouvez également ouvrir plus de deux navigateurs si nécessaire. Rappelez-vous que chaque navigateur est contrôlé par un objet WebDriver. Et n'oubliez pas de fermer tous les navigateurs à la fin du test.

Ouvrir plusieurs onglets ou fenêtres dans un même navigateur peut se révéler compliqué, car les différents navigateurs et systèmes d'exploitation ont des méthodes différentes pour le faire. Une façon qui fonctionne pour Windows dans le navigateur Chrome est d'exécuter un appel de fonction en code JavaScript comme suit :

```
driver.execute_script("$ (window.open ( ' ' ) ) ")
```

*Figure 28: Appel JavaScript pour ouvrir un nouvel onglet*

Une présentation plus approfondie de l'ouverture de plusieurs onglets/fenêtres dans un navigateur Web n'entre pas dans le cadre de ce syllabus.

Pour basculer entre les onglets ouverts dans un navigateur, vous devez d'abord obtenir la liste de tous les onglets ouverts. Cette liste se trouve dans l'attribut **window\_handles** de l'objet WebDriver. Sachez que l'ordre des onglets dans le tableau **window\_handles** peut être différent de celui des onglets dans le navigateur.

Vous pouvez faire défiler tous les onglets/fenêtres en utilisant le code suivant :

```
for handle in driver.window_handles:
    driver.switch_to.window(handle)
```

*Figure 29: Parcourir les onglets ouverts*

La façon la plus sûre de déterminer quelle fenêtre est la fenêtre actuellement ouverte est d'utiliser l'attribut **driver.title** mentionné précédemment.

Le code Python suivant ouvre la page d'accueil de Python, ouvre un nouvel onglet, puis ouvre la page d'accueil de Perl dans le deuxième onglet, puis ramène le navigateur Web dans l'onglet contenant la page d'accueil de Python :

```
From selenium import webdriver
driver = webdriver.Chrome()
driver.get('https://python.org')
driver.execute_script("$ (window.open(' ')) ")
driver.switch_to.window(driver.window_handles[1])
driver.get('https://perl.org|')
driver.switch_to.window(driver.window_handles[0])
```

*Figure 30: Ouvrir deux onglets et passer d'un onglet à l'autre*

Veillez noter que ce code n'est pas valide pour un test de production réel car il suppose que les contrôles de l'onglet seront dans l'ordre. Il ne sert qu'à des fins de référence. Pour visualiser le déroulement de ce code, ajoutez des fonctions **sleep()** pour qu'il ne soit pas exécuté trop rapidement pour en voir le résultat à l'affichage.

La troisième situation où vous pouvez vouloir changer de contexte dans un navigateur Web est le changement de cadre ("frame" en anglais). Ceci est souvent nécessaire ; si vous ne changez pas le contexte d'un cadre particulier, vous ne serez pas en mesure de trouver des éléments dans ce cadre, et donc d'automatiser les tests pour les éléments de ce cadre.

Pour modifier le contexte d'un cadre spécifique, vous devez d'abord trouver ce cadre. Si vous connaissez l'ID du cadre, vous pouvez directement y accéder, comme suit :

```
driver.switch_to.frame('foo')
```

*Figure 31: Changer de cadre*

où **foo** est l'**ID** du cadre sur lequel vous voulez changer le contexte.

Si le cadre n'a pas d'ID ou si vous voulez utiliser une stratégie différente pour le trouver, le changement de contexte peut se faire en deux étapes. La première étape, comme mentionnée plus haut, est de trouver le cadre comme un élément web, et la deuxième étape est de passer au cadre trouvé. La bascule dans ce cas est exécutée par la même méthode que la bascule par identifiant, mais l'argument donné à une fonction est l'élément web trouvé.

Voici une section du code Python qui montre un exemple de basculement de cadre dans lequel trouver le cadre est la première étape :

```
frm_message = driver.find_element_by_name('message')
driver.switch_to_frame(frm_message)
```

*Figure 32: Trouver puis changer de cadre*

Notez que lors de l'appel à **switch\_to.frame()**, nous n'utilisons pas d'apostrophes ni de guillemets, car nous passons la **variable frm\_message** en argument variable plutôt qu'une chaîne contenant l'ID du cadre.

Si vous voulez revenir au cadre parent, utilisez la commande suivante :

```
driver.switch_to.parent_frame()
```

*Figure 33: Basculer vers le parent du cadre*

Vous pouvez aussi revenir à la page entière en utilisant :

```
driver.switch_to.default_content()
```

*Figure 34: Passage à la fenêtre principale*

Outre la modification du contexte d'une fenêtre ou d'un cadre du navigateur Web, le framework Selenium WebDriver vous permet de manipuler la taille de la fenêtre du navigateur Web. Le terme fenêtre est ici utilisé comme désignant toute la fenêtre vue du système d'exploitation, plutôt qu'un simple onglet dans un navigateur Web.

Selenium permet de minimiser et de maximiser les fenêtres du navigateur Web et de le mettre en mode plein écran. Les raccourcis Python pour cette fonction sont les suivants :

```
maximize: driver.maximize_window()
minimize: driver.minimize_window()
fullscreen: driver.fullscreen_window()
```

*Figure 35: Dimensionnement du navigateur*

Ces fonctions ne prennent aucun argument, car elles fonctionnent sur le navigateur web contrôlé par l'objet **driver**.

## 3.4 Capturer des captures d'écran de pages Web

Lorsqu'un testeur manuel exécute un cas de test, il interagit visuellement avec les objets de l'interface graphique à l'écran. En cas d'échec d'un cas de test, les testeurs manuels peuvent le constater car ce qu'ils voient à l'écran n'est pas correct.

Un testeur utilisant l'automatisation n'a pas cette possibilité.

Les scripts d'automatisation des tests ne sont pas en mesure de vérifier de manière fiable la mise en page et l'apparence des pages Web. Il est souvent utile de prendre des captures d'écran d'une page ou d'un élément particulier de l'écran et de les enregistrer dans les logs ou à un emplacement connu, afin de pouvoir les visualiser ultérieurement, dans les cas suivants :

- Lorsque le script automatisé détecte qu'une défaillance s'est produite.
- Lorsque le test est très visuel et que la détermination de la réussite ou de l'échec ne peut se faire qu'en visualisant l'image de l'écran.
- Lorsqu'il s'agit de logiciels critiques pour la sûreté ou la mission, qui pourraient nécessiter un audit des tests.
- Lors des tests de configuration sur différents systèmes.

Pour tirer profit des informations fournies par les captures d'écran, elles doivent être prises au bon moment. Généralement, les parties des scripts de test qui font des captures d'écran sont placées juste après les étapes de test qui contrôlent l'interface utilisateur ou dans les fonctions de fin de test (étape « tear down »). Mais comme ces captures sont un outil précieux pour comprendre les résultats des tests automatisés, ces captures peuvent être réalisées à tout endroit jugé utile du script.

Une question importante à traiter est de nommer les fichiers avec des noms et des emplacements uniques afin que votre automatisation n'écrase pas les captures d'écran effectuées plus tôt dans l'exécution. Il y a une variété de façons différentes de le faire ; mais cela dépasse le cadre de ce syllabus.

Les captures d'écran peuvent être prises selon deux portées différentes : la page entière du navigateur, ou un seul élément dans la page du navigateur<sup>1</sup>. Les deux méthodes utilisent le même appel de fonction mais sont appelées à partir de contextes différents.

L'appel de méthode à utiliser est **screenshot()**. L'exemple suivant en Python montre comment faire une capture d'écran d'une page entière et la placer à un endroit spécifique :

```
driver.get_screenshot_as_file('C:\\temp\\screenshot.png')
```

*Figure 36: Enregistrement d'une capture d'écran d'une page entière*

Cet exemple montre comment faire une capture d'écran d'un élément et l'enregistrer à un emplacement spécifique :

---

<sup>1</sup> Voir la documentation de Selenium WebDriver Python library version 3.13.0 qui indique que la fonctionnalité permettant de faire une capture d'écran d'un élément WebElement est disponible en appelant la fonction : **WebElement.screenshot('Filename.png')**. Les auteurs de ce syllabus n'ont pas pu vérifier que cette fonctionnalité fonctionne dans le navigateur Chrome 67 avec chromedriver 2.36, bien qu'elle soit définie dans la recommandation technique WebDriver W3C. Si vous avez besoin de cette fonctionnalité, il existe des solutions de contournement documentées sur plusieurs sites Web. Essayez une recherche Google avec " Python WebDriver screenshot of WebElement ". Cette fonction semble fonctionner lorsque vous testez avec le navigateur Firefox.

```
ele = driver.find_element_by_id('btnLogin')
ele.screenshot('c:\\temp\\element_screenshot.png')
```

*Figure 37: Enregistrer la capture d'écran de l'élément*

La réalisation d'une capture d'écran dans le navigateur peut prendre un certain temps car une grande partie du traitement doit être effectuée par le poste de travail. Si l'état de l'interface graphique change rapidement (par exemple, en exécutant simultanément des bibliothèques AJAX), la capture d'écran prise peut ne pas montrer l'état exact de la page ou l'élément attendu. De nouveau, il existe des solutions à cette situation, mais elles dépassent le cadre de ce syllabus.

Si vous voulez prendre une capture d'écran et la traiter autrement que comme un fichier, il existe des alternatives dans WebDriver. Par exemple, supposons qu'au lieu d'utiliser un fichier HTML pour un log, vous souhaitez vous connecter à une base de données à la place. Lorsque vous prenez une capture d'écran, ou d'un élément, vous ne voulez pas créer un fichier **\*.png** avec cette capture, mais vous voudrez peut-être la lire directement dans un enregistrement de base de données. Les appels suivants créeraient une image en tant que chaîne de caractères codée base64 de la capture d'écran. La version codée base64 est beaucoup plus sûre à lire en continu qu'un fichier binaire, tel qu'un fichier **\*.png**. Le premier appel ci-dessous capturera la totalité de la fenêtre, le second un seul élément :

```
img_b64 = driver.get_screenshot_as_base64()
img_b64 = element.screenshot_as_base64
```

*Figure 38: Création d'une chaîne encodée en base64 à partir d'une image*

De même, si vous voulez obtenir une chaîne binaire représentant une image, sans l'enregistrer dans un fichier, vous pouvez utiliser les appels suivants pour retourner la représentation binaire d'un fichier **\*.png**. Le premier appel renvoie l'image d'un écran entier, le second renvoie l'image d'un seul élément :

```
png_str = driver.get_screenshot_as_png()
png_str = element.screenshot_as_png()
```

*Figure 39: Création d'une chaîne binaire à partir d'une image*

## 3.5 Localiser les éléments de l'interface graphique

### 3.5.1 Introduction

Pour effectuer la plupart des opérations avec WebDriver, vous devez localiser les éléments de l'interface utilisateur sur lesquels vous voulez opérer dans l'écran actuellement actif. Ceci peut être réalisé en utilisant les méthodes **find\_element\_** ou **find\_elements\_** de WebDriver. Ces deux méthodes ont des actions différentes selon l'objet de la recherche. Pour la recherche, par exemple, les éléments suivants peuvent être utilisés :

- **by\_id (id\_)**
- **by\_class\_name (name)**
- **by\_tag\_name (name)**
- **by\_xpath (xpath)**

- **by\_css\_selector (css\_selector)**

Dans chaque cas, l'argument pris par la fonction est une chaîne représentant le ou les éléments que nous recherchons. Les valeurs de retour sont différentes ; par exemple, la version au singulier (**find\_element\_**) retournera un seul des éléments Web (si un est trouvé) et la version multiple (**find\_elements\_**) retournera une liste des éléments Web qui correspondent à cet argument.

Pour ce faire, nous devons introduire un concept utilisé dans le développement et les tests web : le DOM (Document Object Model). Lorsqu'une page Web est chargée dans le navigateur, le navigateur crée un DOM, modélisant la page Web comme une arborescence d'objets. Ce DOM définit un standard pour accéder à la page web. Comme défini par le W3C :

*" Le Document Object Model (DOM) du W3C est une interface neutre en termes de plate-forme et de langage qui permet aux programmes et aux scripts d'accéder dynamiquement au contenu, à la structure et au style d'un document et de les actualiser."*

Le DOM définit :

- Tous les éléments HTML en tant qu'objets.
- Les propriétés de tous les éléments HTML.
- Les méthodes qui peuvent être utilisées pour accéder à tous les éléments HTML.
- Les événements qui affectent tous les éléments HTML.

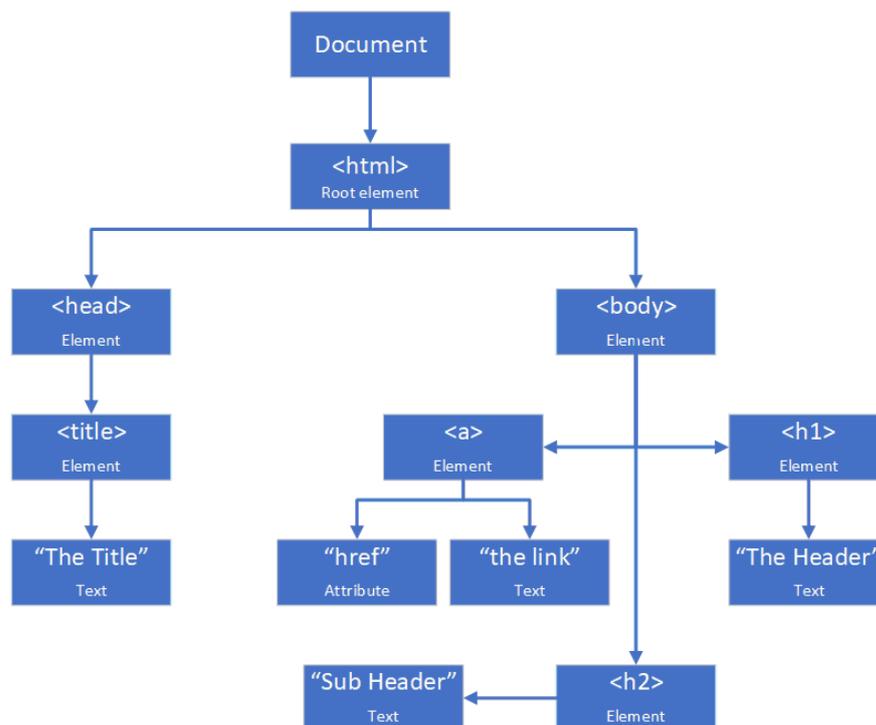


Figure 40: Arborescence DOM

### 3.5.2 HTML Methods

Les méthodes suivantes dépendent de la recherche d'éléments de l'écran en recherchant des artefacts HTML dans le document. La première méthode que nous allons discuter est l'utilisation de l'attribut HTML **ID**. Pour chaque façon différente de localiser un élément Web, il y a des avantages et des inconvénients possibles.

```
<element id="unique_id">
```

*Figure 41: Extrait HTML*

Exemple de localisation Python WebDriver :

```
element_found = driver.find_element_by_id('unique_id')
```

*Figure 42: Localisation par ID*

Avantages :

- Façon performante de réaliser l'opération.
- Par définition, chaque ID doit être unique dans le document HTML.
- Un testeur peut facilement ajouter des ID au SUT (note : ces changements doivent être revus et vérifiés au moins sous la forme d'une revue informelle).

Inconvénients :

- Les IDs peuvent être générés automatiquement, ce qui implique qu'ils peuvent être modifiés dynamiquement.
- Les ID ne sont pas appropriés pour le code qui est utilisé à plusieurs endroits, par exemple, un template utilisé pour générer des en-têtes et des pieds de page pour différents modes de dialogue.
- Il se peut qu'un testeur ne soit pas autorisé à modifier le code du SUT.

La deuxième façon de trouver un élément web est de le trouver en recherchant son nom de classe. Il s'agit de l'attribut **class** HTML d'un élément DOM, par exemple, dans cet extrait HTML :

```
<element class="class-name1">
```

*Figure 43: Extrait HTML*

Exemple de localisation WebDriver en Python :

```
element_found = driver.find_element_by_class_name('class-name1')
```

*Figure 44: Localiser par classe*

Avantages :

- Les noms de classes peuvent être utilisés à plusieurs endroits dans le DOM, mais vous pouvez limiter la localisation à la page chargée (par exemple dans une fenêtre popup modale).
- Un testeur peut facilement ajouter des noms de classes au SUT (note : ces changements doivent être revus et vérifiés au moins sous la forme d'une revue informelle).

Inconvénients :

- Comme les noms de classe peuvent être utilisés à plusieurs endroits, il faut faire plus attention de ne pas localiser le mauvais élément.
- Il se peut qu'un testeur ne soit pas autorisé à modifier le code du SUT.

La troisième façon de réaliser la localisation d'un élément web est d'utiliser le nom de balise HTML de l'élément. Il s'agit du nom de la balise DOM d'un élément, par exemple, dans cet extrait HTML :

```
<h2>
```

*Figure 45: Tag HTML*

Exemple de localisation Python WebDriver :

```
heading2_found = driver.find_element_by_tag_name('h2')
```

*Figure 46: Localisation par Tag*

Avantage : Si une balise est unique à une page, vous pouvez restreindre l'endroit où chercher.

Inconvénient : Si une balise n'est pas unique à une page, vous pouvez trouver le mauvais élément.

Notre quatrième façon d'identifier l'élément web est d'utiliser le texte du lien. Il s'agit uniquement d'une balise a ancre qui contient du texte qui sera surligné pour qu'un utilisateur puisse cliquer dessus. Il existe en fait deux méthodes similaires : vous pouvez spécifier la chaîne de texte entière ou une partie de la chaîne. Par exemple, dans l'extrait HTML suivant, le texte du lien a été mis en gras pour l'identifier :

```
<html>
  <body>
    <p class="paragraph">XyzAbc.</p>
    <a href="next.html">Next Page</a>
  </body>
</html>
```

*Figure 47: Extrait HTML*

Exemples de localisation Python WebDriver :

```
element = driver.find_element_by_link_text('Next Page')
```

*Figure 48: Localisation par texte de lien*

```
element = driver.find_element_by_partial_link_text('Next Pa')
```

*Figure 49: Localisation par texte de lien partiel*

Avantages :

- Si le texte du lien est unique à une page, vous pouvez trouver l'élément.
- Le texte du lien est visible par l'utilisateur (dans la plupart des cas), il est donc facile de savoir ce que le code de test recherche.
- Un texte de lien partiel est un peu moins susceptible d'être modifié que le texte intégral du lien.

Inconvénients :

- Le texte du lien est plus susceptible de changer qu'un ID ou un nom de classe.
- L'utilisation d'un texte partiel peut rendre plus difficile l'identification unique d'un lien unique.

### 3.5.3 Méthodes XPath

Comme indiqué dans la section 2.2, XPath (XML Path Language) est un langage qui peut être utilisé pour sélectionner des nœuds spécifiques à l'aide de différents critères dans un document XML. Par exemple, prenons le fragment HTML suivant. Comme HTML est un sous-ensemble de XML, il peut être parcouru à l'aide de XPath. À noter que des erreurs peuvent apparaître si le HTML est mal formé (p. ex. si toutes les balises de fermeture ne sont pas incluses).

```
<html>
  <body>
    <form id="sample_form1">
      <input name="text1" type="text" />
      <input name="text2" type="text" />
      <input name="submit_button" type="submit" value="Enter" />
    </form>
  </body>
</html>
```

*Figure 50: Extrait HTML*

WebDriver peut utiliser XPath pour trouver un nœud spécifique, et à partir de là, un élément peut être localisé. Vous pouvez spécifier un chemin absolu, mais c'est une mauvaise pratique, car tout changement invaliderait probablement votre code. Une meilleure façon est d'identifier un chemin relatif à partir d'un nœud trouvé qui correspond à un critère. Ci-dessous se trouve un exemple de spécification d'un chemin absolu, puis d'une version plus

robuste trouvant l'élément souhaité via XPath. Les deux renverront le deuxième champ d'entrée dans l'extrait HTML.

```
element = driver.find_element_by_xpath('/html/body/form[2]')
```

Figure 51: Utilisation de XPath avec chemin absolu

```
element = driver.find_element_by_xpath("//form[@id='sample_form1']/input[2]")
```

Figure 52: Utilisation de XPath avec chemin relatif

L'utilisation du chemin relatif demandera souvent une expression XPath plus longue, mais il sera beaucoup plus robuste pour trouver le nœud désiré. C'est un bon compromis.

Dans une chaîne XPath, vous pouvez rechercher **un identifiant, un nom, une classe, un nom de balise**, etc. Ainsi, il est possible de créer des fonctions de localisation génériques en utilisant XPath en passant un type d'attribut à la fonction, ou une chaîne de chemin qui incorpore l'attribut dans celle-ci. Par exemple :

```
def find_by_xpath(driver, path_string):
    element = driver.find_element_by_xpath('path_string')
    return element
```

Figure 53: : Fonction de recherche générique XPath

La variable **path\_string** peut alors être définie en fonction de l'attribut que nous cherchons à localiser. Ci-dessous deux exemples, la première recherche par **identifiant**, la seconde recherche par **classe** :

```
path_string = "//*[@id = '%s']" % 'id_to_find'
path_string = "//*[@class = '%s']" % 'class_to_find'
```

Figure 54: Construire une chaîne XPath

L'appel de la fonction générique s'effectuerait alors de la manière suivante :

```
element_found = find_by_xpath(driver, path_string)
```

Figure 55: Appel de la fonction XPath générique

Cette méthode n'est pas élégante, mais elle montre la flexibilité de XPath pour localiser des éléments.

Avantages :

- Vous pouvez trouver des éléments qui n'ont pas d'attributs uniques (id, classe, nom, etc.).
- Vous pouvez utiliser XPath dans les localisateurs génériques, en utilisant les différentes stratégies "By" (par id, par classe, etc.) si nécessaire.

Inconvénients :

- Le code XPath absolu est " fragile " et peut ne pas fonctionner après un petit changement dans la structure HTML.
- Le code XPath relatif peut trouver le mauvais nœud si l'attribut ou l'élément que vous recherchez n'est pas unique sur la page.
- Comme XPath peut être implémenté différemment d'un navigateur à l'autre, des efforts supplémentaires peuvent être nécessaires pour exécuter des tests WebDriver dans chaque environnement.

### 3.5.4 Méthodes de sélection CSS

Comme nous l'avons vu dans la section 2.3, nous pouvons également trouver des éléments à l'aide de sélecteurs CSS. Par exemple, dans cet extrait HTML :

```
<html>
  <body>
    <p class="paragraph">Some Latin nonsense.</p>
  </body>
</html>
```

*Figure 56: Extrait HTML*

En utilisant les règles des Sélecteurs CSS, décrites à la section 2.3, le code suivant identifiera le nœud souhaité :

```
element = driver.find_element_by_css_selector('p.paragraph')
```

*Figure 57: Identification d'un élément WebElement via le sélecteur CSS*

Avantage: Si un élément est unique à une page, vous pouvez restreindre l'endroit où vous cherchez.

Inconvénient : Si un élément n'est pas unique à une page, vous pouvez trouver le mauvais élément.

### 3.5.5 Localisation via des conditions prédéfinies

Selenium avec Python intègre a un module appelé **expected\_conditions** qui peut être importé à partir de **selenium.webdriver.support** avec des conditions prédéfinies. Vous pouvez créer des classes de conditions personnalisées, mais les classes prédéfinies devraient satisfaire la plupart de vos besoins. Ces classes offrent une plus grande spécificité que les localisateurs mentionnés ci-dessus. En d'autres termes, ils ne se contentent pas de déterminer si un élément existe, ils permettent aussi de vérifier les états spécifiques dans lesquels cet élément se trouve. Par exemple, la fonction **element\_to\_be\_selected()** détermine non seulement que l'élément existe, mais elle vérifie également s'il est dans un état sélectionné.

Cette liste n'est pas exhaustive, mais en donne quelques exemples :

- `alert_is_present`
- `element_selection_state_to_be(element, is_selected)`
- `element_to_be_clickable(locator)`
- `element_to_be_selected(element)`
- `frame_to_be_available_and_switch_to_it(locator)`
- `invisibility_of_element_located(locator)`
- `presence_of_element_located(locator)`
- `text_to_be_present_in_element(locator, text_)`
- `title_is(title)`
- `visibility_of_element_located(locator)`

Nous en discutons plus en détail à la section 4.2, puisque plusieurs d'entre eux sont également utilisés comme mécanismes d'attente.

### 3.6 Obtenir l'état des éléments de l'interface graphique

Obtenir simplement l'emplacement d'un élément web spécifique n'est souvent pas suffisant. Pour l'automatisation des tests, il peut y avoir plusieurs raisons différentes pour lesquelles nous avons également besoin d'accéder à certaines informations sur un élément. L'information peut inclure sa visibilité actuelle, si elle est activée ou non, ou si elle est sélectionnée ou non. Les raisons en sont notamment les suivantes :

- S'assurer que l'état est tel que prévu à un moment donné du test.
- S'assurer qu'un contrôle est dans un état tel qu'il puisse être manipulé selon les besoins dans le cas de test (c.-à-d., activé).
- S'assurer qu'après que le contrôle a été manipulé, il est maintenant dans l'état prévu.
- S'assurer que les résultats attendus sont corrects après l'exécution d'un test.

Différents éléments web ont différents moyens d'accéder aux informations qu'ils contiennent. Par exemple, de nombreux éléments web ont une propriété texte qui peut être récupérée en utilisant le code suivant :

```
element_text = Element.text
```

Figure 58: Récupération d'une propriété texte

Tous les éléments web n'ont pas de propriété texte. Par exemple, si nous examinons un nœud d'en-tête (éventuellement en utilisant une méthode `find_by_css_selector("h1")`), nous nous attendons à ce qu'il ait une propriété texte. D'autres éléments web peuvent ne pas avoir la même propriété. Le contexte d'un élément web et la façon dont il est utilisé peuvent vous aider à comprendre les propriétés qu'il est susceptible d'avoir.

Certaines propriétés d'un élément web doivent être accédées à l'aide d'une méthode de la classe `WebElement`. Par exemple, supposons que vous souhaitez déterminer si un élément web particulier est actuellement activé ou désactivé. Vous pouvez appeler la méthode suivante pour obtenir cette valeur booléenne :

```
cur_state = element.is_enabled()
```

Figure 59: Vérifier si WebElement est activé

Le tableau suivant n'est pas exhaustif, mais il énumère un ensemble de propriétés communes et de méthodes d'accès dont vous pouvez avoir besoin pour l'automatisation.

Table 16: Propriétés communes et méthodes d'accès

Propriété/Méthode	Arguments	Retours	Description
<b>get_attribute()</b>	propriété à récupérer	propriété, attribut ou None	Obtient la propriété. Si aucune propriété de ce nom, récupère l'attribut du nom. Si ni l'un ni l'autre, renvoie None
<b>get_property()</b>	propriété à récupérer	propriété	Obtient la propriété.
<b>is_displayed()</b>		booléen	Retourne true si l'élément est visible par l'utilisateur
<b>is_enabled()</b>		booléen	Retourne true si l'élément est activé
<b>is_selected()</b>		booléen	Renvoie vrai si la case à cocher ou le bouton radio est sélectionné
<b>location</b>		Localisation X,Y	Retourne l'emplacement X,Y sur le canevas de rendu
<b>size</b>		Hauteur, Largeur	Renvoie la hauteur et la largeur de l'élément
<b>tag_name</b>		La propriété tag_name	Renvoie le tag_name de l'élément
<b>text</b>		Le texte de l'élément	Renvoie le texte associé à l'élément

## 3.7 Interagir avec les éléments de l'interface utilisateur à l'aide des commandes WebDriver

### 3.7.1 Introduction

Vous avez donc localisé l'élément web avec lequel vous voulez que votre test automatisé interagisse (voir la section 3.5). Vous vous êtes assuré que l'élément est dans l'état correct, de sorte que vous pouvez le manipuler selon votre cas de test (discuté dans la section 3.6). À présent, il convient de réaliser l'action voulue dans le script de test automatisé.

L'une des raisons principales pour lesquelles les interfaces utilisateur graphiques sont devenues courantes est qu'il existe un ensemble limité de commandes qui peuvent être manipulées ; chaque commande est essentiellement un contrôle à l'écran qui peut être facilement manipulé en utilisant le clavier et/ou la souris. Chaque contrôle est conçu pour être compris de façon intuitive, même par les utilisateurs débutants. Ainsi, je peux taper dans un champ texte, cliquer sur un bouton radio, sélectionner un élément dans une zone de liste, etc.

L'automatisation de la manipulation de ces contrôles peut toutefois être plus difficile à comprendre. Ce qu'un testeur réalise lors de l'exécution manuelle des tests est souvent fait inconsciemment. En tant qu'automaticiens, nous devons nous assurer que nous comprenons toutes les subtilités des modifications qui sont faites aux contrôles à l'écran.

Dans les prochaines sections, nous discuterons de la manipulation des éléments suivants :

- Champs texte
- Éléments web cliquables (champs sur lesquels vous pouvez cliquer)
- Cases à cocher
- Listes déroulantes

Pour tout élément web que vous souhaitez manipuler, vous avez plusieurs aspects à vérifier :

- Si l'élément web existe
- Si l'élément web est affiché
- Si l'élément web est activé

Selon le site Web, la façon dont le code HTML a été écrit, si Ajax est utilisé et d'autres conditions, il peut y avoir une disparité quant à la nécessité d'afficher ou non l'élément Web que vous voulez traiter pour que vous puissiez le manipuler. Par exemple, si l'élément web est sur la page active et qu'il est activé, mais qu'il a été placé hors de la vue, essayer de le manipuler peut ne pas fonctionner. Des tests effectués avec Chrome ont montré que, sur certaines pages, travailler avec un élément web non visible actuellement fonctionne et sur d'autres navigateurs, cela lève une exception. Notre conseil est d'essayer d'afficher à l'écran tous les éléments web que vous utilisez (affichés).

Puisqu'un écran de navigateur peut être modifié dynamiquement (via Ajax, par exemple) ou mis à jour sur la base d'actions précédentes, ces vérifications peuvent être effectuées en utilisant les classes **expected\_condition** qui nous permettent de synchroniser les temporisations ; nous traitons cela dans le chapitre 4.

### 3.7.2 Manipulation des champs de texte

Lorsque vous saisissez dans un champ de texte modifiable, vous voudrez généralement d'abord effacer le texte de l'élément, puis saisir la chaîne souhaitée dans le champ. Nous supposons que vous avez déjà vérifié que l'élément existe, est affiché et est activé. Si vous ne vous assurez pas de l'état du contrôle, et qu'un ou plusieurs d'entre eux sont faux, votre tentative de manipulation de l'élément peut causer une exception.

Nous considérerons que vous avez déjà localisé le contrôle d'entrée, et qu'il se trouve dans la variable nommée **element**.

```
# First clear the control
element.clear()

# Now type into the control
string_to_type = 'XYZ'
element.send_keys(string_to_type)
```

Figure 60: Saisir dans un champ d'édition

### 3.7.3 Cliquez sur des éléments web

Cliquer sur un élément simule un clic de souris. Ceci peut être fait sur un bouton radio, un lien ou une image ; en fait, tout endroit sur lequel vous pouvez cliquer manuellement avec votre souris. Nous n'incluons pas les cases à cocher ici ; elles seront abordées dans la prochaine section. Il est important de vérifier que l'élément web est réellement cliquable. Vous pouvez utiliser la méthode **element\_to\_be\_clickable** de la classe **expected\_condition** pour attendre qu'il devienne cliquable.

Nous supposons de nouveau que l'élément web a été localisé. Nous devons éventuellement attendre pour nous assurer qu'il est prêt à être cliqué, en utilisant une méthode de la classe **expected\_condition** :

```
Driver.support.expected_conditions.element_to_be_clickable(locator)
```

Figure 61: Méthode de synchronisation

La synchronisation sera abordée au chapitre 4. En supposant que l'élément web a été référencé dans la variable **element** et qu'il est cliquable, on peut appeler alors :

```
element.click()
```

Figure 62: Cliquer sur un élément web

Si l'élément web est un lien ou un bouton, on doit s'attendre à un changement de contexte qui peut être vérifié à l'écran. S'il s'agit d'une case à cocher, le résultat peut être sélectionné ou non sélectionné : ceci sera discuté dans la section suivante. S'il s'agit d'un bouton radio sur lequel on clique, vous pouvez vérifier que le bouton est bien sélectionné en appelant :

```
element.is_selected()
```

Figure 63: Vérifier l'état d'un élément web

### 3.7.4 Manipulation des cases à cocher

Si nous cliquons sur les cases à cocher, elles doivent être traitées différemment des autres contrôles cliquables. Si vous cliquez plusieurs fois sur un bouton radio, il reste sélectionné. Cependant, chaque fois que vous cliquez sur une case à cocher, elle fait basculer l'état *sélectionné*. Cliquez une fois, désormais la case est çà l'état *sélectionné*. Cliquez à nouveau, la case redevient *non sélectionnée*. Cliquez à nouveau et vous revenez à l'option *sélectionnée*.

De ce fait, il est important de maîtriser l'état que l'on veut atteindre lors de la manipulation d'une case à cocher. Nous pouvons écrire une fonction qui utilisera la case à cocher et l'état désiré et effectuera l'action correcte quel que soit l'état actuel de la case à cocher.

Supposons que la case à cocher a été chargée dans la variable **checkbox** et qu'une variable booléenne **want\_checked** soit passée pour déterminer l'état final que nous voulons.

```
# if we want it selected and it is not, then click on it
# if we want it deselected and it is selected, click on it
def set_check_box(element, want_checked):
    if want_checked and not element.is_selected():
        element.click()
    elif element.is_selected() and not want_checked:
        element.click()
```

Figure 64: Fonction de sélection de la case à cocher

```
set_checkbox (checkbox, True)
```

Figure 65: Appel de la fonction

### 3.7.5 Manipulation des menus déroulants

Les menus déroulants (contrôles de sélection) sont utilisés par de nombreux sites Web pour permettre aux utilisateurs de sélectionner une des options proposées. Certains de ces menus déroulants permettent de sélectionner simultanément plusieurs éléments de la liste.

Il existe de nombreuses façons de travailler avec la liste d'un champ de sélection. Il s'agit notamment d'options de sélection d'éléments simples ou multiples. Il existe également différentes façons de désélectionner des éléments.

Les options de sélection sont les suivantes :

- Cherchez dans le code HTML pour trouver un élément désiré et cliquez dessus.
- Sélectionner par une valeur (**select\_by\_value(value)**)
- Sélectionner tous les éléments qui affichent le texte correspondant (**select\_by\_visible\_text(text)**)
- Sélectionner un élément par son index (**select\_by\_index(index)**)

Les options de désélection sont les suivantes :

- Désélectionner tous les éléments (**deselect\_all()**)
- Désélectionner par index (**deselect\_by\_index(index)**)
- Désélectionner par valeur (**deselect\_by\_value(value)**)
- Désélectionner en fonction du texte visible (**deselect\_by\_visible\_text(text)**)

Une fois que vous avez fini de sélectionner/désélectionner les éléments dans la liste déroulante, il y a plusieurs options pour vous permettre de voir ce qui a été sélectionné :

- Retourner la liste de tous les éléments sélectionnés (***all\_selected\_options***)
- Retourner le premier élément sélectionné (ou un seul dans le cas d'un seul élément sélectionné) (***first\_selected\_option***)
- Voir toutes les options de la liste (***options***)

Puisque cliquer sur les éléments de la liste est une opération courante que nous devons faire, nous pouvons construire une fonction comme nous l'avons fait avec les cases à cocher. Dans ce cas, nous devons d'abord cliquer sur le bouton déroulant pour afficher la liste complète. Ensuite, une fois la liste ouverte, trouvez l'option désirée et cliquez dessus. Voici une fonction qui permet de réaliser cela.

```
def click_dropdown_option_by_id_and_id(driver, dropdown_id, option_id):
    dropdown_element = driver.find_element_by_id('dropdown_id')
    dropdown_element.click()
    option_element = driver.find_element_by_id('option_id')
    option_element.click()
```

*Figure 66: Fonction pour cliquer sur l'élément de la liste déroulante*

### 3.7.6 Travailler avec les boîtes de dialogue modal

Une boîte de dialogue modal s'ouvre au-dessus d'une fenêtre de navigateur et n'autorise pas l'accès à la fenêtre sous-jacente tant qu'elle n'a pas été traitée. Ces boîtes de dialogue sont similaires aux invites utilisateur, mais suffisamment différentes pour en discuter séparément. Nous aborderons les invites utilisateur, telles que les alertes, dans le prochain chapitre.

Généralement, ce type de boîtes de dialogue sont utilisées lorsque l'auteur du site Web souhaite obtenir des informations spécifiques de l'utilisateur (par exemple, un nom d'utilisateur/mot de passe) ou faire exécuter certaines tâches par l'utilisateur. Par exemple, sur un site de commerce électronique, l'ajout d'un article à un panier peut faire apparaître une fenêtre d'information comme suit :

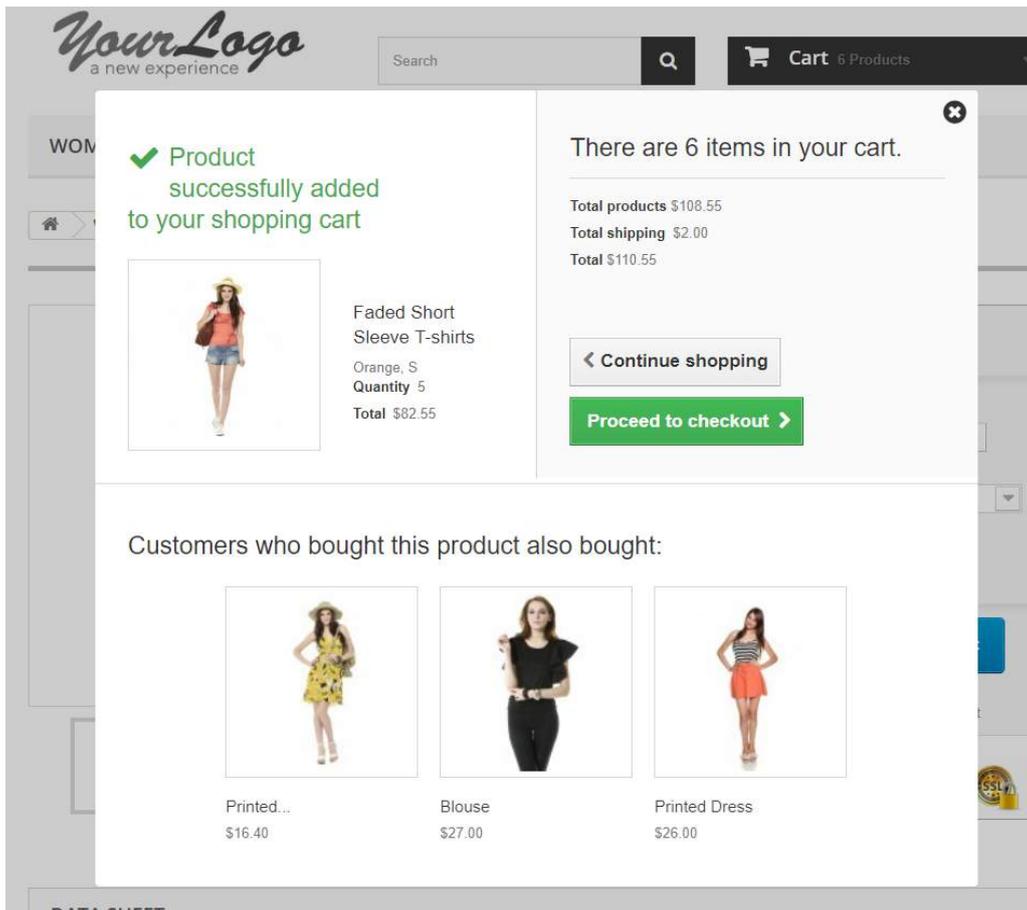


Figure 67: Fenêtre modale du panier de commerce électronique

Dans l'exemple ci-dessus, l'utilisateur peut choisir le chemin à suivre : passer à sa commande ou continuer ses achats. De plus, il est possible de présenter des informations supplémentaires dans le dialogue modal. Dans ce cas, le site de commerce électronique a pour but d'inciter l'utilisateur à acheter quelque chose d'autre en fonction de ce qu'il a placé dans le panier.

Tout le code de la boîte de dialogue modal se trouve dans le code HTML qui a fait apparaître la boîte de dialogue modal. Par conséquent, manipuler la boîte de dialogue modal revient à trouver le code dans la page appelante et à la manipuler. Ceci suit les mêmes règles que pour localiser et manipuler les contrôles de formulaire que celles dont nous avons discuté précédemment.

Par exemple, supposons que nous voulions cliquer sur le bouton **Proceed to Checkout** dans cette boîte de dialogue modal. La première étape serait de déterminer l'emplacement du code pour la boîte de dialogue modal. Dans ce cas, l'**ID** de la section représentant l'élément modal est **layer\_cart**. Nous allons créer une référence objet WebDriver pour cet élément du code de cette manière :

```
modal = driver.find_element_by_id('layer_cart')
```

Figure 68: Recherche de l'élément modal

La tâche suivante serait d'identifier l'élément représentant le bouton en utilisant la fonctionnalité Inspecter du navigateur. Dans ce cas, le nom de classe du bouton est **button\_medium**. Cette fois encore, nous voulons créer une référence à cet élément, afin de pouvoir le manipuler. Nous pouvons utiliser le code suivant :

```
proceed_button = modal.find_element_by_class_name('button-medium')
```

Figure 69: Recherche d'un bouton dans l'élément modal

Une fois trouvé, cliquer sur le bouton équivaut à appeler la méthode click() pour cette référence :

```
proceed_button.click()
```

Figure 70: Cliquer sur le bouton

À cette étape, la boîte de dialogue modale est fermée, et nous nous déplaçons vers un nouvel emplacement dans la fenêtre principale du navigateur, dans ce cas la page de résumé du panier d'achat est affichée comme indiqué ci-dessous :

01. Summary 02. Sign in 03. Address 04. Shipping 05. Payment

Product	Description	Avail.	Unit price	Qty	Total	
	Printed Dress SKU : demo_3 Color : Orange, Size : S	In stock	\$26.00	1	\$26.00	
	Faded Short Sleeve T-shirts SKU : demo_1 Color : Orange, Size : S	In stock	\$16.51	6	\$99.06	
					Total products	\$125.06
					Total shipping	\$2.00
					Total	\$127.06
					Tax	\$0.00
					TOTAL	\$127.06

[Continue shopping](#) [Proceed to checkout](#)

Figure 71: Écran du panier d'achat

## 3.8 Interagir avec les invites de l'utilisateur dans les navigateurs Web à l'aide des commandes WebDriver

Les invites utilisateur (appelée « User prompts » en anglais) sont des fenêtres modales dans lesquelles l'utilisateur doit interagir avec les commandes avant de pouvoir continuer à interagir avec les commandes de la fenêtre du navigateur elle-même.

Pour les besoins de l'automatisation, les invites utilisateur ne sont généralement pas traitées automatiquement. Par conséquent, si votre script essaie d'ignorer l'invite et de continuer à envoyer des commandes à la fenêtre du navigateur lui-même, une **erreur d'ouverture d'alerte inattendue** sera renvoyée par l'action suivante réalisée.

Chaque invite utilisateur est associée à un message d'invite utilisateur (qui peut être NULL). Il s'agit d'une zone de texte qui peut être saisie à l'aide du code indiqué ci-dessous.

Le W3C définit trois boîtes de dialogue de type alerte différentes :

- Alert
- Confirm
- Prompt

Comme ils sont définis de telle sorte qu'ils fonctionnent pratiquement tous de la même façon, nous abordons l'invite d'alerte (Alert).

La boîte de dialogue d'alerte est souvent utilisée pour s'assurer que l'utilisateur est informé de certaines informations importantes.

Comme les boîtes de dialogue d'alerte ne font pas partie de la page Web, elles nécessitent un traitement spécial. WebDriver utilisant Python dispose d'un ensemble de méthodes qui vous permettent de contrôler le dialogue d'alerte depuis votre script d'automatisation. Ces méthodes sont communes aux trois boîtes de dialogue d'invite utilisateur.

Ceci crée d'abord une référence en utilisant une méthode de l'objet WebDriver appelée **switch\_to**. La syntaxe pour utiliser cette méthode est la suivante :

```
alert = driver.switch_to.alert
```

*Figure 72: Créer un objet d'alerte*

Nous pouvons obtenir le texte dans l'alerte via :

```
msg_text = alert.text
```

*Figure 73: Obtenir du texte à partir d'une alerte*

Pour déterminer si le texte attendu se trouve dans l'alerte, passez dans l'alerte, récupérez le texte, puis vérifiez si le texte attendu se trouvait dans l'alerte. Si le texte est là, la variable **passed** sera mise à **True**. Si ce n'est pas le cas, la variable **passed** sera mise à **False**.

```
alert = driver.switch_to.alert
msg_text = alert.text
expected_text = 'XYZ'
assert expected_text in msg_text, "The expected text not found"
```

Figure 74: Comparaison du texte d'une alerte

Les boîtes de dialogue d'alerte peuvent être fermées de deux façons. Considérez la méthode **accept()** comme appuyant sur le bouton OK, et la méthode **dismiss()** comme appuyant sur le bouton Cancel, ou le bouton de fermeture de fenêtre dans le coin supérieur de l'invite.

```
alert = driver.switch_to.alert
alert.accept()
alert.dismiss()
```

Figure 75: Fermeture d'une alerte

## Chapitre 4 - Préparer des scripts de test maintenables

### Termes

fixture, Page Object Pattern, persona

### Objectifs d'apprentissage pour la préparation de scripts de test maintenables

- STF-4.1 (K2) Comprendre les facteurs qui favorisent et affectent la maintenabilité des scripts de test
- STF-4.2 (K3) Utiliser des mécanismes d'attente appropriés
- STF-4.3 (K4) Analyser l'interface graphique du SUT et utiliser Page Objects pour créer des abstractions
- STF-4.4 (K4) Analyser les scripts de test et appliquer les principes du Keyword Driven Testing à la construction de scripts de test

#### 4.1 Maintenabilité des scripts de test

Au chapitre 1, section 2, nous avons discuté de la différence entre un test manuel et un script automatisé. Ce point a une influence directe sur la maintenabilité du logiciel d'automatisation des tests, c'est pourquoi nous abordons de nouveau cette question dans cette section.

Réduit à l'essentiel, un cas de test manuel est un ensemble d'instructions abstraites qui ne sont utiles que lorsqu'elles sont utilisées par un testeur manuel pour effectuer le test. Les données et les résultats attendus doivent aussi être définis, mais un élément important réside dans la manière dont ces informations sont interprétées. En effet, le testeur manuel par son interprétation du cas de test, ajoute le contexte et le réalisme à ces énoncés abstraits, ce qui permet d'exécuter avec succès des tests de différents niveaux de complexité.

Si une étape du script manuel indique de cliquer sur un bouton, le testeur manuel peut le faire aisément. Il ne s'arrête pas pour se demander si le contrôle est visible, s'il est activé ou s'il s'agit du bon contrôle pour la tâche. Le fait est, cependant, qu'il peut traiter ces questions de façon intuitive. Ainsi, si le contrôle n'est pas visible, il peut essayer de le rendre visible. S'il n'est pas activé, il n'essaiera pas d'utiliser le contrôle à l'aveuglette ; il essaiera de comprendre pourquoi il est désactivé et de voir s'il est possible de le réparer. Une fois que le contrôle est activé, il peut réaliser les actions de test nécessaires. Et, si quelque chose se passe incorrectement avec ce contrôle, il peut identifier ce qui s'est passé, de sorte qu'il peut rédiger le rapport de défaut et/ou modifier la procédure de test manuelle.

Un outil d'automatisation ne possède pas cette intelligence humaine. Même s'il est coûteux, chacun d'eux ne prend en compte le contexte que de façon très limitée, sans intégrer le caractère vraisemblable ou non du test. Mais, les automaticiens de tests sont des êtres humains intelligents qui peuvent pallier à cela en ajoutant des éléments de contexte et le caractère vraisemblable du test lors de la programmation des scripts automatisés.

Cependant, plus nous essayons de programmer cette intelligence dans le script automatisé, plus le script devient complexe, et plus il y aura de risques que le script échoue simplement en raison de sa complexité intrinsèque.

L'automatisation de l'écriture pour les tests a souvent été comparée à une tentative d'atteindre l'inaccessible.

Ceci signifie que l'automatisation est une activité complexe. Cette complexité est liée à deux aspects. Nous devons intégrer de l'intelligence dans nos scripts pour qu'ils simulent au mieux un testeur humain exécutant un test. Et nous devons gérer une complexité sans cesse croissante dans les SUT sur lesquels nous travaillons.

Mais plus nous ajoutons de complexité à notre automatisation, plus il y a de risques prévisibles d'échecs de l'automatisation.

Peu importe à quel point nous sommes habiles en tant qu'automaticien de test, il y a des limites aux conditions que nous pouvons contrôler à l'aide du script automatisé.

Un testeur manuel peut essayer de comprendre pourquoi un certain contrôle n'est pas visible sur l'interface graphique. C'est plus difficile pour l'automaticien de tests lorsqu'il code le script automatisé ; cela peut conduire par exemple à indiquer au script d'attendre un certain temps, en espérant que le problème se résoudra de lui-même. Il en va de même pour la commande qui est activée.

Ce qu'un automaticien de tests peut faire, c'est essayer de s'assurer que l'élément sur l'interface est prêt à être utilisé, et si ce n'est pas le cas, attendre un certain temps, et s'il n'est toujours pas utilisable, enregistrer l'erreur et fermer l'exécution du script comme il faut pour passer à la prochaine exécution du test. Si nous pouvons accéder au contrôle, nous pouvons vérifier que le contrôle se comporte comme prévu après l'avoir manipulé. Et, s'il y a eu un problème, nous pouvons écrire un message de log utile, de sorte que nous puissions résoudre le problème plus efficacement et de manière plus efficiente ensuite.

Une façon de faciliter ce travail consiste à mettre de l'intelligence dans des fonctions plutôt que d'avoir à programmer l'intelligence dans chaque script individuellement. En d'autres termes, il s'agit de faire remonter l'intelligence au niveau de l'architecture d'automatisation des tests et/ou du framework d'automatisation plutôt qu'au niveau de chaque script individuel.

En déplaçant l'intelligence du test en dehors des scripts eux-mêmes, les scripts deviennent plus faciles à maintenir et plus évolutifs. Et si notre code qui ajoute de l'intelligence échoue, il fera échouer beaucoup de scripts, mais les corriger tous, pourra être réalisé en une seule action corrective.

Nous vous avons donné dans les chapitres précédents quelques exemples de construction de ces fonctions appelables. Le code que nous avons décrit, cependant, n'est pas aussi robuste que nous le souhaitons pour l'automatisation en production réelle. Les bonnes pratiques d'automatisation visent à construire des fonctions agrégées appelées fonctions Wrapper. Un exemple de fonction Wrapper avec l'intelligence intégrée est présenté ci-dessous sous une forme abstraite. Supposons que l'on veuille cliquer sur une case à cocher, nous allons construire la fonction Wrapper de telle sorte qu'elle imite ce que le testeur manuel fait réellement.

Les arguments de cette fonction Wrapper sont susceptibles d'inclure : la case à cocher à utiliser, l'état final désiré (coché ou non coché), et éventuellement le temps que nous sommes prêts à l'attendre si elle n'est pas prête tout de suite. Ainsi, les actions du script suivraient la logique suivante :

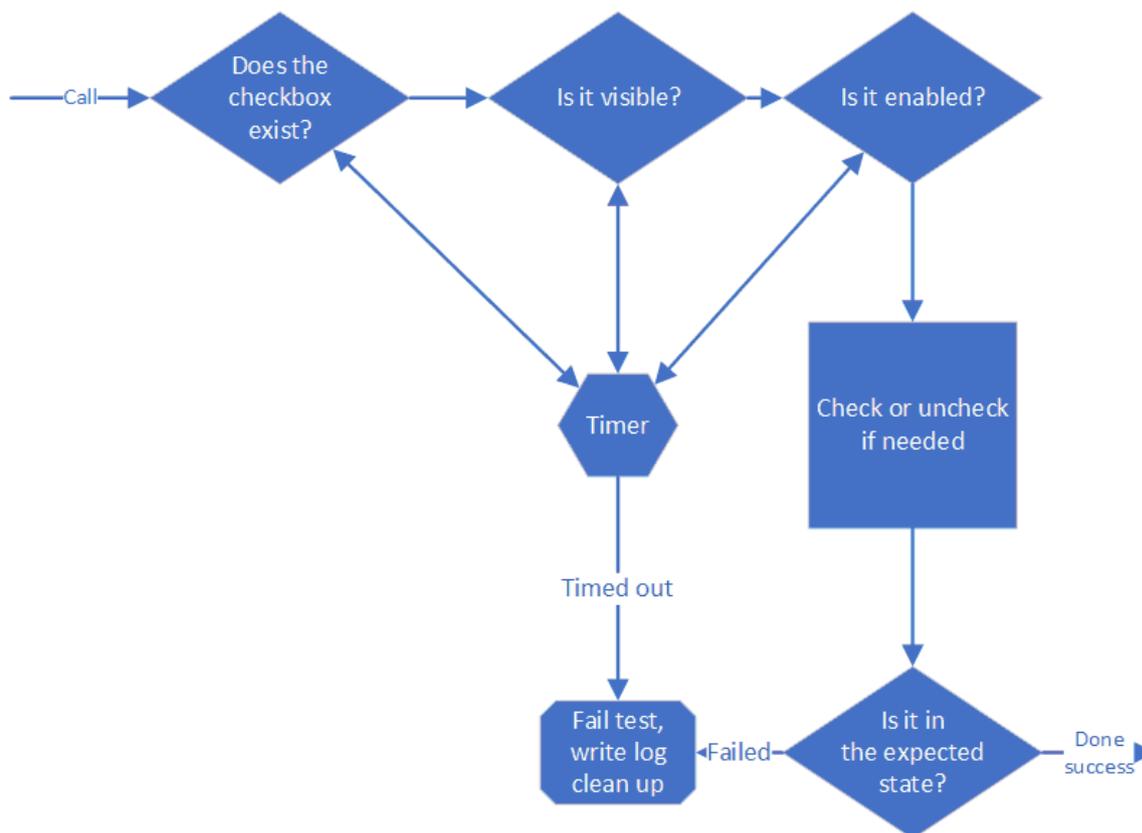


Figure 76: Logique de la fonction Wrapper

Le traitement des cas d'échec du test est à réaliser avec minutie. Les relevés du log doivent être complets afin de réduire l'effort de diagnostic en cas d'échec du test. Les actions de terminaison de l'exécution sur script de test (en particulier le nettoyage – clean up) doit garantir que l'exécution des scripts de la suite de test pourra être réalisée pour chacun des scripts l'un après l'autre. Une bonne pratique de l'automatisation est de s'assurer que l'exécution des scripts va pouvoir être enchaînée quel que soit le résultat d'exécution d'un script (qu'il échoue ou qu'il réussisse, le suivant doit pouvoir être exécuté. Si vous pouvez toujours exécuter le test suivant, vous pouvez exécuter toute la suite, quel que soit le nombre de tests échoués pendant l'exécution.

En définitive, les testeurs visent à détecter là où le SUT ne fonctionne pas correctement (et d'acquiescer de la confiance quand il semble fonctionner correctement). Si l'automatisation des tests est bien réalisée, nous devons nous attendre à ce que des défaillances se produisent ; l'exécution du script doit recueillir l'information sur ces défaillances et passer au test suivant.

Il convient de ne pas oublier que chaque test est conçu pour examiner le SUT, son environnement et son utilisation pour nous dire ce que nous ignorons encore. Un test qui ne dit pas au testeur une information qu'il n'avait pas n'est pas un test utile, qu'il soit automatisé ou non. Dans le programme de certification de niveau Fondation de l'ISTQB, il est question de techniques de conception de tests telles que le partitionnement des classes d'équivalence et l'analyse des valeurs limites. Ces techniques sont conçues pour réduire le

nombre de tests afin de trouver le plus rapidement possible les informations essentielles sur le SUT.

Les fonctions Wrappers sont importantes, mais d'autres fonctions peuvent être construites pour améliorer l'automatisation. Tout code que l'automatisation doit appeler de façon répétée peut être créé sous de la forme d'une fonction. Ceci est conforme aux bonnes pratiques de développement de la décomposition fonctionnelle et du refactoring. Vous pouvez créer des bibliothèques de fonctions qui peuvent être incluses dans votre code pour vous fournir, à vous et à vos coéquipiers, des boîtes à outils. N'oubliez pas de déplacer tout ce qui se trouve dans les scripts actuels vers ces bibliothèques de fonctions lorsque c'est pertinent.

Il faut du temps et des ressources pour mettre en place ces fonctions. Mais si vous voulez être un automaticien de tests efficient, c'est là un excellent investissement.

Lorsque vous testez un navigateur, assurez-vous de ne pas utiliser de chemins d'accès absolus lorsque vous localisez des éléments web avec XPath et sélecteurs CSS. Comme nous l'avons vu dans le chapitre précédent, tout changement apporté au code HTML est susceptible d'entraîner une modification de ces chemins, mettant en échec le code d'automatisation. Bien que les chemins relatifs puissent aussi changer, ils ont tendance à changer moins souvent et nécessitent donc moins de mise à jour.

Assurez-vous de discuter de l'automatisation avec les développeurs de votre organisation. Faites-leur connaître les difficultés rencontrées. Ils peuvent faire beaucoup de choses dans la façon dont ils écrivent leur code pour éviter les mises à jour répétées de votre code d'automatisation en rapport direct avec le code HTML.

Définissez les noms globaux (variables, constantes, noms de fonctions, etc.) qui ont un sens. Cela rend le code plus lisible, ce qui a pour effet secondaire de faciliter la maintenance du code d'automatisation. Une maintenance plus facile du code signifie généralement moins de défauts de régression lorsque des modifications sont apportées. Il faut quelques secondes de plus pour trouver des noms adéquats, mais cela peut faire gagner des heures de travail ensuite.

Commentez. Beaucoup de commentaires. Des commentaires significatifs. Beaucoup d'automaticiens de test se disent que c'est leur code et qu'ils se rappelleront pourquoi ils l'ont écrit ainsi. Ou ils pensent que l'automatisation est "différente", pas comme le vrai code. Ils ont tort. Vous pouvez oublier d'un jour à l'autre ce que vous avez fait d'intelligent pour résoudre un problème. Ne vous forcez pas à réinventer la roue. Lorsque votre projet d'automatisation commence à réussir, d'autres personnes vont y participer et vont travailler sur le code d'automatisation. Facilitez-leur la tâche.

Créez des comptes utilisateurs pour le test et des fixtures<sup>2</sup> qui sont spécifiquement destinés à l'automatisation. Ne les partagez pas avec des testeurs manuels. L'automatisation a besoin d'avoir une certitude sur les données utilisées dans l'automatisation ; les testeurs

---

<sup>2</sup> Note du traducteur : Voir la définition du terme « fixture » en annexe. Ce terme a été conservé en anglais car c'est ainsi qu'il est utilisé en général en France.

manuels peuvent être amené à effectuer des changements susceptibles de mettre en échec l'automatisation.

Ces comptes et fixtures associées ne devraient pas être liés à une personne en particulier. Les comptes génériques qui imitent les comptes réels sont beaucoup plus efficaces pour les isoler des changements. Vous devriez avoir suffisamment de comptes différents pour que plusieurs tests n'interfèrent pas avec les données de l'autre.

Créez suffisamment de données dans ces comptes pour qu'ils simulent des comptes réels. N'oubliez pas les différents profils de ces comptes. Si votre SUT est utilisé par différents types d'utilisateurs (par exemple, novices, techno geeks, utilisateurs expérimentés, etc.), ceux-ci doivent être représentés dans les comptes de test.

Nous avons discuté de l'enregistrement des résultats d'exécution (fichiers log) au chapitre 3, section 3.1. Prenez ce sujet au sérieux. Si votre automatisation est un succès, votre direction voudra toujours plus de tests automatisés. L'un des moyens de rendre l'automatisation plus évolutive est de bien gérer les logs dès le début. Vous ne voudrez sûrement pas avoir à reprendre l'ensemble de vos centaines (milliers ?) de scripts pour améliorer les logs.

Python dispose d'un ensemble robuste d'outils pour la création de logs ; vous pouvez aussi créer vos propres outils. Vous pouvez faire preuve d'initiative sur ce sujet. Les logs précis peuvent réduire le temps de diagnostic des problèmes en cas de défaillance. Tenez compte des besoins de votre organisation. Si votre SUT est un système critique pour la mission ou à la sûreté, vos logs doivent être suffisamment complets pour être audités.

Essayez de modéliser les processus de réflexion du testeur manuel. Qu'est-ce qu'ils apprennent et retirent du SUT lorsqu'ils font un test manuel ? Voyez si vous pouvez capturer et enregistrer ces mêmes informations. N'oubliez pas, vous pouvez être aussi créatif que vous le souhaitez pour déterminer le « comment », le « quand » et le « où » de vos enregistrements de logs.

Contrôlez les fichiers de logs que vous créez. Utilisez des conventions de nommage cohérentes et enregistrez les fichiers dans des dossiers bien organisés. Pensez à créer un dossier avec un horodatage dans le nom et à y placer tous les fichiers d'une exécution. Si plusieurs machines exécutent l'automatisation, ou si vous testez dans des environnements différents, vous pouvez ajouter le nom du poste de travail ou les noms d'environnement aux noms des dossiers.

Pensez à inclure des horodatages dans les noms de fichiers. Cela garantira que votre automatisation n'écrase pas les résultats des tests précédents. Si les fichiers ne sont pas nécessaires à l'avenir, placez-les dans des répertoires qui peuvent être détruits sans dommage. Si les fichiers doivent être sauvegardés, assurez-vous que la structure de vos dossiers contient ces informations.

Utilisez des noms de fichiers cohérents. Réunissez-vous avec les autres automaticiens et établissez une convention de style et de dénomination. Enseignez ces normes et lignes directrices aux nouveaux automaticiens qui se joignent à l'équipe. Il n'est pas difficile de

créer la confusion entre les répertoires quand chacun fait ce qu'il veut lorsqu'il crée des artefacts persistants.

Les automaticiens de test n'ont pas toujours respecté les bonnes pratiques du développement en appliquant leurs propres méthodes et de façon parfois désordonnée. Cependant, comme l'investissement dans l'automatisation est très couteux, des normes et lignes directrices minimales de codage devraient être adoptées pour en augmenter les chances de succès.

## 4.2 Mécanismes d'attente

Lorsqu'un testeur manuel effectue un test, la prise en compte du temps de réponse du système n'est pas vraiment un problème. Par exemple, supposons que le testeur ouvre un fichier. Si le dossier s'ouvre en temps voulu (le délai étant défini par le testeur), il n'y a pas d'autre considération à prendre en compte. Chaque fois qu'un testeur manuel clique sur une commande ou manipule autrement le SUT, il a une horloge implicite dans sa tête. S'il pense que quelque chose prend trop de temps, il est probable qu'il recommencera le test, avec une attention toute particulière au temps de réponse.

Mais, ce qui ne se produit jamais, c'est qu'un testeur reste assis pendant des heures à attendre patiemment qu'une action particulière se produise.

Nous avons discuté à plusieurs reprises dans ce syllabus du contexte qu'un testeur manuel apporte à un test. Le temps de réponse est l'un de ces aspects contextuels qui doivent être pris en considération lors des tests.

Supposons que le testeur ouvre un très petit fichier : quelques centaines d'octets. S'il ne s'ouvre pas instantanément, le testeur est susceptible d'être interpellé et peut ouvrir un rapport d'incident à ce sujet. Supposons que ce même testeur essaie d'ouvrir un fichier de deux gigaoctets ; s'il faut trente secondes pour l'ouvrir, il ne sera peut-être pas surpris du tout. S'il reçoit un message d'information indiquant que le fichier va être lent à s'ouvrir, il efface le message et continue à attendre la fin du traitement.

Le contexte est important.

Dans le domaine de l'automatisation, quel que soit l'outil, il n'a que peu ou pas de contexte intégré.

Généralement, l'outil d'automatisation a un temps intégré pendant lequel il est prêt à attendre qu'une action se produise. Si l'action attendue ne se produit pas dans ce délai, alors l'outil enregistre un échec et il poursuit le test (ce qu'il réalise dépend de l'outil, de l'installation, et une variété d'autres choses).

Si l'outil est réglé pour attendre 3 secondes, alors une action qui se produit en 3,0001 secondes - ce qui est probablement dans la plage de temps contextuelle du testeur manuel - sera considérée comme un échec.

Si l'automaticien de tests supprime toute prise en compte du temps, il est alors possible que l'outil soit bloqué en attente d'une réponse du SUT à une action. Il est très pénible que de découvrir un lundi matin que la suite d'automatisation que vous avez lancé le vendredi soir avant de partir est bloquée au test n°2 et qu'elle s'est arrêtée là trois minutes après votre départ du bureau.

Un automaticien doit maîtriser les besoins de l'automatisation et le fonctionnement de son ou ses outils.

Selenium WebDriver avec Python dispose de plusieurs mécanismes d'attente différents qu'un automaticien peut utiliser lorsqu'il configure la synchronisation pour son automatisation. Utiliser un mécanisme d'attente explicite n'est généralement pas la meilleure approche pour cela, mais si cette méthode est utilisée couramment par de nombreux automaticiens.

Le code suivant devrait être reconnu par presque tous ceux qui ont déjà automatisé des tests :

```
import time
...
time.sleep(5)
```

*Figure 77: Une attente explicite*

Bien que ce soit parfois une bonne solution, ce n'est pas le cas en général. Nous avons vu des cas où des automaticiens ont utilisé si souvent les temps d'attente explicites que l'automatisation était plus lente qu'un testeur manuel effectuant exactement le même test.

Ce type de mécanisme d'attente considère le temps de réponse dans le pire des cas. Comme cela ne se produit pas souvent, la plus grande partie de ce temps d'attente est donc perdue.

Dans cette formation, nous considérons que la fonction **sleep()** ne devrait être utilisée que lors du débogage d'un exercice ; en dehors de cela, ne l'utilisez pas.

Selenium avec WebDriver a deux principaux types de mécanismes d'attente : les attentes implicites et les attentes explicites. Nous nous concentrons principalement sur les attentes explicites, mais nous présentons d'abord les attentes implicites.

Une attente implicite dans WebDriver est définie lorsque l'objet WebDriver est créé pour la première fois. Le code suivant va créer le pilote et définir l'attente implicite :

```
driver = webdriver.Chrome()
driver.implicitly_wait(10)
```

*Figure 78: Configurer l'attente implicite*

L'attente implicite, telle que définie ci-dessus, sera effective jusqu'à ce que WebDriver soit désactivé. Cette attente implicite demande à WebDriver d'interroger le DOM pendant un certain temps lorsqu'il essaie de trouver un élément qui n'est pas trouvé immédiatement. Le réglage par défaut est de 0 secondes d'attente. Chaque fois qu'un élément n'est pas trouvé immédiatement, le code ci-dessus indique au WebDriver de scruter pendant dix secondes, demandant (conceptuellement) toutes les millisecondes, "Êtes-vous encore là ?" Si l'élément apparaît pendant cette période, le script continue à s'exécuter. L'attente implicite fonctionnera pour tout ou partie des éléments qui sont définis dans le script.

Les temps d'attente explicites demandent que l'automaticien définisse (généralement pour un élément spécifique) exactement combien de temps WebDriver doit attendre pour cet élément particulier, souvent pour un état particulier de cet élément. Comme mentionné ci-dessus, le cas extrême d'une attente explicite est la méthode **sleep()**. L'utilisation de cette méthode est bien représentée par le dicton « utiliser un marteau pilon pour écraser une mouche ».

Au-delà de la fonction **sleep()**, les temps d'attente explicites sont faciles à utiliser, car les implémentations Python, Java et C# incluent toutes des méthodes pratiques qui fonctionnent avec des temps d'attente pour des conditions attendues spécifiques. En Python, ces temps d'attente sont codés en utilisant la méthode WebDriver **WebDriverWait()**, en conjonction avec une **ExpectedCondition**. Les conditions attendues sont définies pour de nombreuses conditions courantes qui peuvent se produire et sont accessibles en incluant le module `selenium.webdriver.support` comme indiqué ci-dessous :

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

*Figure 79: Importation de méthodes d'attente explicites*

Ce code met à la disposition de l'automaticien la possibilité d'utiliser les temps d'attente prévus prédéfinis qui sont disponibles. L'appel de l'attente explicite peut se faire à l'aide du code suivant (en supposant que les importations ci-dessus sont effectuées) :

```
wait = WebDriverWait(driver, 10)
element = wait.until(EC.element_to_be_clickable((By.ID, 'someID')))
```

Ce code attendra jusqu'à 10 secondes, en vérifiant toutes les 500 millisecondes, qu'un élément soit identifié par un **ID** défini par 'someID'. Si l'élément web n'est pas trouvé après 10 secondes, le code lancera une exception **TimeoutException**. Si l'élément web est trouvé, une référence à celui-ci sera placée dans la variable **element** et le script continuera.

De nombreux automaticiens incluront ce code dans les fonctions Wrappers afin que chaque élément web puisse être géré correctement en cas de lenteur dans les temps de réponse.

Il y a une grande diversité de conditions attendues définies pour une utilisation en Python, notamment les suivantes :

- `title_is`
- `title_contains`

- presence\_of\_element\_located
- visibility\_of\_element\_located
- visibility\_of
- presence\_of\_all\_elements\_located
- text\_to\_be\_present\_in\_element
- text\_to\_be\_present\_in\_element\_value
- frame\_to\_be\_available\_and\_switch\_to\_it
- invisibility\_of\_element\_located
- element\_to\_be\_clickable
- staleness\_of
- element\_to\_be\_selected
- element\_located\_to\_be\_selected
- element\_selection\_state\_to\_be
- element\_located\_selection\_state\_to\_be
- alert\_is\_present

Des conditions d'attente spécifiques peuvent également être créées, mais cela dépasse le cadre de ce syllabus.

### 4.3 Page Objects

Plus tôt dans le chapitre, nous avons fait la recommandation que, dans la mesure du possible, nous éliminions la complexité des scripts pour la déplacer au niveau de l'architecture d'automatisation / le framework d'automatisation. Nous allons aborder ce sujet un peu plus en détail dans cette section au travers du mécanisme appelé *Page Objects*, qui est un schéma s'appuyant sur le modèle appelé *Page Object Pattern*.

Un Page Object représente une zone de l'interface de l'application Web avec laquelle votre test va interagir. Il y a plusieurs raisons pour utiliser ce mécanisme :

- Cela permet de créer du code réutilisable qui peut être partagé par plusieurs scripts de test.
- Cela permet de réduire la quantité de code dupliqué.
- Cela permet de réduire les coûts et les efforts de maintenance des scripts d'automatisation des tests.
- Cela permet d'encapsuler toutes les opérations sur l'interface graphique du SUT en une seule couche.
- Cela sépare clairement les parties métier et les parties techniques de la conception de l'automatisation des tests.
- Quand il y a inévitablement un changement, cela donne un endroit unique dans le code pour corriger tous les scripts concernés.

Page Objects et Page Object Pattern ont été mentionnés à plusieurs reprises dans ce syllabus. Page Object Pattern désigne l'utilisation de Page Objects dans l'architecture d'automatisation des tests, si bien que ces deux termes peuvent être utilisés de manière presque interchangeable.

L'un des principes importants de la mise en place d'une architecture maintenable d'automatisation des tests est de la diviser en couches. L'une des couches proposées par l'ISTQB (dans le syllabus de niveau avancé Automaticien de tests) est la couche d'abstraction des tests ; cette couche assure l'interface entre la logique métier du cas de test et les besoins concrets du pilotage du SUT. Les Page Objects font partie de cette couche en abstrayant l'interface graphique du SUT. Abstraire ici signifie cacher les détails sur la façon dont nous contrôlons les fonctions internes de l'interface graphique qui sont utilisées dans d'autres scripts. Ci-dessous se trouve un fragment de code qui pourrait figurer dans un script lorsque Page Object Pattern n'est pas utilisé. Vous pouvez voir que ce code n'est pas facilement lisible ; les localisateurs de commandes particulières sont ici intimement liés au code.

```
first_name = self.browser.find_element_by_css_selector("#id_first_name")
first_name.send_keys("Clem")
last_name = self.browser.find_element_by_css_selector("#id_last_name")
last_name.send_keys("Kaddidlehopper")
password = self.browser.find_element_by_css_selector("#id_password")
password.send_keys("QWERTY")
email = self.browser.find_element_by_css_selector("#id_email")
email.send_keys("test+43@example.com")
product = self.browser.find_element_by_css_selector("#id_the_product")
product.send_keys("test")
self.browser.find_element_by_css_selector('#create_account_form button').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
self.browser.find_element_by_css_selector('#next-step').click()
```

*Figure 80: Du code d'automatisation mal conçu (sans Page Objects)*

Voici maintenant le même code une fois créé un Page Object :

```
signup_form = homepage.getSignupForm()
signup_form.setName("Clem", "Kaddidlehopper")
signup_form.setPassword("QWERTY")
signup_form.setEmail('test+43@example.com')
signup_form.setProductName('test')
onboarding_1 = signup_form.submit()
onboarding_2 = onboarding_1.next()
onboarding_3 = onboarding_2.next()
items_page = onboarding_3.next()
```

*Figure 81: Même code avec Page Object*

Cette deuxième écriture est plus lisible et plus concise.

Le contenu du premier extrait de code existe toujours. Il est encapsulé sous forme de Page Object avec des appels de fonction dans le second extrait de code. Le code Page Object ressemblerait à ceci :

```

class SignupPage(BasePage):
    url = "http://localhost:8000/account/create/"

    def setName(self, first, last):
        self.fill_form_by_id("id_first_name", first)
        self.fill_form_by_id("id_last_name", last)

    def setEmail(self, email):
        self.fill_form_by_id("id_email", email)

    def setPassword(self, password):
        self.fill_form_by_id("id_password", password)
        self.fill_form_by_id("id_password_confirmation", password)

    def setProductName(self, name):
        self.fill_form_by_id("id_first_product_name", name)

    def submit(self):
        self.driver.find('#create_account_form button').click()
        return OnboardingInvitePage(self.driver)

```

*Figure 82: Exemple de code Page Object*

Nous avons abstrait une grande partie de la complexité du scénario et l'avons transférée au sein de l'architecture d'automatisation des tests. La complexité est toujours là car cela nous permet de créer une automatisation répondant aux besoins. Mais, en la masquant dans la couche d'abstraction, cela permet une création plus rapide de scripts valides et performants par les personnes en charge de les écrire.

Le masquage de la complexité, sa gestion au niveau souhaité, est une pratique de base en matière de programmation. Songez qu'il y a de nombreuses années, la première programmation se faisait en branchant des câbles de branchement qui connectaient directement les entrées au processeur. Plus tard, tous les programmeurs écrivaient en assembleur, fabriquant du code qui parlait essentiellement le même langage que le processeur de l'ordinateur mais sans besoin de connexions physiques. Viennent ensuite les langages de plus haut niveau, Fortran, Cobol, C, qui sont beaucoup plus faciles à programmer ; le code est compilé pour créer le code objet qui fonctionnera avec le processeur, mais le programmeur n'a pas besoin de connaître le microcode du processeur. Les langages orientés objet, C++, Delphi, Java, sont apparus ensuite et ont simplifié encore plus la tâche.

Dans cette formation, nous utilisons Python qui permet de masquer l'essentiel de la complexité de la manipulation des objets du navigateur. L'utilisation du modèle Page Object Pattern n'est qu'une étape supplémentaire.

Page Object désigne une classe, un module ou un ensemble de fonctions qui contient l'interface vers un formulaire, une page ou un fragment de page du SUT qu'un automaticien de tests souhaite contrôler.

Page Object permet de gérer les actions métier qui sont utilisées comme étapes du test dans la couche d'exécution des tests. Page Object Pattern est ainsi une mise en application de l'approche appelée Keyword Driven Testing, qui permet à un projet d'automatisation de

réduire les efforts de maintenance. Nous discutons des tests dirigés par les mots-clés dans la section suivante.

Le modèle Page Object Pattern est particulièrement utile lors de la maintenance des scripts de test et peut réduire considérablement le temps nécessaire à la mise à jour des scripts après modification de l'interface graphique du SUT. Mais Page Object Pattern n'est pas une recette miracle pour les problèmes de maintenance de l'automatisation des tests. Bien qu'il puisse réduire les coûts, dans certaines situations, comme le changement de la logique métier du SUT, la mise à jour des scripts de test peut encore prendre beaucoup de temps. Dans ce cas, notez qu'il ne s'agit pas directement d'une question d'automatisation. Si vous modifiez la logique du SUT, vos scénarios de test manuels devront également être modifiés. Mais le changement de la logique métier des SUT n'est pas si fréquent, car si c'était le cas, cela obligerait les utilisateurs de cette application à réapprendre fréquemment comment l'utiliser.

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.

Dans une architecture d'automatisation des tests bien conçue, un Page Object dans une couche d'abstraction de test encapsule les appels aux méthodes Selenium WebDriver, de sorte que la couche d'exécution de test traite uniquement le niveau métier. Dans ces architectures d'automatisation, il n'y a pas d'importation ou d'utilisation de la bibliothèque Selenium dans la couche d'exécution des tests.

## 4.4 Tests dirigés par mots-clé (Keyword Driven Testing)

Au chapitre 1, section 2 de ce syllabus, nous avons discuté de l'efficacité d'un cas de test manuel. Nous avons écrit :

*Au minimum, un script de test manuel tend à contenir des informations en trois colonnes.*

*La première colonne contiendra un résumé de la tâche à accomplir. Abstraite, elle n'a donc pas besoin d'être modifiée lorsque le logiciel change. Par exemple, la tâche "Ajouter un enregistrement à la base de données" est une tâche abstraite. Quelle que soit la version de la base de données, cette tâche abstraite peut être exécutée -*

*en supposant qu'un testeur manuel possède les connaissances du domaine pour la traduire en actions concrètes.*

L'idée de tâche abstraite est un concept très puissant. Songez à l'ouverture d'un document dans un traitement de texte. Appelons-le *OpenFile*. Nous devons ouvrir un fichier dans un traitement de texte sous DOS, Windows 3.1, Windows 95, Macintosh et sous Unix. Chaque version de processeur nécessite que cette tâche soit effectuée. Le "comment" de l'ouverture d'un fichier est différent pour chaque implémentation de traitement de texte, mais le "quoi" reste le même. *OpenFile* est ainsi une tâche abstraite, prototype pour un mot-clé.

Le principe de base du Keyword Driven Testing (KDT) est que chaque application comporte un ensemble de tâches différentes qui doivent être effectuées pour utiliser l'application. Nous ouvrons et sauvegardons les fichiers. Nous créons, lisons, mettons à jour et supprimons les enregistrements de la base de données. Ces tâches tendent à être une représentation abstraite de ce qu'un utilisateur doit faire pour interagir avec le logiciel. Comme il s'agit de concepts abstraits, ils changent rarement à mesure que les versions du logiciel sont mises à jour.

Les tests manuels tirent parti de cette abstraction ; les tâches fonctionnelles pour presque toutes les applications changent rarement. Une fois qu'un utilisateur apprend l'interface, les tâches fonctionnelles qu'il doit effectuer avec l'application resteront presque toujours les mêmes. La façon dont les utilisateurs accomplissent les tâches individuelles peut changer, mais les concepts derrière les tâches ne changent pas. Les tests manuels n'ont pas besoin d'être changés à chaque version parce que le 'quoi' que nous testons tend à être très stable par rapport au 'comment'.

Les mots-clés que nous créons sont avant tout un méta-langage que nous pouvons utiliser pour tester. Comme les testeurs manuels lorsqu'ils écrivent des scripts de test manuels, nous identifions ces tâches 'quoi' dans le logiciel et les identifions avec un terme descriptif qui devient notre mot-clé. Les analystes de tests sont les personnes adéquates pour définir les mots-clés dont ils ont besoin pour tester une application.

L'une des conditions préalables les plus importantes pour implémenter des scripts de test maintenables (manuels ou automatisés) est une bonne structure où les actions du script restent abstraites, de sorte qu'elles changent rarement. De nouveau, nous constatons que la séparation de la partie implémentation concrète de l'interface de la tâche abstraite est une excellente technique de conception.

Dans le chapitre 1, section 5, lorsque l'architecture d'automatisation des tests a été décrite, nous avons souligné la séparation en couches bien définies de l'automatisation des tests. Le cas de test est décrit dans la couche de définition de test en termes abstraits. La couche d'adaptation de test qui sert d'interface entre le scénario de test et l'interface graphique physique du SUT. La couche d'exécution des tests qui inclut l'outil qui va réellement exécuter les tests sur le SUT.

L'ISTQB définit le KDT comme une technique de scripting qui utilise des fichiers de données pour contenir à la fois les données de test et les résultats attendus, mais aussi les mots-clés (les énoncés abstraits représentant le " quoi " – c'est-à-dire ce que le système est censé

faire). Les mots-clés sont souvent aussi appelés *mots d'action*. La définition de l'ISTQB indique également que les mots-clés sont interprétés par des scripts de support spécifiques qui sont appelés par le script de contrôle du test.

Examinons cette définition de plus près. KDT est une technique de scripting, ce qui signifie que c'est un moyen d'automatiser les tests. Les fichiers KDT contiennent les données de test, les résultats attendus et les mots-clés.

Les mots-clés sont des actions métier ou les étapes d'un scénario de test. Ce sont exactement les mêmes que ceux de la première colonne d'un cas de test manuel.

Lors de la mise en œuvre de l'automatisation des tests selon les principes du KDT, les cas de test eux-mêmes ne contiennent pas d'actions spécifiques de bas niveau à réaliser sur le SUT (le 'comment' de l'action). Les cas de test automatisés contiennent des séquences d'étapes de test qui sont généralement des actions métier abstraites et de haut niveau (par exemple, "se connecter à un système", "effectuer un paiement", "trouver un produit"). Les actions concrètes exactes sur le SUT sont cachées dans l'implémentation des mots-clés (par exemple, dans les Page Objects comme décrit dans la section précédente). Notez que cela reproduit la séparation entre un cas de test manuel et le testeur humain qui va ajouter le contexte et le caractère vraisemblable au test tout en manipulant concrètement l'interface graphique pour exécuter le test.

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 automaticiens 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.
- 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.

- Parce que les scripts sont séparés du niveau d'implémentation, différents outils peuvent être utilisés de manière interchangeable.

Parfois, les bibliothèques de mots-clés utilisent d'autres mots-clés, de sorte que toute la structure des mots-clés peut devenir assez complexe. Dans une telle structure, certains mots-clés auront un haut niveau d'abstraction, par exemple, "Ajouter un produit X à la facture avec le prix Y et la quantité Z", et certains d'entre eux auront un niveau assez bas, par exemple, "Cliquez le bouton >>Cancel<<<".

De ce fait, les mots-clés ont plusieurs lieux de définition et d'utilisation dans l'architecture générale d'automatisation des tests :

- Les mots-clés de bas niveau sont implémentés en tant que Page Objects dans la couche d'adaptation des tests, ils réalisent les actions concrètes sur le SUT.
- Les mots-clés de bas niveau sont utilisés comme parties de mots-clés de niveau supérieur dans la bibliothèque de la couche d'exécution des tests.
- Des mots-clés de haut niveau sont implémentés dans la bibliothèque des mots d'action.
- Les mots-clés de haut niveau sont utilisés comme étapes de test des procédures de test dans la couche d'exécution des tests.

La définition du KDT selon l'ISTQB précise que les mots-clés sont interprétés par des scripts de support dédiés. C'est le cas lorsque le KDT est mis en œuvre à l'aide d'outils spécialisés tels que Cucumber ou RobotFramework.

Cependant, il est possible de suivre les principes du KDT tout en implémentant un framework d'automatisation des tests avec des langages de programmation généraux tels que Python, Java ou C#. Dans ce cas, chaque mot-clé devient l'invocation d'une fonction ou d'une méthode pour les objets de la bibliothèque de test.

Il y a deux façons de mettre en œuvre le KDT dans la pratique. La première, qui peut être appelé « KDT classique », tel que décrit par Dorothy Graham dans son livre sur l'automatisation des tests, est une approche descendante. Sa première étape consiste à concevoir des cas de test comme ils seraient conçus pour des tests manuels. Ensuite, les étapes de ces cas de test sont abstraites pour devenir des mots-clés de haut niveau, qui peuvent être décomposés en mots-clés de bas niveau ou sont implémentés dans un outil ou un langage de programmation. Cette méthode est plus efficace lorsque la bibliothèque de tests contient déjà de nombreuses fonctions/méthodes qui exécutent des tâches sur le SUT. Elle peut également être utile lorsque des scripts automatisés sont conçus à partir de cas de test manuels plutôt que d'être écrits à partir de rien.

La deuxième méthode est la mise en œuvre de scripts de façon ascendante. Cela signifie que les scripts sont enregistrés par un outil (par exemple, Selenium IDE), puis sont remaniés et restructurés dans une architecture appropriée d'automatisation des tests KDT. Cette approche permet aux équipes de test d'écrire rapidement plusieurs scripts d'automatisation de test et de les exécuter sur le SUT.

Malheureusement, écrire plus de vingt ou trente scripts en utilisant cette approche « quick-and-dirty » sans créer une architecture d'automatisation des tests bien conçue risquerait d'exposer le projet d'automatisation à des coûts et des efforts de maintenance élevés par la suite. De plus, les tests qui sont d'abord écrits sans suivre des cas de test manuels risquent de ne pas être des tests pertinents (c.-à-d. qu'ils peuvent ne pas réellement être focalisés sur les risques importants qui doivent être testés).

Lors de la mise en œuvre de l'architecture d'automatisation des tests basée sur les principes du KDT, il faut également prendre en compte les points suivants :

- Des mots-clés à grain fin permettent des scénarios plus spécifiques, mais au prix de la complexité de la maintenance des scripts.
- Plus un mot-clé est à grain fin, plus il est intimement lié à l'interface du SUT et moins il est abstrait (comme dans l'exemple donné précédemment, "Click bouton >>Cancel<<< ").
- La conception initiale des mots-clés est importante, mais des mots-clés nouveaux et différents seront par la suite nécessaires, ce qui fait intervenir à la fois la logique métier et la fonctionnalité d'automatisation pour l'exécution.

Lorsqu'il est fait correctement, le KDT s'est révélé être une excellente approche d'automatisation qui permet de produire des scripts avec des coûts de maintenance minimum. Il permet de construire des frameworks d'automatisation de tests bien structurés, et utilise les bonnes pratiques de la conception logicielle.

## Annexe - Glossaire des termes Selenium

**Attribut de classe** : Un attribut HTML qui pointe vers une classe dans une feuille de style CSS. Il peut également être utilisé par un script JavaScript pour apporter des modifications aux éléments HTML avec une classe spécifiée.

**Comparateur** : Un outil pour automatiser la comparaison des résultats attendus avec les résultats effectifs.

**Sélecteur CSS** : Les sélecteurs sont des patterns qui ciblent les éléments HTML que vous voulez traiter.

**Document Object Model (DOM)** : une API qui permet de traiter un document HTML ou XML comme une arborescence dans laquelle chaque nœud est un objet représentant une partie du document.

**Fixture**: un objet fictif ou un environnement utilisé pour tester de façon cohérente un élément, un dispositif ou un logiciel.

**Framework**: Fournit un environnement pour l'exécution des scripts de test automatisés, y compris les outils, les bibliothèques et les fixtures.

**Fonction**: Une fonction Python est un groupe d'instructions réutilisables qui exécutent une tâche spécifique.

**Hook**: Une interface qui est introduite dans un système conçue principalement pour fournir une meilleure testabilité à ce système.

**HTML (HyperText Markup Language):** Le langage standard à base de balises pour la création de pages et d'applications Web.

**ID:** Un attribut qui spécifie une identification unique pour un élément HTML. La valeur doit être unique dans le document HTML.

**iframe:** Un cadre HTML en ligne, utilisé pour intégrer un autre document dans un document HTML.

**Fenêtre de dialogue modal :** Un écran ou une boîte de dialogue qui oblige l'utilisateur à interagir avec lui avant de pouvoir accéder à l'écran sous-jacent.

**Page Object Pattern :** Un modèle d'automatisation des tests qui implique que la logique technique et la logique métier soient traitées à des niveaux différents.

**Paradoxe du pesticide :** Un phénomène où la répétition d'un même test plusieurs fois le conduit à trouver moins de défauts.

**Persona :** Un profil d'utilisateur créé pour représenter un type d'utilisateurs qui interagissent avec un système d'une manière semblable.

**Pytest :** Un framework de test en Python.

**Tag :** Les éléments HTML sont délimités par des balises (tags), écrites à l'aide de crochets d'angle.

**Dette technique :** Le surcoût de la reprise causé par le choix de ne pas tenir compte des défauts de conception ou de mise en œuvre sur le court terme.

**WebDriver :** L'interface avec laquelle les tests Selenium sont écrits. Les différents navigateurs peuvent être contrôlés via différentes classes java, telles que ChromeDriver, FirefoxDriver, etc..

**Wrapper :** Une fonction dans une bibliothèque de logiciels dont le but principal est d'appeler une autre fonction, souvent en ajoutant ou en améliorant des fonctionnalités tout en masquant la complexité.

**XML (eXtensible Markup Language):** Un langage à base de balises qui définit un ensemble de règles pour coder des documents dans un format lisible à la fois par l'homme et par la machine.

**XPath (XML Path Language):** Un langage de requête pour la sélection de nœuds à partir d'un document XML.