REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE SAAD DAHLAB DE BLIDA
FACULTE DES SCIENCES
DEPARTEMENT D'INFORMATIQUE

MEMOIRE DE FIN D'ETUDES

Pour l'obtention

D'un Diplôme de Master en Informatique

Option : Systèmes Informatiques et Réseaux

THÈME :

Une Approche d'Automatisation de Test Fonctionnel pour les Applications Mobiles

Réalisé par :

BOUKHALFA Salim

ACHOUR Youcef

Soutenu devant :

Mr. KAMECHE Abdellah Hicham	USDB,	Encadreur
Mr. DERRAR Hacene	USDB,	Promoteur
Mme. ARKAM Meriem	USDB,	Présidente
Mme. FAREH Messaouda	USDB	Examinatrice

Promotion 2017/2018

Dédicaces

Nous dédions ce mémoire à :

***Nos mères**, qui ont œuvrées pour notre réussite, par leurs amour, leurs soutien, tous les sacrifices consentis et leurs précieux conseils, pour toute leurs assistance et leurs présence dans notre vie, reçois à travers ce travail aussi modeste soit-il, l'expression de nos sentiments et de notre éternelle gratitude.*

***Nos pères**, qui peuvent être fiers et trouver ici le résultat de longues années de sacrifices et de privations pour nous aider à avancer dans la vie. Puisse Dieu faire en sorte que ce travail porte son fruit. Merci pour les valeurs nobles, l'éducation et le soutient permanent venu de vous.*

***Nos frères et sœurs** qui n'ont cessé d'être pour nous des exemples de persévérance, de courage et de générosité.*

***Nos** professeurs qui doivent voir dans ce travail la fierté d'un savoir bien acquis.*

***Nos** amis qui nous ont aidé et soutenu durant notre travail.*

***A tous** ceux que nous aimons.*

Remerciements

Avant tout, nous tenons à remercier Allah le tout Puissant et Miséricordieux qui nous a donné la force et la patience d'accomplir ce mémoire, sans sa prospérité, nous ne serons jamais aptes à faire ce travail.

Nous adressons notre profonde gratitude et nos remerciements distingués à notre promoteur Mr. DERRAR Hacene et notre encadreur Mr. Kameche Abdallah Hicham pour nous avoir accordé durant tout ce projet, ainsi que pour tous leurs efforts fournis, leurs aide immense, leurs conseils précieux et leurs orientations pour atteindre nos objectifs.

Nous présentons nos gratitudes aux membres du jury qui ont bien voulu examiner et évaluer notre travail et qui nous font le grand honneur de participer à la soutenance.

Un grand merci aux enseignants du pavillon 1 de l'Université Saad Dahleb Blida (USDB), pour leurs abnégations dans notre instruction et orientation et notre suivi durant notre cursus universitaire. Plus particulièrement, au professeur Mr. OULD-KHAOUA Mohamed pour ses précieux conseils et son immense aide durant les deux années du Master.

Un grand merci à l'enseignante Mme. MELYARA Mezzi pour son encouragement permanent, et son soutien moral tout au long de notre parcours universitaire.

Nous voudrions exprimer notre profonde reconnaissance à nos parents qui ont tant sacrifié dans leurs vies, pour nous garantir une bonne éducation et un meilleur avenir. Ce travail leurs est particulièrement dédié.

Enfin, nos remerciements vont également à toutes les personnes qui ont contribué de près ou de loin à la concrétisation de ce travail soit par leurs paroles, leurs écrits, leurs conseils ou leurs critiques. Qu'ils trouvent tous ici, l'expression de notre gratitude et notre sincère considération.

Résumé

Afin d'assurer la qualité du logiciel, différents tests doivent être conduit, plus essentiellement le test fonctionnel. Mais jusqu'aujourd'hui, les tests sont souvent faits manuellement. Ces tests manuels se relève chronophages, laborieux et répétitifs, étant donné la grande diversité des systèmes d'exploitation mobiles et d'autres critères.

Automatiser l'ensemble du processus de test fonctionnel se relève un défi qui offre l'avantage de combler les lacunes des tests manuels.

C'est ici que s'inscrit l'objectif principal de ce projet, qui est de concevoir et réaliser un outil permettant d'automatiser tous le processus de test fonctionnel pour les applications mobiles, ce qui simplifie le travail des testeurs en termes de temps d'exécution, l'organisation et la gestion des tests.

Mots clés : Test logiciel, qualité du logiciel, automatisation du test fonctionnel, Appium, exécution symbolique, algorithme graphique.

ملخص

حتى نضمن جودة البرنامج، يجب إجراء تجارب مختلفة، وبشكل أساسي، التجارب الوظيفية. حتى يومنا هذا، كل هذه التجارب ماتزال يدوية. ولكن الجانب السلبي في هذه التجارب، أنها توقعنا في التكرار وتتطلب وقت طويل ومتعبة ومرد ذلك يعود إلى اختلاف أنظمة تشغيل الهواتف ومعايير أخرى، مما يشكل عائق أمام مبرمجي تطبيقات الهواتف الذكية. أتمتة هذه العملية تعتبر تحدي يمنحنا ميزات لسد ثغرات التجارب اليدوية.

هدفنا من خلال هذا البحث هو تصميم أداة وبرمجتها والتي ستسمح لنا بأتمتة عملية التجريب الوظيفي لتطبيقات الهواتف الذكية التي ستسهل عمل المجرب من ناحية وقت التشغيل وتنظيم وتسيير هذه التجارب.

الكلمات المفتاحية: اختبار البرمجيات، جودة البرامج، أتمتة الاختبار الوظيفي، Appium، التنفيذ الرمزي، الخوارزميات.

Abstract

In order to ensure the quality of the software, different tests must be conducted, more essentially the functional test. However, until today, tests are often done manually. These manual tests are time-consuming, laborious and repetitive, given the wide variety of mobile operating systems and other criteria.

Automating the entire functional testing process is a challenge that offers the benefit of filling the gaps of manual testing.

This is the main objective of this project, which is to design and implement a tool to automate all the functional testing process for mobile applications, which simplifies the work of the testers in terms of time of execution, organization and management of tests.

Keywords: Software test, software quality, automation of functional testing, Appium, symbolic execution, graphic algorithm.

Table des matières

Liste des figures

Liste des tableaux

INTRODUCTION GENERALE

Les modes de développement de logiciels s'orientent vers les méthodes plus agiles et réactives pour faire face aux évolutions perpétuelles demandées par le métier surtout dans ce contexte de digitalisation profonde qui caractérise les entreprises d'aujourd'hui. Partant de ce fait, les entreprises sont obligées de faire des choix et les méthodes de tests ont évolué en conséquence. Néanmoins, c'est au nombre et à la valeur des tests exercés sur un logiciel que détermineront son qualité, et par conséquent le succès de son accueil auprès des utilisateurs.

Dans cet environnement agile et réactif en matière de méthode et d'environnements technologiques, le test pour s'assurer de la qualité du logiciel est devenu une activité à part entière et plus particulièrement l'automatisation du processus de test.

Lorsqu'une application est créée ou améliorée dans une entreprise, elle est soumise à différents types de tests et plus essentiellement, le test fonctionnel. Ceux-ci permettent de vérifier le bon fonctionnement de ce qui a été développé et contrôler que la solution correspond à ce que le client a demandé. Cependant, l'exécution des tests est souvent faite d'une manière manuelle.

Les tests manuels se relève chronophages, laborieux et répétitifs. Plus particulièrement les tests des applications mobiles, notamment qui sont orientées grand public, peuvent relever d'un art en soi. Étant donné la grande diversité des systèmes d'exploitation mobiles, des formats d'appareils et des scénarios d'utilisation, la tâche peut être insurmontable.

Les tests manuels engendrent plusieurs problèmes pour les testeurs et pour l'ensemble des membres du projet. Ces problèmes peuvent être résumés dans les points suivants :

- Les tests manuels ne peuvent pas être utilisés pour tester l'ensemble de l'application. Les cas de test sont si nombreux qu'il devient impossible de les exécuter tous manuellement.
- Les tests manuels prennent beaucoup de temps et exigent des ressources humaines.
- Les tests manuels nécessitent de bonnes compétences. Avoir des testeurs inexpérimentés peut aggraver le problème au lieu de le simplifier. Et cela conduit également à des tests inefficaces.
- Les tests manuels sont souvent sujet à l'erreur humaine.
- Les tests manuels ne peuvent pas être répliqués facilement.

Automatiser l'ensemble du processus de test permet de combler les lacunes des tests manuels offrant un gain du temps considérable aux testeurs qui délèguent l'exécution des tests principaux, et améliore l'organisation et la rentabilité à terme. Ainsi, assurer une meilleure qualité des applications produit et un retour sur investissement pour l'entreprise.

C'est dans ce contexte que s'inscrit notre projet de fin d'études qui vise à proposer et développer une approche selon nos besoins permettant d'intégrer l'ensemble du cycle de la qualification logicielle de la conception des scénarios jusqu'à l'automatisation des tests fonctionnels appliqués aux applications mobiles.

Pour remédier aux carences des tests manuels et gérer les problèmes mentionnés avant, nous nous sommes fixés les objectifs suivants :

- Automatisé le processus de test.
- Améliorer la couverture des tests.
- Réduire le temps nécessaire pour effectuer les tests.
- Avoir la possibilité de répéter et réutiliser les tests autant de fois que nécessaire.
- Améliorer la précision des tests.
- Avoir une bonne organisation et flexibilité dans la préparation et l'exécution des tests.

Afin de faciliter la lecture de ce mémoire, nous allons présenter brièvement les chapitres qui le composent. Ces chapitres sont répartis comme suit :

- **Chapitre I – Test et Qualité du Logiciel – :** Dans ce chapitre, nous commencerons par une étude préliminaire sur le test et la qualité du logiciel, ensuite, nous détaillerons le processus du test logiciel, et enfin nous terminerons par présenter les méthodes de test.
- **Chapitre II – Automatisation du Test Logiciel – :** Dans ce chapitre, nous allons présenter des généralités sur l'automatisation du test, et décrire l'automatisation des activités du processus de test. Nous allons faire par la suite une étude comparative des outils d'automatisation du test logiciel ainsi de mettre en évidence les recherches académiques sur les méthodes de génération des tests automatique.
- **Chapitre III – Présentation de la Méthode Hybride de Test – :** Dans ce chapitre, nous allons commencer par les spécifications des besoins, puis nous exposerons notre solution.

- **Chapitre IV – Conception de la Solution – :** Dans ce chapitre, nous allons définir l'architecture de notre système ainsi que présenter la modélisation de notre système à travers les diagrammes d'UML (cas d'utilisation, séquence, classe).
- **Chapitre V – Implémentation et Test – :** Dans ce dernier chapitre de notre mémoire, nous allons citer les différents outils utilisés durant notre projet pour réaliser le système, ainsi que les différents tests effectués pour valider notre solution.

Chapitre I

Test et Qualité du Logiciel

1. Introduction

Dans ce chapitre nous présentons d'une façon générale l'aspect de test logiciel afin de fournir une compréhension de ce qu'est le test et pourquoi c'est un tel défi, et de souligner que à chaque fois que nous testons un logiciel, le processus doit être fait pour être aussi efficace et efficient que possible, pour cela nous débutons par une étude préliminaires sur le test, puis nous détaillerons le processus de test, et on termine par présenter les méthodes de test.

2. Préliminaires et définitions

2.1. Définition d'une application mobile

Une application mobile est un type de logiciel conçu pour fonctionner sur un appareil mobile, tel qu'un smartphone ou une tablette. Les applications mobiles servent souvent à fournir aux utilisateurs des services similaires à ceux accessibles sur les ordinateurs. Les applications sont généralement de petites unités logicielles individuelles avec une fonction limitée [1].

2.2. Types d'application mobile

Il existe trois types d'application mobile que tout utilisateur peut rencontrer, la figure ci-dessous illustre ces 3 types :

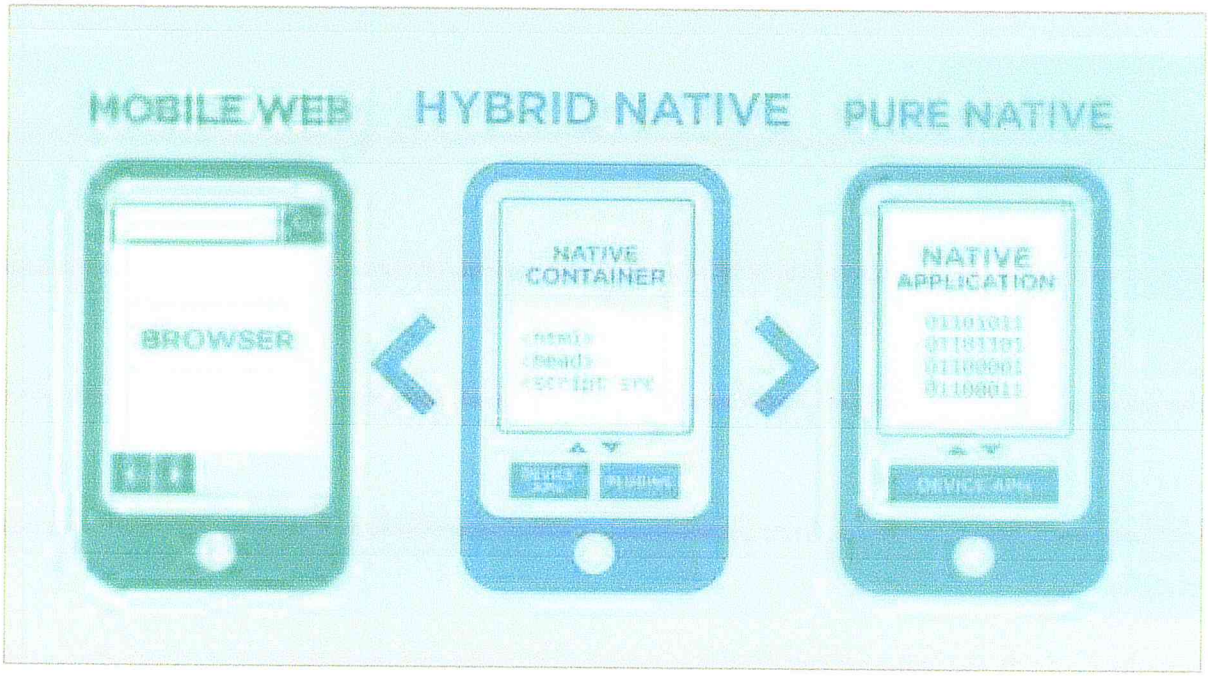

Figure I.1 : Les types d'application mobile

Chaque type d'application à ses caractéristiques propres :

- **Application web :** Chaque application conçue avec HTML et CSS, qui est opérationnelle sur un navigateur internet pour un smartphone est appelée application web [2].
- **Application hybride :** Les applications hybrides sont des applications Web développées avec HTML5 et JavaScript. Ces applications sont des sites Web intégrés de manière à se comporter comme une application native. Les applications hybrides s'exécutent sur iOS comme sur Android [3].
- **Application native :** Les applications natives sont développées spécialement pour une plateforme de système d'exploitation mobile qui prend en charge des langages de programmation tels que Swift et Objective-C pour iOS et Java™ pour Android. L'application permet d'accéder facilement à des fonctionnalités intégrées de l'appareil, telles que l'appareil photo, le Bluetooth et le GPS [3].

2.3. Qu'est-ce qu'un test ?

Le test est une activité dans laquelle un système ou un composant est exécuté dans des conditions spécifiées, les résultats sont observés ou enregistrés, et une évaluation est faite de certains aspects du système ou du composant pour identifier les différences entre les résultats obtenus et les résultats attendus [4].

2.4. Test du logiciel

Le test du logiciel n'est pas limité à une seul définition mais plusieurs :

- **Définition 01** : Le test de logiciel est un processus, ou une série de processus, conçu pour s'assurer que le code écrit par l'utilisateur fait ce pour quoi il a été conçu et qu'il ne fait rien d'imprévu. Les logiciels doivent être prévisibles et cohérents, n'offrant aucune surprise aux utilisateurs [5].
- **Définition 02** : Le test logiciel est un ensemble de processus visant à étudier, évaluer et vérifier l'exhaustivité et la qualité des logiciels informatiques. Les tests logiciels garantissent la conformité d'un produit logiciel aux exigences réglementaires, commerciales, techniques, fonctionnelles et les exigences de l'utilisateur [6].

L'objectif principal du test logiciel est de mesurer la performance et la qualité des logiciels ainsi que leur exhaustivité en termes d'exigences de base. Les tests logiciels impliquent

l'examen et la vérification des logiciels au moyen de différents processus de test. Les objectifs de ces processus peuvent inclure :

- Déterminer si le produit logiciel satisfait les exigences spécifiées.
- Démontrer si le produit logiciel atteint les objectifs requis. On cherche à savoir si le logiciel aide bien les utilisateurs à réaliser leurs tâches.
- Identifier les défauts techniques et s'assurer que le logiciel est sans erreur.
- Évaluation de la convivialité, performances, sécurité, compatibilité et de l'installation.

2.5. Qualité du logiciel

La qualité du logiciel est évaluée par un certain nombre de variables. Ces variables peuvent être divisées en critères de qualité externes et internes. La qualité externe est ce que l'utilisateur éprouve lors de l'exécution du logiciel dans son mode opérationnel. La qualité interne fait référence aux aspects qui dépendent du code et qui ne sont pas visibles par l'utilisateur final. La qualité externe est critique pour l'utilisateur, tandis que la qualité interne est significative pour le développeur seulement [7]. Le tableau ci-dessous répertorie les critères de qualité logicielle les plus évidents, ainsi que certains moins répandues :

	Utilisateur	Développeur	Mesurable
Qualité externe			
Fonctionnalités	X		Oui
Vitesse	X	X	Oui
Espace	X	X	Oui
Performance du réseau	X	X	Oui
Stabilité	X	X	Oui
Robustesse	X	X	Peu
Rétrocompatibilité	X		Oui
Sécurité	X		Difficile
Consommation d'énergie	X		Difficile
Qualité interne			
Couverture du test		X	Oui

	Utilisateur	Développeur	Mesurable
Testabilité		X	Difficile
Portabilité		X	Peu
Maintenabilité		X	Difficile
Documentation		X	Subjectif
Lisibilité		X	Subjectif
L'évolutivité		X	Peu

Tableau I.1 : Critères de qualité logicielle interne et externe [7]

Les critères de qualité logicielle externe sont détaillés comme suit :

- **Fonctionnalités :** C'est la raison même de l'écriture du logiciel qui est de fournir un service. La fonctionnalité, c'est le résultat attendu par le logiciel, par exemple : un résultat numérique, une chaîne, une capture d'écran, une page web, un son, etc., indépendamment des performances (vitesse, mémoire).
- **Vitesse :** à quelle vitesse l'application fournit-elle le service ? L'utilisateur fait l'expérience du temps écoulé entre le moment où il demande le service et le moment où le service est fourni.
- **Espace :** Combien de RAM et d'espace disque est pris par l'application ? Mais plus encore, combien de fois déplaçons-nous les données qui déclenchent une erreur de mémoire cache ou une écriture de disque, ont un impact dominant sur la vitesse de l'application. Une conception de données médiocre peut entraîner de très mauvaises performances.
- **Performance du réseau :** C'est une question de bande passante et de latence. Une mauvaise gestion des « sockets[1] » et des canaux peut entraîner un temps supplémentaire inutile à l'ouverture et à la fermeture des « sockets », « handshakes » et les « round-trips ». En ce qui concerne la mémoire, les techniques de mise en cache peuvent être utilisées pour réduire les ressources réseau consommatrices.
- **Stabilité :** à quelle fréquence faut-il faire une mise à jour de logiciel pour corriger les problèmes ? Pour l'utilisateur, ceci est un inconvénient. Pour le développeur, cela

[1] **Socket** désigne une interface logicielle avec les services du système d'exploitation, grâce à laquelle un développeur exploitera facilement les services d'un protocole réseau.

signifie que le code est fragile et pourrait bénéficier de meilleurs tests ou d'une réécriture partielle.

- **Robustesse :** à quelle fréquence l'application se stagne, fige ou se bloque complétement ? Quel est le degré de tolérance aux conditions extrêmes : ressources (CPU et mémoire / disque / réseau) limitées, panne de système ? Cet aspect est fortement lié à la testabilité et à la couverture.
- **Rétrocompatibilité :** Une nouvelle version de l'application peut-elle être utilisée avec les données d'une ancienne version ? Cet aspect est essentiel pour l'utilisateur, car une nouvelle version ne devrait pas nécessiter une migration coûteuse des données existantes.
- **Sécurité :** Qui est autorisé à accéder aux données ? Les données traitées par l'application peuvent-elles être compromises ? C'est un aspect crucial pour de nombreuses applications, et il devient de plus en plus difficile de l'évaluer avec la grande propagation de logiciels mobiles et web.
- **Consommation d'énergie :** La consommation d'énergie est un aspect important pour les applications mobiles, car un développeur doit prendre en considération dans une application, la gestion des producteurs et consommateurs d'énergie de l'appareil (batterie, processeur, Wi-Fi, écran, audio) et ne pas dépendre entièrement du système d'exploitation.

Les critères de qualité logicielle interne sont détaillés comme suit :

- **Couverture du test :** Quelle est la proportion de code exécutée par un test d'unité ou de régression ? Ceci est mesuré par le nombre de lignes, le nombre de fonctions et le nombre de branches de contrôle qui sont exercées par les tests.
- **Testabilité :** Un aspect souvent négligé ou simplement ignoré du développement de code, la testabilité est la capacité à déclencher une ligne de code ou une condition de branchement spécifique.
- **Portabilité :** L'application peut-elle fonctionner sur des machines 32 et 64 bits ? Devrait-il fonctionner sur un téléphone portable ? Est-ce qu'il fonctionne sur plusieurs systèmes d'exploitation (par exemple, Windows, Linux, Mac OS X, Solaris, iOS, Android, RIM) ? Fonctionne-t-il correctement sur tous les navigateurs Web (IE, Firefox, Chrome, Safari, Opera) ?
- **Maintenabilité :** Est-il facile de déboguer le code ? à quelle vitesse est-ce pour fournir une solution ? Avec quelle rapidité un nouveau développeur peut-il comprendre le

code ? La maintenabilité est un aspect très important, assez difficile à quantifier. La maintenabilité est augmentée avec une bonne testabilité et un design flexible.

- **Lisibilité :** Il s'agit de savoir comment mettre le code facile à lire. Des directives sont établies pour unifier le style du code, de sorte qu'un développeur puisse facilement lire le code écrit par un autre développeur.
- **Evolutivité :** Est-ce facile d'étendre une fonctionnalité ? Ou pour en ajouter une nouvelle ? Ou augmenter la taille du cluster sur lequel s'exécute l'application ? Encore une fois, c'est une question d'architecture logicielle et d'anticipation des besoins futurs.

La qualité du logiciel est le résultat de l'expérience et l'appréciation globale de l'utilisateur final, elle est considérée comme la base du succès d'un produit.

3. Processus de test

Le processus de test est composé de plusieurs activités. Ce processus commence à partir de la planification des tests, puis la conception des cas de test, la préparation de l'exécution et l'évaluation du statut jusqu'à la clôture du test [8]. La figure suivante illustre ce processus :

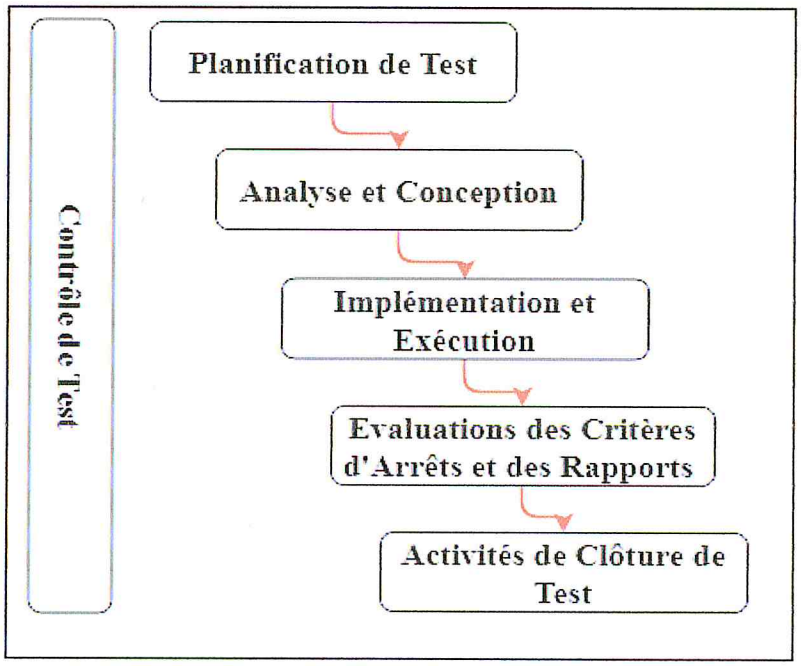

Figure I.2 : Les phases fondamentales du processus de test logiciel

Chaque phase du processus de test comporte un ensemble de tâches :

- **Planification de test :** Lors de la planification des tests, il faut bien comprendre les buts et les objectifs des clients, des parties prenantes[2] et du projet, ainsi que les risques que les tests sont censés de résoudre [9]. La planification de test comporte les tâches principales suivantes :
  - Déterminer la portée et les risques et identifier les objectifs du test.
  - Déterminer l'approche de test.
  - Implémenter la politique de test et / ou la stratégie de test (la stratégie de test est un schéma qui décrit la partie test du cycle de développement logiciel et cela comprend les objectifs de test, la méthode de test, le temps total et les ressources nécessaires pour le projet et les environnements de test).
  - Déterminer les ressources de test requises, telles que les environnements de test, les ordinateurs, etc.
  - Planifier l'analyse des tests et les tâches de conception, tester la mise en œuvre, l'exécution et l'évaluation.
  - Déterminer les critères d'arrêts, et pour cela nous devons définir des critères tels que les critères de couverture (les critères de couverture sont le volume d'instructions dans le logiciel qui doivent être exécutées lors des tests).

- **Contrôle de test :** Le contrôle de test est une activité en continuité. Il consiste à comparer les progrès réels par rapport aux progrès prévus, et faire un rapport au gestionnaire de projet et au client sur l'état actuel des tests, y compris les changements ou les écarts par rapport au plan [9]. Le contrôle de test comporte les tâches principales suivantes :
  - Mesurer et analyser les résultats des examens et des tests.
  - Surveiller et documenter les progrès de test de couverture et les critères d'arrêts.
  - Fournir des informations sur les tests.
  - Initier des actions correctives.
  - Prendre des décisions.

- **Conception et analyse :** L'analyse et la conception des tests sont les activités dans lesquelles les objectifs généraux de test sont transformés en conditions de test tangibles

[2] **Partie prenante** est une personne ou une organisation qui a un intérêt légitime dans un projet ou une entité.

et en conceptions de test [9]. Cette phase de test comporte les tâches principales suivantes :

- Réviser la base de test (la base de test est l'information dont nous avons besoin pour démarrer l'analyse de test et créer nos propres cas de test. Il s'agit d'une documentation sur laquelle sont basés les tests comme les spécifications de conception, l'analyse des risques, l'architecture et les interfaces. Nous pouvons utiliser les documents de base de test pour comprendre ce que le système devrait faire une fois construit).
- Identifier les conditions de test.
- Concevoir les tests.
- Évaluer la testabilité du système.
- Concevoir l'environnement de test et identifier l'infrastructure et les outils requis.

❖ **Implémentation et exécution :** Lors de cette phase, les conditions de test sont transformées en cas de test et en testware[3] et aussi une configuration de l'environnement de test est nécessaire [9]. L'implémentation et l'exécution de test ont les tâches principales suivantes :

- Développer et prioriser les cas de test avec des données de test différents pour chaque cas.
- Automatiser certains tests en utilisant un harnais de test et des scripts de tests (un harnais de test est un ensemble de logiciels et de données de test permettant de tester une portion de programme en l'exécutant dans différentes conditions et en surveillant son comportement et ses résultats).
- Créer des suites de tests[4] à partir des scénarios de test pour une exécution efficace.
- Implémenter et vérifier l'environnement de test.
- Journaliser les résultats de l'exécution du test et enregistrer les identités et les versions du logiciel testé. Le journal de test est utilisé pour la piste d'audit.
- Comparez les résultats réels avec les résultats attendus.

[3] **Testware** est un terme qui désigne tous les outils qui servent à tester un logiciel
[4] **Suite de test** est une collection de cas de test qui sont utilisés pour tester un programme.

- Lorsqu'il y a des différences entre les résultats réels et prévus, signaler les écarts comme des incidents.

❖ **Evaluation des critères d'arrêts et des rapports :** L'évaluation des critères d'arrêts est l'activité où l'exécution du test est évaluée par rapport aux objectifs définis [9]. L'évaluation des critères d'arrêts comporte les tâches principales suivantes :

- Vérifier les journaux de test par rapport aux critères d'arrêts spécifiés dans la planification de test.
- Vérifier si d'autres tests sont nécessaires ou si les critères d'arrêts spécifiés doivent être modifiés.
- Rédiger un rapport de synthèse de test pour les parties prenantes.

❖ **Activités de clôture de test :** Au cours des activités de clôture des tests, une collecte des données est faite à partir des activités de test complétées afin de consolider l'expérience [9]. L'activité de clôture de test comporte les tâches principales suivantes :

- Vérifier les produits planifiés pour la livraison sont effectivement livrés et s'assurer que tous les rapports d'incidents ont été résolus.
- Finaliser et archiver les testware, tels que les scripts, l'environnement de test et toute autre infrastructure de test, pour une réutilisation ultérieure.
- Donner le testware à l'organisme de maintenance qui supportera le logiciel et apportera des corrections des bugs[5] ou des modifications, pour les tests de confirmation et les tests de régression.
- Évaluer comment les tests ont été effectués et analyser les leçons tirées de ces tests pour améliorer les versions et les projets futurs.

4. Classification des tests

L'objectif de la classification des tests est d'avoir une bonne structuration et organisation des tests afin de d'offrir une meilleure expérience pour l'utilisateur et de délivrer un produit conforme aux spécifications demandées par les clients. La classification des tests est axée selon trois perspectives illustrées dans la figure suivante :

[5] **Bug** désigne une erreur de conception ou d'écriture dans un programme informatique

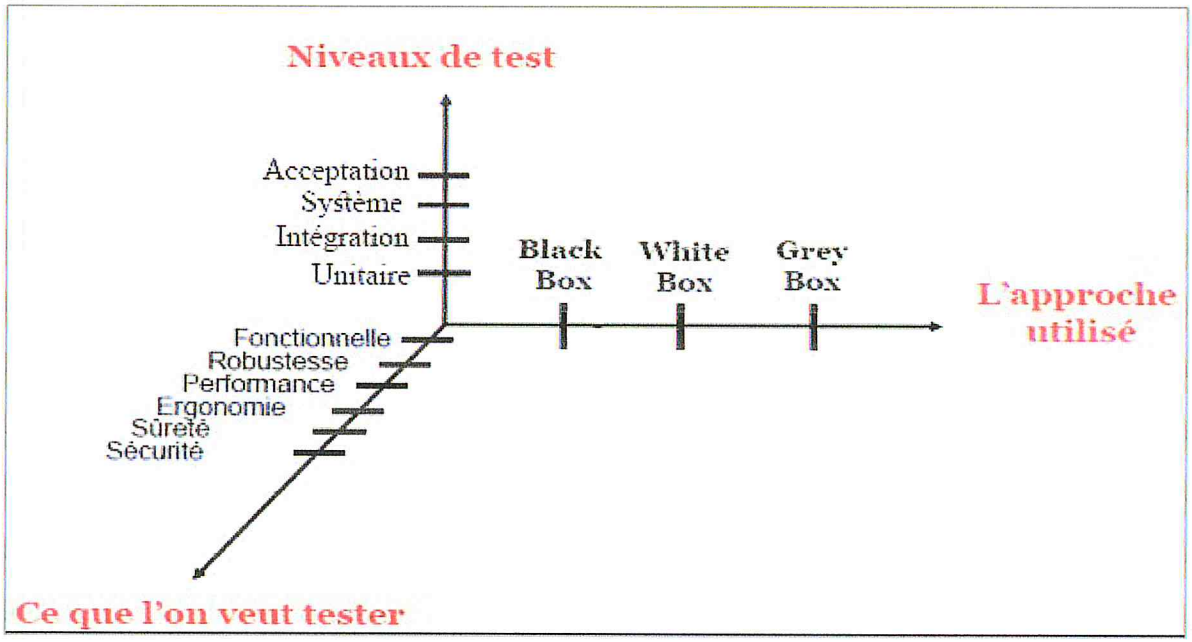

Figure I.3 : La classification des tests du logiciel [40]

4.1. Niveaux de test

Une compréhension et une définition approfondies des différents niveaux de test permettront d'identifier les zones de faiblesse et d'éviter les chevauchements et les répétitions dans chaque phase du cycle de développement du logiciel [10]. Le tableau suivant décrit les différents niveaux de test par ordre ascendant :

N°	Niveau de test	Définition
1	Test unitaire/composant	C'est le plus bas niveau de test principalement effectué par le développeur pour but d'isoler chaque partie du programme et de montrer que les parties individuelles sont correctes en termes d'exigences et de fonctionnalités.
2	Test d'intégration	Ce niveau permet de tester la communication entre différents modules afin de s'assurer que les données circulent correctement entre les différents composants. Ceci est fait suivant une approche ascendante[6] ou une approche descendante[7].

[6] **Approche ascendante** désigne un test qui commence par des tests unitaires, suivis par des tests de combinaisons progressivement plus élevées d'unités appelées modules ou builds.

[7] **Approche descendante** désigne un test dont les modules de plus haut niveau sont testés en premier et progressivement, les modules de niveau inférieur sont testés par la suite.

N°	Niveau de test	Définition
3	Test du système	Le système global est testé pour s'assurer qu'il se comporte ou fonctionne comme prévu et comme spécifié dans le cahier de charge.
4	Test d'acceptation	Les tests de pré-acceptation sont principalement connus sous le nom de tests alpha et bêta afin de s'assurer que les clients sont en mesure d'exécuter les fonctionnalités prévues et de prendre des informations en retour pour améliorer la qualité de logiciel.

Tableau I.2 : Les principaux niveaux de test logiciel [11]

4.2. Types de test

Plusieurs types de test existent tel que :

- **Le test fonctionnel :** Le test fonctionnel est une approche de test qui vérifie que chaque fonction de l'application fonctionne conformément aux spécifications demandés sans connaitre ou analyser les détails de son code source. La figure ci-dessous illustre cet aspect :

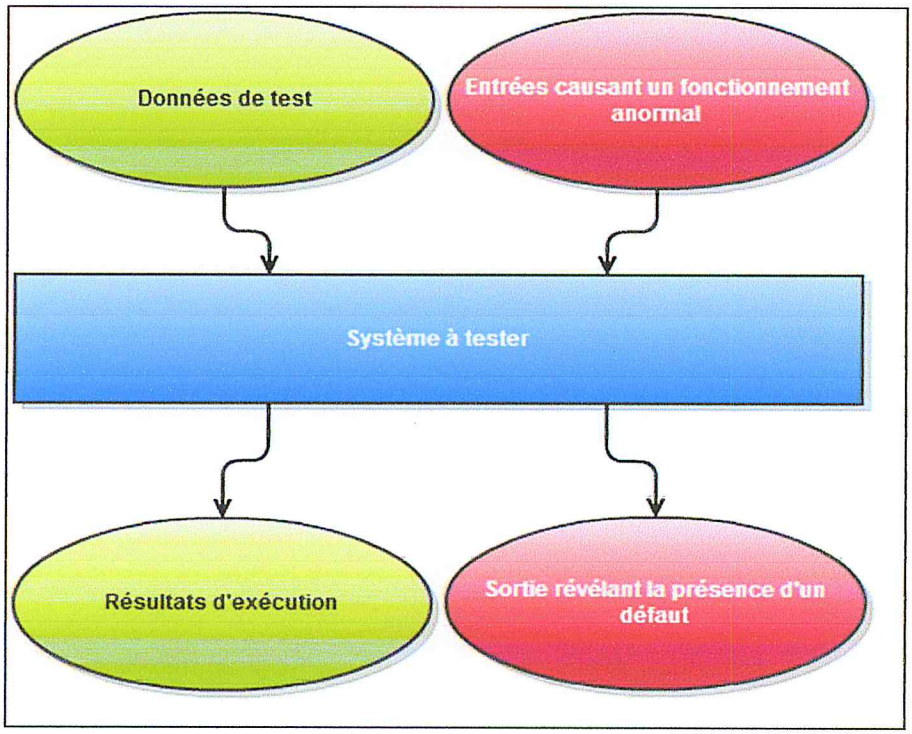

Figure I.4 : Le déroulement d'un test fonctionnel [41]

- **Le test non-fonctionnel :** Le test non-fonctionnel est un type de test permettant de vérifier des aspects non fonctionnels d'une application logicielle. Il est conçu pour tester la disponibilité d'un système en fonction de paramètres qui ne sont jamais traités par des tests fonctionnels. Il s'agit de vérifier des caractéristiques telles que :
  - **Test de performance :** Les tests de performance sont conçus pour mesurer la rapidité avec laquelle le programme exécute une tâche donnée. L'objectif principal est de déterminer si la vitesse de traitement est acceptable dans toutes les parties du programme [12].
  - **Test de sécurité :** Les tests de sécurité déterminent si le programme et ses données sont protégés contre les accès hostiles. Une distinction est faite entre les attaques externes, via le système de fichiers par exemple, et les attaques internes, dans lesquelles les utilisateurs tentent d'accéder à l'information sans autorisation appropriée [12].
  - **Test de fumée (Smoke) :** Les tests de fumée révèlent des défaillances de base suffisamment graves pour empêcher les tests de validation, comme par exemple la vérification si le programme démarre correctement ou s'il est capable d'afficher des messages [12].
  - **Test de stress :** Le test de stress est une forme de test de performance qui mesure la vitesse de traitement du programme lorsque la charge du système augmente. La charge du système signifie généralement le nombre d'utilisateurs de programmes concurrents (clients). Il peut également se référer au volume de données à traiter, au nombre d'enregistrements dans une base de données, à la fréquence des messages entrants ou à des facteurs similaires [12].
  - **Test basé sur le code :** Le test basé sur le code traite directement avec le code source[8], il doit vérifier si le code du programme génère des erreurs d'exécution. Les tests basés sur le code sont également utilisé pour évaluer l'efficacité des algorithmes [12].

[8] **Code source** désigne une liste d'instructions écrite dans un langage de programmation et qui peut être converti pour constituer un programme exécutable.

- **Le test structurel :** Le test structurel est une approche dans laquelle les tests sont dérivés de la connaissance de la structure du logiciel ou de son implémentation interne (code source).

Chaque type de test admet une ou plusieurs méthodes pour tester et évaluer la qualité et le rendement du logiciel afin de bien satisfaire les besoins des clients.

4.2.1. Méthodes de test du logiciel

Plusieurs méthodes existent qui se base sur différents types de test :

- **La Méthode de test en Black-Box :** Avec la méthode de test en Black-Box, le monde extérieur entre en contact avec l'élément de test (un programme ou un composant de programme) uniquement via une interface spécifiée. Cette interface peut être l'interface de l'application, une interface de module interne ou la description INPUT / OUTPUT d'un traitement par lots. Les tests Black-Box vérifient si les définitions d'interface sont respectées dans toutes les situations. Ils testent si le produit est conforme à toutes les exigences fixés [12]. la figure ci-dessous présente la vue globale de la méthode de test en Black-Box :

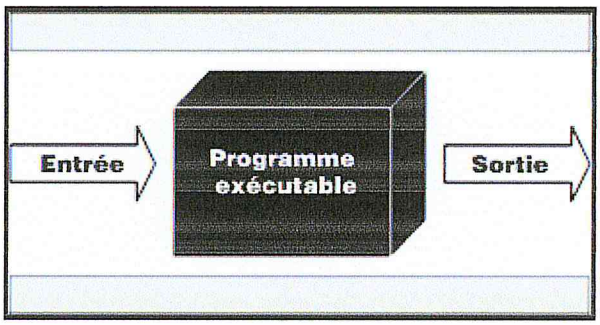

Figure I.5 : La méthode de Black-Box

- **La méthode de test en White-Box :** Dans la méthode de test en White-Box, le fonctionnement interne (code source) de l'application est connue. Les cas de test sont créés en fonction de ces connaissances. Les tests en White-Box sont donc des tests de développeur, ils s'assurent que chaque fonction implémentée est exécutée au moins une fois et vérifie le bon comportement. L'examen des résultats des tests en White-Box peut être fait en gardant à l'esprit les spécifications du système [12]. La figure ci-dessous présente la vue globale de la méthode de test en White-Box :

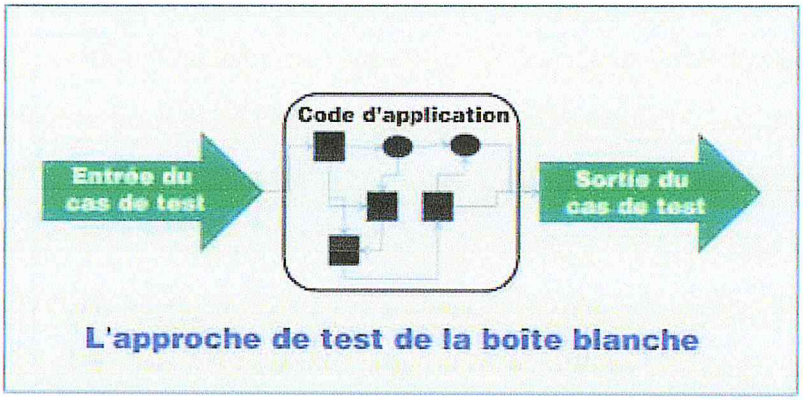

Figure I.6 : La méthode de White-Box

- **La méthode de test en Grey-Box :** La méthode de test en Grey-Box c'est un mélange des deux méthodes de test (Black-Box et White-Box), elle permet de tester l'application en ayant une connaissance limitée du fonctionnement interne (code source) d'une application. Contrairement aux tests en Black-Box, qui sont limités aux différentes interfaces (utilisateur, module interne) de l'application. Dans les tests en Grey-Box, le testeur a accès aux documents de conception, à la base de données et les algorithmes utilisés [13]. La figure ci-dessous présente la vue globale de la méthode de test en Grey-Box :

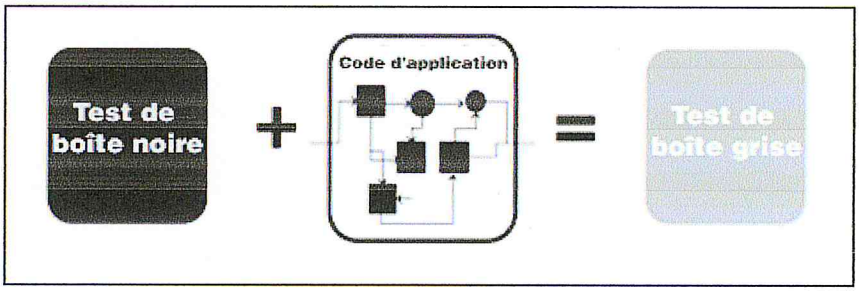

Figure I.7 : La méthode de Grey-Box

- **La méthode de test statistique :** La méthode de test statistique implique d'exercer une portion du logiciel en lui fournissant des valeurs d'entrée sélectionnées de manière aléatoire en fonction d'une distribution de probabilité définie sur son domaine d'entrée. Le but de cette méthode est de déterminer la fiabilité du logiciel et de révéler les bugs [14], elle comporte plusieurs tâches :
- Le logiciel est testé avec des données de test qui modélisent statistiquement l'environnement de travail.
- Les erreurs sont collationnées et analysés.

- ❖ À partir des données calculées, une estimation du taux d'échec du programme est calculée.

4.2.2. Comparaison des méthodes de test (Box Approach)

Le tableau suivant liste les points qui différencient les tests en Black-Box, les tests en Grey-Box et les tests en White-Box [13] :

Black-Box	Grey-Box	White-Box
Points de différences		
Le fonctionnement interne (code source) d'une application n'a pas besoin d'être connu.	Le testeur a une connaissance limitée du fonctionnement interne de l'application.	Le testeur a une connaissance complète du fonctionnement interne de l'application.
Également connu sous le nom de test fonctionnel.	Également connu sous le nom de test translucide.	Également connu sous le nom de test structurel.
Effectué par les utilisateurs finaux[9] et aussi par les testeurs et les développeurs.	Effectué par les utilisateurs finaux et aussi par les testeurs et les développeurs.	Effectué par les testeurs et les développeurs.
Les tests sont basés sur l'extérieur de l'application. Le comportement interne de l'application est inconnu.	Les tests sont effectués sur la base de diagrammes de flux de données.	Les fonctionnements internes sont entièrement connus et le testeur peut concevoir des données de test en conséquence.
C'est exhaustif et prend peu de temps.	Partiellement long et exhaustif.	La méthode de test la plus exhaustif et la plus long.
Ne convient pas pour les tests d'algorithmes.	Ne convient pas pour les tests d'algorithmes.	Convient pour les tests d'algorithmes.

[9] **L'utilisateur final** désigne la personne qui va utiliser ledit logiciel.

Black-Box	Grey-Box	White-Box
Points de différences		
Cela ne peut être fait que par une méthode d'essai et d'erreur.	Les domaines de données et les limites internes peuvent être testés, s'ils sont connus.	Les domaines de données et les limites internes peuvent être mieux testés.

Tableau I.3 : Tableau comparatif des méthodes de test en Box Approach [13]

5. Conclusion

L'étude préliminaire du test et qualité du logiciel nous a permis de bien comprendre le fonctionnement de ce processus et de savoir les différentes méthodes et types de test. Cela dans le but d'apercevoir une vue globale sur le concept du test logiciel et son importance dans l'assurance de la qualité du logiciel. Le chapitre suivant sera consacré à l'automatisation du test logiciel.

Chapitre II

Automatisation du Test Logiciel

1. Introduction

Après avoir étudié le processus de test et qualité du logiciel et comprendre le fonctionnement de ce dernier, nous allons présenter dans ce chapitre le processus d'automatisation du test logiciel. Pour ce faire on va commencer en premier lieu par des généralités sur l'automatisation du test. Après, on va présenter l'automatisation des activités du processus de test, puis on va faire une étude comparative des outils d'automatisation du test. Ensuite, on va décrire les différentes techniques de Scripting de test automatisé. Et on finira par présenter les recherches académiques sur la génération automatiques des cas de test ainsi que notre méthode proposée.

2. Généralités sur l'automatisation du test

2.1. Définition de l'automatisation

L'automatisation est l'exécution automatique de tâches sans interférence périodique. Elle vise à minimiser et à éliminer progressivement l'intervention humaine. Un logiciel qui fonctionne seul ou un dispositif électronique qui fonctionne de manière autonome sont des exemples d'automatisation. L'automatisation simplifie les tâches compliquées en les réduisant à une seule instance. Il suffit d'un appui sur un bouton ou une commande simple mettra en mouvement une chaîne d'événements vers un objectif spécifié [15].

2.2. Automatisation du test

L'automatisation du test est une méthode de test logiciel qui utilise des outils pour contrôler l'exécution des tests, puis compare les résultats des tests obtenus avec les résultats attendus. Tout cela est fait automatiquement avec peu ou pas d'intervention d'un testeur. L'automatisation du test est considérée comme complémentaire du test manuel [16].

2.3. Objectif de l'automatisation du test

Tester manuellement toutes les fonctionnalités d'un logiciel à chaque fois et avec l'ajout de nouvelles fonctionnalités se relève fastidieux et prend beaucoup de temps. L'objectif d'automatisation du test est de réduire le temps requis pour l'exécution des tests, minimiser l'investissement sur les ressources humaines et augmenter l'efficience du test. En d'autres termes, l'automatisation du test permet de combler les lacunes du test manuel [16].

2.4. Types de test logiciel

Il existe deux types de test logiciel que les testeurs peuvent utiliser pour assurer que les modifications ou les changements apportés au logiciel fonctionnent comme prévu. Ces deux types de test sont :

- **Le test manuel :** Le test manuel est un processus de découverte des défauts dans un logiciel. Dans cette méthode, le testeur joue un rôle d'un utilisateur final et vérifie que toutes les fonctionnalités de l'application fonctionnent correctement. Le testeur exécute manuellement les cas de test sans utiliser d'outils d'automatisation [17].
- **Le test automatique :** Le test automatique est une technique dans laquelle le testeur écrit ses propres scripts[10] et utilise un outil approprié pour tester un logiciel. Aucune intervention manuelle n'est requise lors de l'exécution d'une suite de tests automatisée. Le test automatique est beaucoup plus robuste et fiable que le test manuel, mais la qualité des tests automatisés dépend de la qualité des scripts de test écrite [18]. Le test automatique peut être effectué en quatre étapes :
  1) Préparation du plan de test ou création des cas de test.
  2) Sélection préliminaire de l'outil de test.
  3) Écriture des scripts de test.
  4) Exécution des scripts de test.

2.5. Comparaison entre le test manuel et automatisé

Les tests manuels et les tests automatisés couvrent différents catégories de test. Dans chaque catégorie de test, des méthodes spécifiques sont disponibles, telles que les tests en Black-Box, les tests en White-Box, les tests de performance...etc. Certaines de ces méthodes sont mieux adaptées aux tests manuels, d'autres sont mieux réalisées grâce à l'automatisation. Le tableau ci-dessous nous donne une brève comparaison entre ces deux types de test [19] :

Test manuel	Test automatisé
☒ Les tests manuels ne sont pas toujours précis, ils sont souvent sujet à l'erreur humaine, donc ils sont moins fiables.	☑ Les tests automatisés sont plus fiables, car ils sont effectués par des outils et / ou des scripts.

[10] **Script** désigne un programme écrit dans un langage interprété pour exécuter une tâche particulière

Test manuel	Test automatisé
☒ Les tests manuels prennent beaucoup de temps et exigent des ressources humaines.	☑ Les tests automatisés sont exécutés par des outils logiciels, ils sont beaucoup plus rapides qu'une approche manuelle.
☑ L'investissement est requis pour les ressources humaines, mais il est moins coûteux.	☒ Un investissement est requis pour les outils de test, mais il est plus coûteux.
☒ Le test sur une machine différente avec une combinaison de différents systèmes d'exploitation n'est pas possible, en même temps. Pour exécuter une telle tâche, différents testeurs sont requis.	☑ Les tests d'automatisation peuvent être effectués sur différentes machines avec différents systèmes d'exploitation, en même temps.
☒ Les tests manuels ne sont pratiques que lorsque les tests sont exécutés une ou deux fois, et une répétition fréquente n'est pas nécessaire.	☑ Le test automatisé est une option pratique lorsque les scénarios de test sont exécutés de manière répétée sur une longue période.
☑ Il est très utile dans le test de l'interface utilisateur.	☒ Parfois, il n'est pas utile dans les tests d'interface utilisateur.

Tableau II.1 : Tableau comparatif entre le test manuel et automatique [19]

3. Automatisation des activités du processus de test

Les gestionnaires des projets et les développeurs de logiciels font face aujourd'hui à un défi qui est de développer des applications dans un calendrier de plus en plus réduit et avec un minimum de ressources. Dans le cadre de leurs efforts pour en faire plus avec moins, les organisations veulent tester le logiciel de manière adéquate, mais aussi rapidement et minutieusement que possible. Pour atteindre cet objectif, les organisations se tournent vers les tests automatisés. Ces tests automatisés doivent être effectués selon une méthodologie qui fournit les phases nécessaires à la réalisation de cet objectif, la figure ci-dessous illustre la méthodologie des tests automatisés [20] :

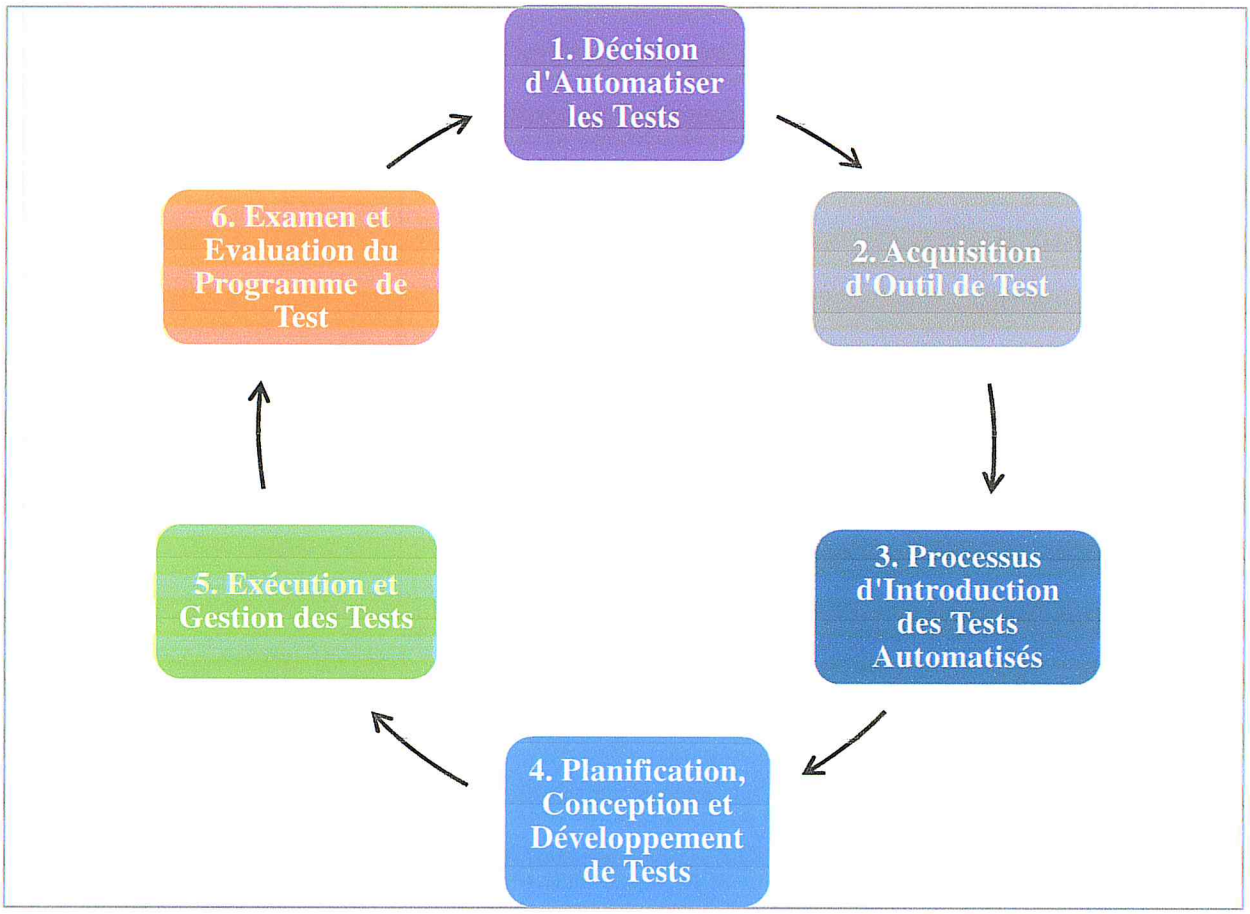

Figure II.1 : Méthodologie du cycle de vie de test automatisé [20]

La méthodologie du cycle de vie de test automatisé comprend six phases principales décrites comme suit :

1) **La décision d'automatiser les tests** : Au cours de cette phase, il est important que l'équipe de test gère les attentes des tests automatisés et décrive les avantages potentiels lorsqu'ils sont correctement mise en œuvre. De plus, une proposition d'un outil d'automatisation du test doit être présentée. La décision d'automatisation de test est basée sur deux aspects [20] :
    1. **Surmonter les fausses attentes pour les tests automatisés :** Il a été prouvé que les tests automatisés sont précieux et peuvent produire un retour sur investissement, mais il n'y a pas toujours un retour sur investissement immédiat. Il est important que certaines fausses idées qui persistent dans l'industrie du logiciel soient traitées et que l'utopie[11] des tests automatisés soit gérée [20].

[11] **Utopie** désigne un projet dont la réalisation est imaginaire et hors de portée.

2. **Décrire les avantages potentiels des tests automatisés :** Le testeur doit évaluer si les avantages potentiels correspondent aux critères d'amélioration requis et si la poursuite des tests automatisés sur le projet est encore logique, pour les besoins de l'organisation. Il y a plusieurs avantages significatifs du test automatisé qui comprennent : la production d'un système fiable, l'amélioration de la qualité des tests avec un effort minimal et un temps d'exécution réduit [20].

2) **Acquisition d'outil de test :** L'acquisition d'outil de test suit différentes étapes [20] :

- Identification des types d'outils qui conviennent à l'environnement du système de l'organisation, en tenant compte des éléments suivants :
  - Le groupe / département qui utilisera l'outil.
  - Le budget alloué à l'acquisition de l'outil.
  - les fonctions les plus / les moins importantes de l'outil.
- Le choix de type d'outil en fonction du stade courant du cycle de vie du test logiciel.
- Évaluation des différents outils de la catégorie d'outils sélectionnée.
- Après l'évaluation, un rapport devrait être préparé, pour enfin choisir l'outil correspondant aux besoins de l'organisation.

Le responsable des tests doit rechercher des outils déjà disponibles. Si le responsable ne trouve pas d'outil approprié, il prend un outil disponible et le personnaliser en fonction des besoins. Si ce n'est pas possible alors il va développer un nouvel outil.

3) **Processus d'introduction des tests automatisés :** Cette phase décrit deux étapes nécessaires pour introduire les tests automatisés dans un nouveau projet, ces étapes sont résumées ci-dessous [20] :

1. **Analyse du processus de test :** L'analyse du processus de test garantit qu'un processus de test global et une stratégie sont mis en place et sont modifiés, si nécessaire, pour permettre l'introduction réussite d'un test automatisé. Ici, les objectifs, les buts et les stratégies de test doivent être définis et le processus de test doit être documenté et communiqué à l'équipe de test [20].
2. **Considération de l'outil de test :** La considération de l'outil de test décrit les étapes nécessaires pour vérifier que l'outil de test répond aux besoins de test spécifiques du projet et fournit des directives pour déterminer s'il est possible

d'introduire un outil de test automatisé, selon le calendrier du projet et d'autres critères comme tenir en compte les exigences de test du projet, la disponibilité de l'environnement de test et les fonctionnalités de l'application sous test. Dans cette étape on cherche également à s'assurer que l'expertise des outils de test automatisés est en place et que les membres de l'équipe de test comprennent leurs rôles [20].

4) **La Planification, conception et développement de test :** Cette phase est importante car elle inclut l'identification des procédures de test, la définition des tests et le développement des tests. Elle comporte trois étapes [20] :
    1. **Planification de test :** La planification de test contient beaucoup d'informations, y compris une grande partie des exigences de documentation de test pour le projet. La planification de test décrira les rôles et les responsabilités de l'équipe de test, le calendrier de test du projet, les activités de conception des tests, la préparation de l'environnement de test, les risques et les contingences du test [20].
    2. **Conception de test :** La conception de test répond à la nécessité de définir le nombre des tests à effectuer, les manières d'aborder le test (chemins, fonctions) et les conditions de test [20].
    3. **Développement de test :** Cette phase implique le développement des procédures de test automatisé pour qu'elles soient maintenables, réutilisables, simples et robustes [20].

5) **Exécution et gestion des tests :** Dans cette phase, l'équipe de test a déjà abordé la conception et le développement des tests. Les tests sont maintenant prêts à être exécutées [20].

6) **Examen et évaluation du programme de test :** Les activités d'examen et d'évaluation du programme de test doivent être menées tout au long du cycle de vie des tests, afin de permettre une amélioration continue de l'application et réduire les défauts le plus tôt possible. Tout au long du cycle de vie des tests et après les activités d'exécution des tests, les activités finales d'évaluation doivent être menées pour permettre l'amélioration des processus et déterminer si le processus de l'automatisation de test a été bénéfique et bien déroulé [20].

En résumé la méthodologie du cycle de vie de test automatisé est une méthodologie structurée qui vise à assurer une implémentation réussie des tests automatisés.

4. Étude comparative des outils d'automatisation de test logiciel

Les outils d'automatisation du test jouent un rôle important dans l'efficacité et la valeur des tests. Comme il existe une grande variété d'outils, la sélection de l'outil approprié dépend de plusieurs facteurs. Lors de la sélection des outils de test, beaucoup de critères doivent être prises en comptes afin de s'assurer que l'outil choisi répond bien aux besoins [21]. Ces critères sont décrits comme suit :

- Le type d'application à tester (Desktop / Web / Mobile).
- Le type de test (unitaire, sécurité, performance … etc.).
- Les différentes fonctionnalités disponibles dans l'outil.
- La facilité d'utilisation.
- Le coût de l'outil lui-même, quand il s'agit d'un outil propriétaire, il est nécessaire d'acheter des licences pour les testeurs.

Tous ces critères doivent être vérifiés pour bénéficier d'une utilisation maximale de l'outil et conduire des tests appropriés. Il existe deux types de licence pour les outils d'automatisation du test, qui sont :

- Les outils sous licence libre (Open-Source).
- Les outils sous licence payante (propriétaires).

4.1. Outils Open-Source

L'Open Source est une méthode d'ingénierie logicielle[12] qui consiste à développer un logiciel, et de laisser en libre accès le code source produit. Ce code source peut alors être exploité par les développeurs et les entreprises souhaitant soit l'adapter à leurs besoins métiers, soit affiner son intégration avec leur système d'information [22]. Plusieurs outils de test Open-Source sont disponibles et classifiés selon différents types de test tels que :

[12] **L'ingénierie logicielle** désigne de recouvrir d'une façon générale l'ensemble des prestations liées à l'intégration de composants logiciels et produits applicatifs spécialisés dans un projet global.

- **Le test fonctionnel :** Les meilleurs outils Open-Source de test fonctionnel sont :
    - **Selenium :** Selenium est un outil qui permet d'automatiser les tests effectués sur des applications web ou des applications mobiles à travers des Framework[13] comme Selendroid, Appium... etc. Il comporte deux composants principaux :
        - **Selenium IDE :** Il s'agit d'un environnement de développement conçu pour permettre aux testeurs et aux développeurs d'enregistrer leurs interactions avec le navigateur web permettant par la suite de rejouer un scénario d'interactions pour simuler un processus fonctionnel à tester.
        - **Selenium WebDriver :** Selenium WebDriver est une collection d'interface de programmation applicative[14] (API) Open-Source utilisées pour automatiser les tests pour les applications web.
    - **Appium :** Appium est un outil d'automatisation de test pour les applications mobiles. Il permet de tester tous les trois types d'applications mobiles : native, hybride et web. Il permet également d'exécuter les tests automatisés sur des périphériques, des émulateurs ou des simulateurs.
    - **Calabash :** Calabash est un outil de test d'acceptation (fonctionnel) qui permet d'écrire et d'exécuter des tests automatisés pour les applications iOS et Android. Calabash fonctionne en activant les interactions automatiques de l'interface utilisateur dans une application, par exemple en appuyant sur des boutons, en saisissant du texte, en validant des réponses … etc.

- **Le test de performance et de monté en charge :** Les meilleurs outils Open-Source de test de performance et de monté en charge sont :
    - **Apache JMeter :** Apache JMeter est un outil conçu principalement pour les applications Web, il permet de tester la monté en charge et mesurer les performances.

[13] **Framework** désigne un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application pour but de simplifier le travail des développeurs informatique.
[14] **Interface de programmation applicative** désigne un ensemble de fonctions qui permettent à un développeur d'utiliser simplement une application dans son programme.

- **WebLOAD :** WebLOAD est un outil de test de monté en charge et d'analyse de performance. C'est un véritable outil puissant et efficace pour le test des applications Web ainsi que pour le test des applications mobiles.

Il existe d'autres outils Open-Source disponibles dédier pour d'autres types de test, ça reste aux testeurs de choisir le meilleur outil qui convient à leurs besoins.

4.2. Outils propriétaires

Un outil ou un logiciel propriétaire est développé la plupart du temps par une entreprise. Il est distribué uniquement sous forme binaire (sans code source). Les détails de sa conception ou développement restent inconnu pour l'utilisateur. Souvent, il est disponible sous une licence d'utilisation assez restrictive et les utilisateurs non pas le droit ni de le modifier, distribuer ou étudier son fonctionnement sans autorisation de son propriétaire [23]. Il existe plusieurs outils de test propriétaires classifiés selon différents types de test tels que :

- **Le test fonctionnel :** Les meilleurs outils propriétaires de test fonctionnel sont :
  - **Eggplant Functional :** Eggplant Functional est un outil de test fonctionnel centré sur la perspective de l'utilisateur, il permet de tester toute application pouvant être contrôlée, comme les applications mobiles, Desktop ou Web. Il intègre de puissantes fonctionnalités comme la recherche d'image avancée et la reconnaissance d'image et de texte.
  - **Ranorex :** Ranorex est un outil puissant pour l'automatisation des tests. Il s'agit d'un Framework d'automatisation de test pour l'interface graphique de l'utilisateur, utilisé pour tester des applications Web, Desktop et mobiles. Ranorex prend en charge de nombreuses technologies telles que .NET, Java, HTML5, Flash, iOS, Android ... etc.
  - **TestComplete :** TestComplete est un outil d'automatisation de test fonctionnel qui permet de créer, gérer et exécuter des tests pour toute application Desktop, Web ou mobile. Il facilite pour tous les utilisateurs de créer des tests automatisés. Les tests peuvent être enregistrés, scriptés ou créés manuellement et ils sont utilisés pour la lecture automatique et la journalisation des erreurs.

- **Le test de performance et de monté en charge :** Les meilleurs outils propriétaires de test de performance et de monté en charge sont :

- **SilkPerformer :** Silk Performer est un outil de test de performance pour les applications Web et mobiles. Il garantit que les applications et les temps de disponibilité des serveurs sont maintenus face à l'utilisation maximale des clients.
- **StresStimulus :** StresStimulus est un outil de test de performance et de monté en charge pour les sites Web et les applications mobiles. Il détermine les performances et l'évolutivité des applications sous une charge de trafic importante. Il permet de surveiller un serveur en temps réel afin de collecter des différentes informations sur les performances pour identifier les goulets d'étranglement.

Il existe beaucoup d'outils propriétaires pour d'autres types de test, l'avantage de ces outils payants par rapport aux outils Open-Source, c'est les fonctionnalités variés et puissantes et l'aspect de sécurité.

4.3. Comparatif des outils d'automatisation de test

Plusieurs outils d'automatisation de test sont disponibles, certains d'eux sont gratuits, d'autres sont payants. La différence entre ces outils réside dans les fonctionnalités fournies, la sécurité de l'outil et l'efficacité des tests. Le tableau suivant illustre une comparaison entre les différents outils d'automatisation de test :

	Selenium	**TestComplete**	**Ranorex**	**Appium**
Plateforme de test	Multiplateforme	Windows	Multiplateforme	iOS, Android, Windows
Type d'applications	Web	Web, Desktop, Mobile	Web, Desktop, Mobile	Mobile
Langage de programmation supporté	Java, C#, Perl, Python, JavaScript, Ruby, PHP, R	JavaScript, Python, VBScript, JScript, Delphi, C++, C#	JavaScript, Python, PHP, Perl, Ruby, C++, C#, Java	Ruby, Python, Java, JavaScript, PHP, C#

	Selenium	TestComplete	Ranorex	Appium
Compétences en programmation	Nécessite des compétences avancées pour intégrer divers outils	Non requis	Non requis	Moyen
Courbe d'apprentissage	Difficile	Moyen	Moyen	Facile
Facilité d'utilisation	Difficulté dans l'installation et l'intégration des divers outils	Facile à installer et à utiliser	Facile à installer et à utiliser	Facile à installer et à utiliser
Type de test supporté	Test fonctionnel, test de régression	Test unitaire, test fonctionnel, test de régression, test de monté en charge, test de couverture	Test fonctionnel, test de régression, test de navigateur	Test fonctionnel
Simulateur / Emulateur	Oui	Oui	Oui	Oui
Documentation / Support de communauté	Beaucoup de documentation et une communauté active	Documentation moyenne et une communauté active	Documentation moyenne et une communauté active	Immense documentation et une large communauté active
Enregistrement / Lecture des scénarios	Oui	Oui	Oui	Non, nécessite un outil supplémentaire
Type de licence	Open-Source	Propriétaire	Propriétaire	Open-Source

Tableau II.2 : Tableau comparatif entre les outils d'automatisation de test

5. Techniques de Scripting de test automatisé

Les tests logiciels pour les entreprises deviennent une nécessité pour assurer une meilleure qualité de leurs produits. Pour assurer cette qualité, les entreprises se tournent toujours vers des techniques efficaces d'automatisation du test. Dans la poursuite de cet objectif, plusieurs recherches ont été consacrées aux techniques de Scripting de test automatisé. Parmi ces techniques, nous citons :

- **Linear Scripting Technique :** John Kent[15] explique l'idée derrière Linear Scripting Technique, qui s'articule sur de mettre l'outil de test en mode d'enregistrement tout en effectuant des actions sur l'application sous test. Le script enregistré et généré consiste en une série d'instructions de test utilisant le langage de programmation supporté par l'outil. Gerald Everett[16] a suggéré que les scripts linéaires sont créés en enregistrant les actions qu'un utilisateur effectue manuellement sur l'interface de l'application, puis en sauvegardant les actions de test en tant que script de test. Ces scripts de test peuvent ensuite être relus et exécuté à nouveau. Ainsi, Linear Scripting Technique est appelée Record / Playback [39], la figure ci-dessous illustre cette technique :

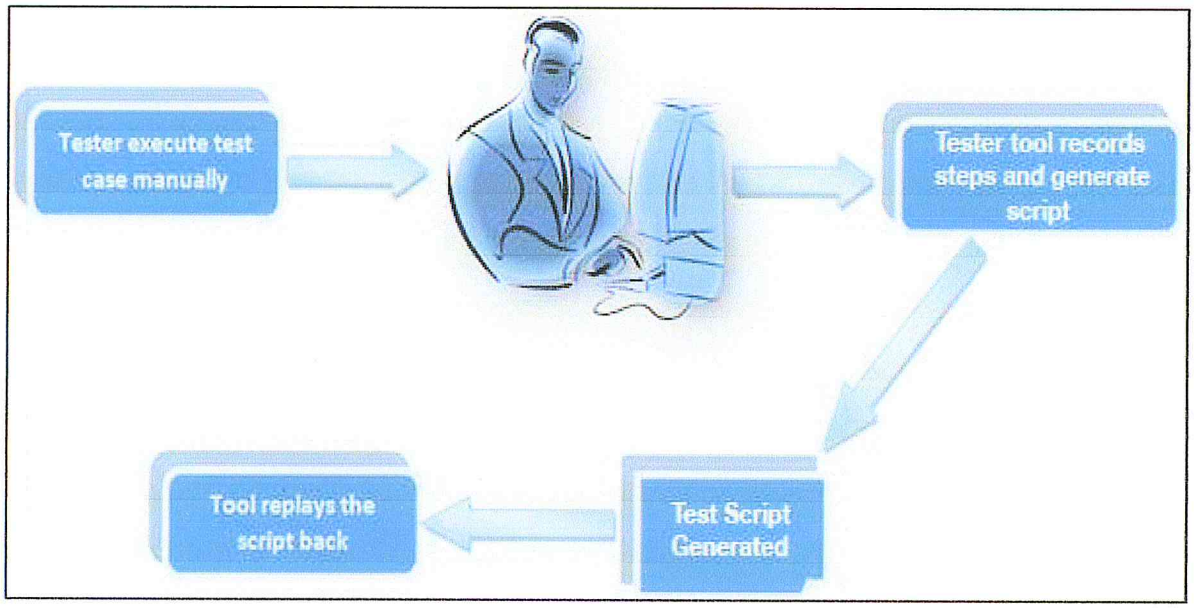

Figure II.2 : Linear Scripting Technique [39]

[15] **John Kent** est un consultant senior en tests de logiciels dans l'Université de Brighton (Royaume-Uni).
[16] **Gerald Everett** PhD. en informatique de l'Université du Texas à Austin (États-Unis), il est un expert en développement et test logiciel.

- **Structured Scripting Technique :** Cette technique utilise des instructions de programmation structurées, qui sont des structures de contrôle ou des structures d'appel. Les structures de contrôle sont utilisées pour contrôler les différents chemins dans le script de test (ex. if condition). Les structures d'appel sont utilisées pour diviser les grands scripts en scripts plus petits afin d'être plus maniables [39].

- **Shared Scripting Technique :** Cette technique permet de stocker des actions communes dans un seul endroit. Cela implique qu'un script commun peut être appelé par d'autres scripts. L'idée derrière les scripts partagés est de générer un script séparé qui exécute une tâche commune spécifique que les autres scripts devront peut-être exécuter plus tard [39], la figure suivante illustre cette technique :

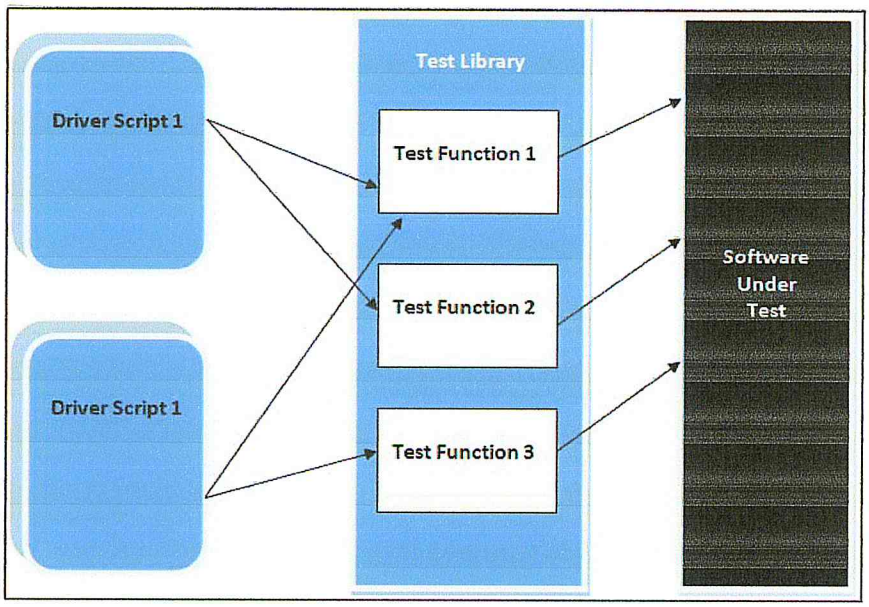

Figure II.3 : Shared Scripting Technique [39]

- **Data-Driven Scripting Technique :** Cette technique permet de stocker les données de test dans un fichier séparé au lieu d'être fortement couplées au script de test lui-même. Lors de l'exécution des tests, les données de test sont lues à partir d'un fichier de données externe. Data-Driven Scripting Technique propose une meilleure organisation des scripts de test et donc des coûts de maintenance plus faibles [39], la figure suivante illustre cette technique :

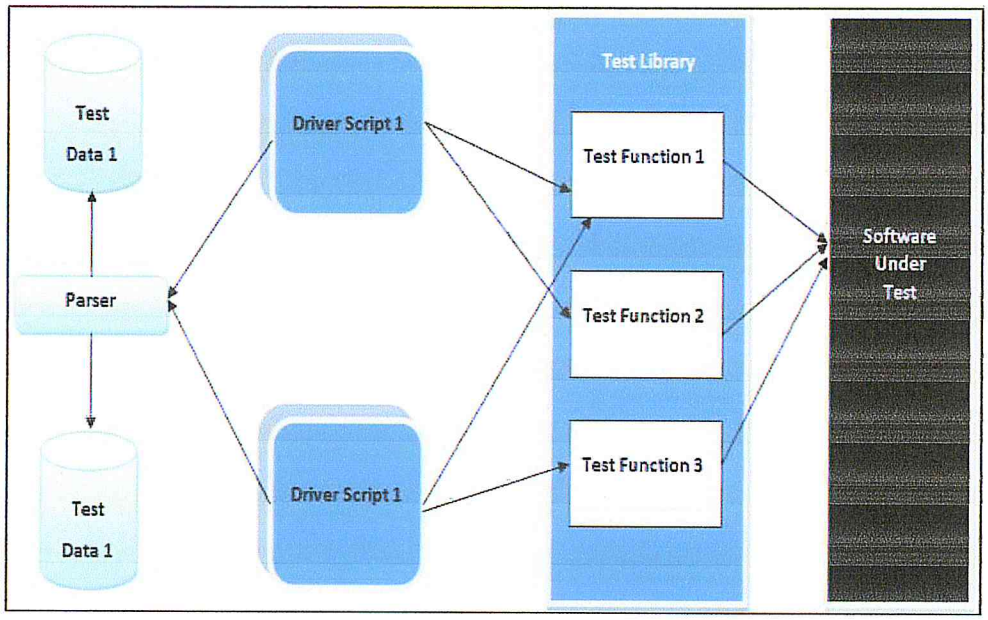

Figure II.4 : Data-Driven Scripting Technique [39]

- **Keyword-Driven Scripting Technique :** Cette technique est plus sophistiquée que Data-Driven Scripting Technique et elle est très similaire aux cas de test manuels. Les fonctions métier de l'application sous test sont stockées dans un tableau ainsi que les instructions pour chaque cas de test. Keyword-Driven Technique sépare non seulement les données de test, mais aussi les mots-clés spéciaux requis pour l'appelle de la fonction associé à ces mots dans un fichier externe (Excel). Le testeur peut créer un grand nombre de scripts de test en utilisant simplement des mots-clés prédéfinis [39]. La figure suivante illustre cette technique :

#	TYPE	KEYWORD	OPERATION	PARAMETER
1	Function	Runapp		http://in.yahoo.com/?p=us
2	Item	Username	SetValue	Username=rashmi
3	Item	Password	SetValue	Password=12345
4	Item	ok	Click	

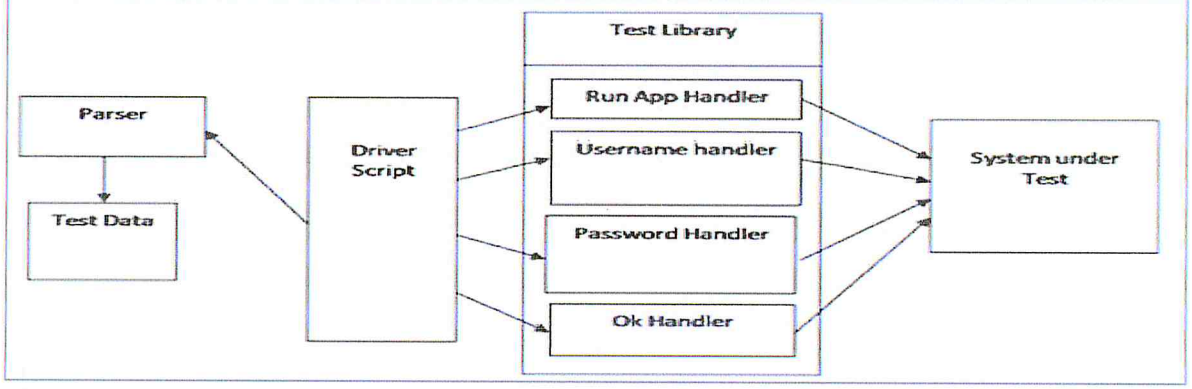

Figure II.5 : Keyword-Driven Scripting Technique [39]

Il est nécessaire de passer du temps à faire des tests pour éviter des coûts de maintenance élevés à long terme. Si le testeur passe plus de temps à développer des scripts de test, les coûts de maintenance seront plus faibles. Par contre, si le testeur utilise le moyen le plus rapide de créer des scripts de test (ex. Record / Playback Technique), le coût de maintenance sera très élevé. La figure suivante montre l'évolution des techniques de Scripting au fil du temps :

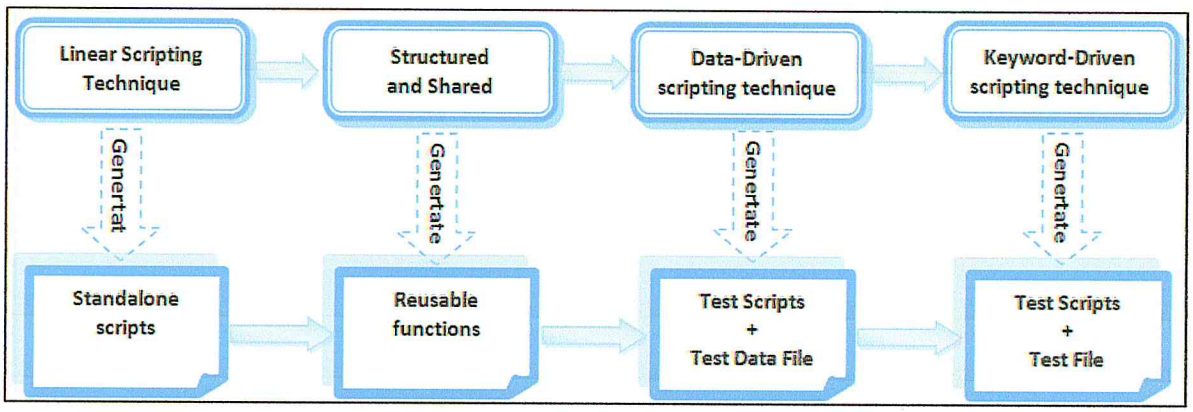

Figure II.6 : Evolution des techniques d'automatisation du test [39]

5.1. Comparatif des techniques d'automatisation du test

Le tableau suivant présente une comparaison entre les différentes techniques d'automatisation du test [39] :

Propriété	Linear	Structured	Shared	Data-Driven	Keyword-Driven
Réutilisabilité des fonctions	Non	Non	Oui	Oui	Oui
Séparation des données du script	Non	Non	Non	Oui	Oui
Séparation des actions du script	Non	Non	Non	Non	Oui
Accès au code requis	Non	Oui	Oui	Oui	Oui
Utilisation de script dans les tests de régression	Non	Oui	Oui	Oui	Oui
Niveau de compétences en programmation	Bas	Moyen	Moyen	Elevé	Elevé

Propriété	Linear	Structured	Shared	Data-Driven	Keyword-Driven
Facilité de création des scripts de test	Facile	Facile	Moyen	Difficile	Difficile
Facilité de la maintenance des scripts de test	Difficile	Difficile	Moyen	Facile	Facile

Tableau II.3 : Tableau comparatif entre les techniques d'automatisation du test [39]

6. Recherches académiques sur l'automatisation du test

Parmi les activités de test, la génération des cas de test automatique est sans doute parmi celles les plus critiques, étant donné le fort impact que celle-ci peut avoir dans l'efficacité et le rendement de tout le processus de test. Par conséquence, la génération automatique des cas de test est un enjeu important qui a fait l'objet de nombreuses recherches jusqu'aujourd'hui. C'est pourquoi un nombre important de différentes approches ont été proposées pour la génération automatique des cas de test. La figure ci-dessous présente le processus générique de génération des cas de test :

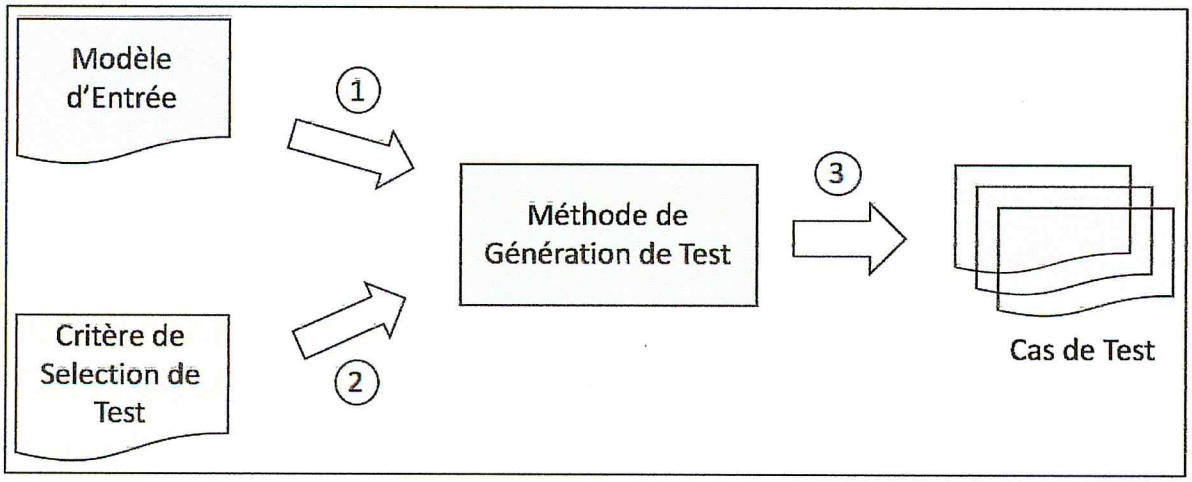

Figure II.7 : Processus générique de génération des cas de test [24]

Typiquement, il comprend trois parties essentielles :

- **Le Modèle d'entrée :** Le modèle d'entrée représente tous les éléments ou informations, issus de la conception du logiciel, utilisés comme référence pour la génération de cas de test : la structure du programme et/ou le code source, les spécifications du logiciel et/ou des modèles de conception, des informations sur les domaines des variables

d'entrées/sorties, des informations dérivées de l'exécution dynamique du programme, ou n'importe quelle combinaison de ces éléments [24].

- **Le critère de sélection de test :** En général, les cas de test sont conçus pour satisfaire une propriété donnée. Le plus souvent, on se réfère à la notion de couverture. Ainsi, le critère de sélection de test concerne toute propriété utilisée pour sélectionner des cas de test à partir d'un modèle d'entrée. Par exemple, le critère de sélection de test, en considérant comme modèle d'entrée le code source du logiciel, peut-être la couverture des instructions [24].
- **La méthode de génération de test :** Elle représente la technique utilisée pour générer les cas de test en prenant en compte le modèle d'entrée et le critère de sélection de test. Par exemple, si le modèle d'entrée est une machine d'état et le critère de sélection choisi est la couverture des transitions, une méthode de génération de test pourrait être un algorithme de génération d'un nombre minimal de chemins couvrant toutes les transitions du modèle [24].

Etant donné la grande diversité des méthodes de génération de cas de test automatique et que notre étude portera sur l'automatisation des tests fonctionnels, alors nous allons discuter les méthodes les plus utilisées.

6.1. Méthodes de génération de test

Dans cette partie, nous allons présenter les différentes méthodes de test utilisées pour la génération automatique des cas de test.

6.1.1. Méthode d'exécution symbolique

L'idée principale derrière cette méthode est de prendre en compte des valeurs symboliques, au lieu des données réelles, comme valeurs d'entrée, et de représenter les valeurs des variables de programme en tant qu'expressions symboliques. Par conséquent, les valeurs de sortie calculées par un programme sont exprimées en fonction des valeurs symboliques d'entrée.

L'état d'un programme exécuté de manière symbolique comprend les valeurs symboliques des variables de programme et une condition de chemin (PC). La condition de chemin est une formule booléenne sans quantificateur sur les entrées symboliques, il accumule les contraintes que les entrées doivent satisfaire pour qu'une exécution suive le chemin associé.

Un arbre d'exécution symbolique caractérise les chemins d'exécution suivis lors de l'exécution symbolique d'un programme. Les nœuds de l'arbre représentent les états du programme et ils sont connectés par les transitions du programme [25]. La figure ci-dessous présente un exemple de cette technique :

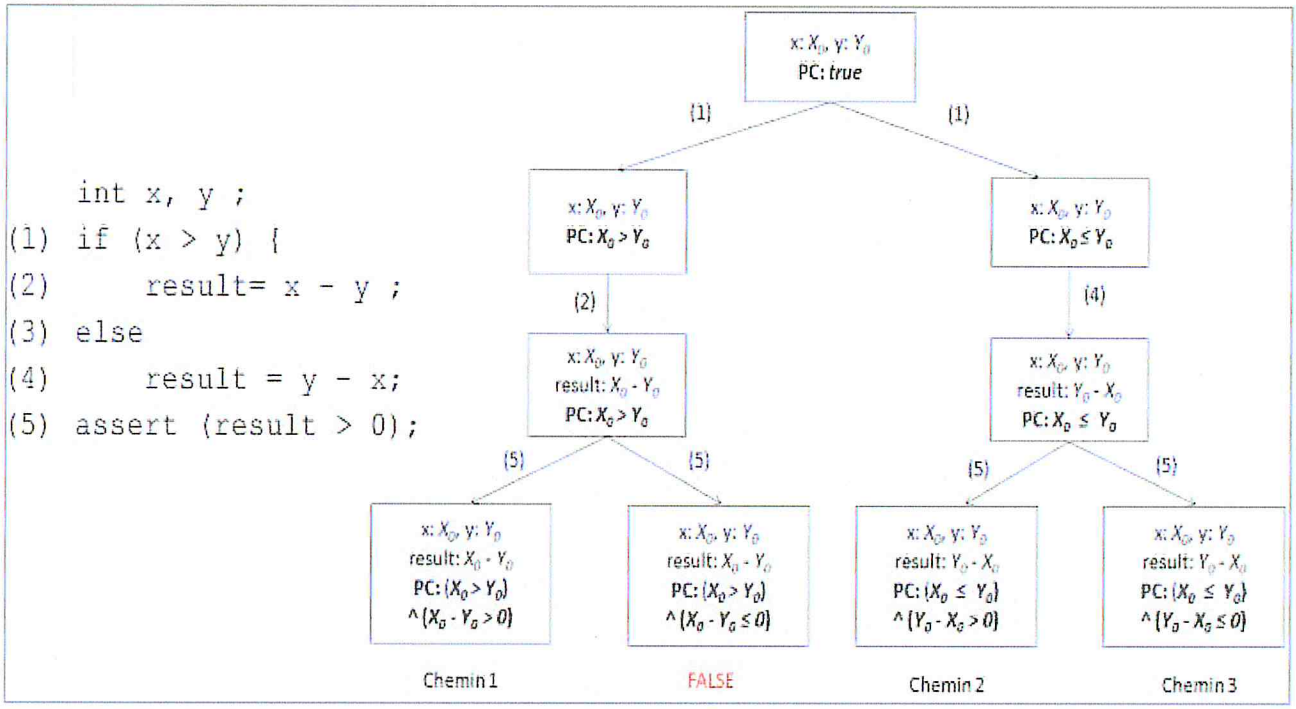

Figure II.8 : Exemple d'un programme et son arbre symbolique correspondant [24]

L'exécution symbolique commence en remplaçant les variables d'entrée x et y par des valeurs symboliques, que nous désignerons respectivement par X0 et Y0. Quant au PC il est initialisé à (true). Ensuite le programme est parcouru de façon séquentielle en exprimant à chaque point (instruction) les valeurs des variables en fonction des valeurs symboliques. Le PC est mis à jour si l'instruction est une branche. Par exemple, après l'exécution de la ligne (1) du programme de la figure II.8, le PC est mis à jour en tenant compte de l'alternative, c'est-à-dire le cas où la condition est vraie et le cas où elle est fausse [24].

L'exécution symbolique a été proposée il y a plus de quarante ans mais a récemment suscité un regain d'intérêt pour la communauté de la recherche, grâce aux progrès des procédures décisionnelles, à la disponibilité d'ordinateurs puissants et à de nouveaux développements algorithmiques [25].

6.1.2. Méthode de vérification de modèles

La vérification de modèles est une technique de vérification des propriétés sur un modèle. Les modèles utilisés pour la vérification sont généralement des machines à états finis. Typiquement, la vérification de modèles consiste à construire à partir du modèle un espace d'états (finis) lui permettant d'analyser et de déduire si oui ou non une propriété donnée est vérifiée dans cet espace [24]. La figure suivante présente un exemple de cette méthode :

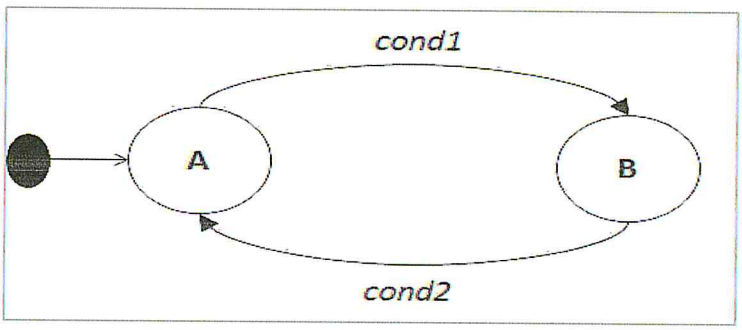

Figure II.9 : Exemple d'un modèle pour la vérification de propriétés [24]

Dans la Figure II.9, si nous considérons la transition A->B (A vers B), nous devons trouver des séquences d'entrée qui forcent le système à se retrouver dans l'état A à partir de l'état initial, et de l'état A, la condition (cond1) doit être vraie, et le prochain état doit être B. Ainsi, nous pouvons formuler une propriété à vérifier par l'outil de vérification en ces termes « Il n'existe aucun moyen (séquences d'entrée) permettant au modèle de se retrouver dans l'état B en passant par l'état A ». Le vérificateur de propriétés va essayer de trouver une trace qui viole cette propriété. Cette trace, quand elle est trouvée, représente en réalité un test qui permet de traverser la transition A->B. Le même raisonnement s'applique aux autres transitions du modèle [24].

Cette méthode a été appliquée dans la vérification du matériel et pour l'amélioration de la qualité logicielle en termes de détection des erreurs car elle est capable de vérifier des propriétés (qui représentent des tests) en explorant de façon exhaustive (en théorie) le modèle [26].

6.1.3. Méthodes basées sur les algorithmes graphiques

De manière générale, les modèles d'entrée utilisés pour la génération automatique de cas de test s'appuient sur des graphes : machines d'états, diagramme d'états-transitions, diagrammes d'activité … etc. Ainsi, dans le but de générer des tests à partir de ces modèles, des algorithmes basés sur des parcours de graphe sont souvent utilisés pour atteindre certains objectifs, par

exemple l'algorithme de Dijkstra[17] pour trouver le plus court chemin. Ces méthodes sont parfaitement adaptées aux modèles d'entrée basés sur des graphes. De plus, il existe une diversité d'algorithmes permettant de résoudre plusieurs problèmes liés au parcours des graphes [24].

6.1.4. Méthode aléatoire

La méthode aléatoire est une méthode qui consiste à générer de façon aléatoire des tests à partir des domaines d'entrée d'un système. Ainsi, elle peut être appliquée à différents types de modèles d'entrée. Par exemple, pour des modèles d'états-transitions ou de flot de données, les tests peuvent être générés en sélectionnant aléatoirement des valeurs ou des séquences de valeurs d'entrée du système sous test modélisé. Cette méthode est largement utilisée dans les tests de divers systèmes logiciels et matériels, elle vise à estimer la fiabilité d'un système et améliorer l'efficacité de la détection des erreurs [24]. La figure suivante présente un exemple de cette méthode :

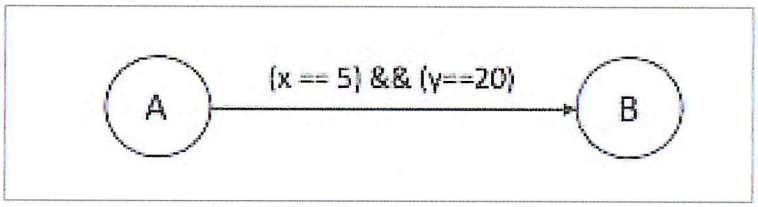

Figure II.10 : Exemple de la méthode aléatoire [24]

Dans la Figure II.10, pour que la transition A->B (A vers B) soit exécutée il faut que la valeur de x soit égale à 5 et celle de y soit égale à 20. La probabilité de tomber sur la valeur 5 parmi 1000 entiers est de 1/1000. De même, la probabilité de tomber sur la valeur 20 vaut 1/1000. Il en résulte que la probabilité de tomber simultanément sur la valeur 5 pour x et la valeur 20 pour y vaut $1/10^6$. Ainsi, cette transition a très peu de chance d'être couverte par une méthode aléatoire [24].

6.2. Comparatif des méthodes de génération de test

Le tableau suivant présente une comparaison entre les méthodes de génération de test automatique :

[17] **Dijkstra** est un algorithme qui sert à résoudre le problème du plus court chemin dans un graphe.

	Description	Avantages	Faiblesses
Méthode d'Exécution Symbolique	Exécution d'un modèle avec des valeurs génériques (symboliques)	* Exploration des chemins du modèle. * Détection des chemins : faisables et infaisables	* Résolution de certains types de contraintes * Explosion des chemins à explorer
Méthode de Vérification de Modèles	Vérification de propriétés sur un modèle	* Vérification formelle des propriétés sur un modèle	* Technique coûteuse. * Difficulté dans la définition des propriétés
Méthodes Basées sur les Algorithmes Graphiques	Techniques de parcours de graphes	* Diversité d'algorithmes * Convient aux graphes	* Complexité de certains algorithmes
Méthode Aléatoire	Sélection aléatoire de tests à partir des domaines d'entrée	* Implémentation facile et pas coûteuse	* Pas souvent efficace en termes de couverture

Tableau II.4 : Tableau comparatif des méthodes de génération de test [24]

7. Proposition de la méthode hybride de test

Après avoir mis en évidence les recherches académiques sur les différentes méthodes de génération de test, nous allons proposer une méthode hybride de test, qui combine la méthode d'exécution symbolique avec la méthode basée sur les algorithmes graphiques (plus particulièrement, le parcours des arbres). Cette méthode hybride va nous aider à automatiser le processus de test, et elle sera intégrée dans notre outil de test fonctionnel que nous allons concevoir.

7.1. Raisons du choix de la méthode hybride de test

La combinaison des deux méthodes de génération de test automatique (exécution symbolique et algorithmes graphiques) nous a donné une solution efficace et fiable, et qui convient à nos besoins. Par opposition aux autres méthodes de génération de test, comme par exemple la méthode aléatoire ou la méthode de vérification de modèles qui ne répond pas à nos besoins.

En effet, l'inconvénient majeur de la méthode aléatoire, c'est qu'elle n'est pas efficace en termes de couverture de test, hors que notre objectif est d'avoir une meilleure couverture de test.

De même, l'inconvénient majeur de la méthode de vérification de modèles réside dans la difficulté de définition des propriétés, car ils sont généralement définis de façon manuelle, ce qui peut être très fastidieux dans la mesure où une propriété est souvent définie pour chaque test, hors que notre objectif est de minimiser/éliminer les opérations manuels, et il ne s'articule pas sur la définition des propriétés.

Et donc, le défi du choix de notre méthode hybride de test été de comment représenter les scénarios de test avec une grande flexibilité dans leurs élaborations ainsi que leurs exécutions.

Le choix de la méthode d'exécution symbolique été le mieux adapté à nos besoins, car il nous a permis de modéliser les scénarios de test sous forme d'une arborescence textuel avec des valeurs symboliques (valeurs des nœuds).

Pour le deuxième choix de la méthode basée sur les algorithmes graphiques, le défi été de comment parcourir l'arbre symbolique. Et donc, dans la méthode basée sur les algorithmes graphiques, il existe de nombreux algorithmes pour le parcours d'un arbre. Chacun ayant ses propres avantages et inconvénients selon le type de structure de données utilisé, son type d'exigences en mémoire, sa complexité de temps … etc. Il est donc difficile de décider quel algorithme ou quelle structure de données sera le mieux adapté à une tâche spécifique.

Le mauvais choix de l'algorithme pour une tâche peut conduire à la production des mauvais résultats ou à des résultats qui ne sont pas optimaux. Donc, au moment de la sélection d'un algorithme ou de la structure de données pour une tâche, nous devons vérifier ses différents aspects. Un algorithme particulier peut être efficace et peut fournir le résultat optimal pour une tâche spécifique, mais il n'existe pas un algorithme qui soit le meilleur dans tous les aspects.

La principale raison du choix de basé notre algorithme sur l'algorithme DFS est que nous avons modélisé notre solution sous forme d'une arborescence et les données objectifs sont les nœuds qui sont situés dans le dernier niveau de l'arbre. Et par conséquent, l'algorithme de parcours en profondeur DFS été le mieux adapté à notre solution. Les avantages que l'algorithme DFS nous a offerts par rapport à d'autres algorithmes sont :

- ✓ Dans notre solution, nous devons parcourir tous les nœuds de l'arbre en profondeur et non en largeur.

- ✓ La profondeur de l'arbre est connue et reste inchangé pour tous les cas.
- ✓ Les nœuds objectifs (ce qu'on cherche) sont situés au dernier niveau de l'arbre qui est loin du nœud racine.
- ✓ Notre solution est destinée pour un nombre fini des nœuds.

Bien que le choix de l'algorithme DFS dans notre solution pour le parcours des chemins de l'arbre en profondeur soit efficace en termes des résultats produits et la quantité de mémoire total requise qu'il utilise pour terminer son exécution, nous avons modifié l'algorithme pour qu'il marche selon nos besoins.

7.2. Avantages et limites de la méthode hybride de test

La méthode hybride de test que nous avons proposé nous a offert beaucoup d'avantages par rapport aux autres méthodes de génération de test automatique, mais elle reste limitée dans certains aspects. Parmi les avantages et les limites de notre méthode hybride de test, nous citons :

- **Avantages :** Parmi les avantages de notre méthode : La possibilité d'explorer rapidement et facilement tous les nœuds de l'arbre à l'aide de la diversité d'algorithmes qu'on peut utiliser. Aussi, l'avantage de la facilité d'élaboration des scénarios de test ainsi que leurs manipulations.
- **Limites :** Parmi les limites de notre méthode : La difficulté de son implémentation (côté programmation) ainsi que la complexité de temps de l'algorithme utilisé, dès que les nœuds sont nombreux (plus de 60 nœuds) alors on va observer une dégradation dans la rapidité d'exécution.

8. Conclusion

L'étude complète du processus d'automatisation du test logiciel nous a permis de bien comprendre ses phases fondamentales et de savoir les différents outils d'automatisation du test ainsi que les techniques de Scripting de test automatisé, et enfin, voir les dernières recherches académiques sur les méthodes de génération de test automatique. Tout cela pour enfin proposer une méthode d'automatisation du test qui convient à nos besoins. Le chapitre suivant sera consacré à la conception de notre solution.

Chapitre III

Conception de la Solution

1. Introduction

Après avoir étudié la méthodologie et les méthodes d'automatisation de test et bien comprendre son fonctionnement, nous allons présenter dans ce chapitre la conception de notre solution. On va commencer par les spécifications des besoins. Puis, nous allons démontrer notre méthode hybride de test. Ensuite, nous allons définir l'architecture de notre système. Et enfin, nous allons présenter la modélisation de notre système à travers les diagrammes d'UML (cas d'utilisation, séquence, classe).

2. Spécification des besoins

Le test du logiciel joue un rôle important dans la détermination de son qualité. Le but est de vérifier si le logiciel répond aux exigences, les besoins et attentes spécifiques du client. Cependant, la réalisation de ce but n'est pas toujours aussi simple, car le test est souvent fait d'une manière manuelle. Lors des tests manuels, le testeur va naviguer dans le produit, il l'utilise comme un utilisateur final. D'une manière générale, les tests manuels peuvent se dérouler de deux façons différentes :

- **Test avec scénario** : Le testeur suit des parcours définis au préalable pour tester l'application.
- **Test exploratoire** : Le testeur navigue librement dans l'application pour y déceler le maximum de bugs et de problèmes.

Ces façons manuelles de test posent beaucoup d'obstacles pour les testeurs, comme la perte du temps, le problème de couverture de test, la difficulté de réutilisation des tests, les erreurs humaines …etc. De plus certains testeurs sont inexpérimenté ce qui engendre plus de travail pour leurs formation. Partant de notre étude portée sur l'automatisation des tests, nous avons identifié et déterminé les besoins des testeurs, qui sont décrits comme suit :

- Effectuer des tests dans un minimum de temps.
- Avoir une grande couverture de test.
- Avoir une fiabilité et une bonne précision lors de l'exécution des tests.
- La possibilité de réutilisation et de répétition des tests.
- La facilité d'écrire des scénarios de test bien organisés.

2.1. Proposition de la solution

Afin de faire face aux obstacles subis par les testeurs et de satisfaire leurs besoins, nous allons présenter notre solution qui va s'articuler sur :

- Le développement d'un outil d'automatisation de test fonctionnel qui va gérer et exécuter les tests d'une manière automatique avec une intervention minimale des testeurs, ce qui facilite grandement leurs tâches. Cet outil va minimiser le temps nécessaire à l'exécution des tests, enregistrer les tests pour une réutilisation future, augmenter la couverture des tests, élaborer des rapports détaillés sur les tests après chaque exécution. Cet outil ne nécessite pas une grande expérience dans le domaine de test pour les utilisateurs ou des compétences dans la programmation, ce qui facilite et simplifie le travail pour les testeurs inexpérimenté.

Dans cet outil, nous allons intégrer notre méthode hybride de test. Cette méthode joue un rôle important dans la manière d'élaborer les tests, et elle représente une grande partie du travail de cet outil, surtout dans l'exécution des tests.

3. Démonstration de la méthode hybride de test

Notre méthode hybride de test s'articule sur la combinaison de deux méthodes utilisées dans le test logiciel (plus particulièrement dans la génération de test automatique). Le concept et l'idée derrière ces méthodes nous ont inspirées de les combiner pour réaliser notre objectif. Le fonctionnement de notre méthode hybride de test est décrit dans les points suivants :

- **L'idée de la méthode d'exécution symbolique :** Dans notre méthode hybride de test, nous avons représenté les scénarios de test, les cas de tests, les composants… etc., comme valeurs symboliques. Et comme la méthode d'exécution symbolique est sous forme d'une arborescence et le parcours des chemins entre les nœuds est représenté par une formule boolienne (dans le cas général), et indique les conditions des chemins qui doivent être résolues. Alors, nous avons modélisé notre solution avec une arborescence, prenant le même principe de la méthode d'exécution symbolique à l'exception que le parcours des chemins n'est pas toujours une condition boolienne et les nœuds du dernier niveau de l'arbre représentent les données nécessaires pour exécuter le test.
- **L'idée de la méthode basée sur les algorithmes graphiques :** Puisque notre méthode hybride consiste à parcourir un arbre, la nécessité d'un algorithme de parcours de graphe été primordiale, et donc nous avons basé l'algorithme utilisé dans notre méthode sur un

algorithme de parcours en profondeur qui est le mieux adapté à notre situation. L'algorithme est appelé Depth First Search, il permet de parcourir l'arbre en profondeur jusqu'au dernier niveau où les données objectifs sont situés.

Dans les sections prochaines, nous allons présenter le graphe de parcours des scénarios de test ainsi que l'algorithme Depth First Search.

3.1. Graphe de parcours des scénarios de test

Le graphe de parcours des scénarios de test représente une vue globale de la manière de l'organisation des scénarios de test, il est sous forme d'un arbre pondéré, le nœud juste après la racine représente un scénario de test. Les fils du scénario représentent les cas de test, et les composants sont les fils des cas de test. La figure suivante illustre ce graphe :

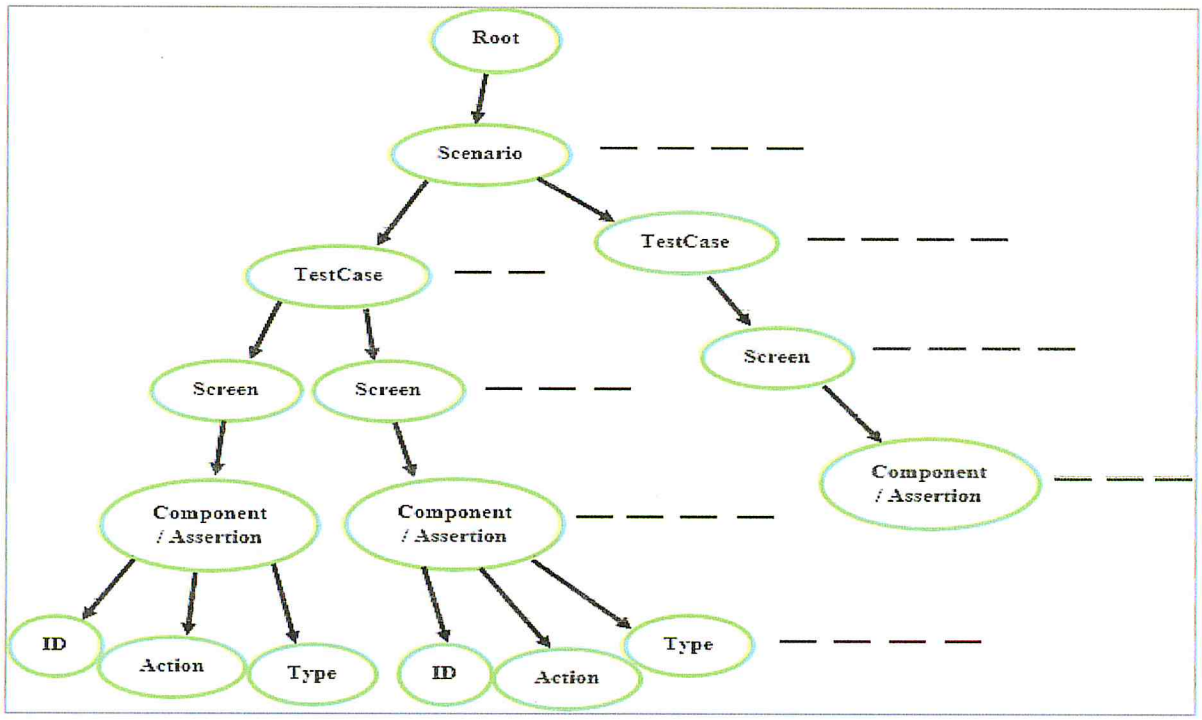

Figure III.1 : Graphe de parcours des scénarios de test

Vu que les actions et les informations nécessaires pour l'exécution des scénarios de test sont situées au dernier niveau de l'arbre, l'utilisation de l'algorithme DFS pour le parcours des chemins a prouvé son efficacité.

3.2. Présentation de l'algorithme Depth First Search

L'algorithme DFS est un algorithme récursif utilisé pour la recherche dans un graphe, plus précisément un arbre, il est basé sur l'idée de retour en arrière. Il implique des recherches exhaustives de tous les nœuds en allant vers l'avant, si possible, sinon en revenant en arrière. L'algorithme DFS traverse un arbre ou un graphe depuis le sommet parent jusqu'à ses enfants et petits-enfants dans un seul chemin jusqu'à ce qu'il atteigne une impasse. Lorsqu'il n'y a plus de vertex à visiter dans un chemin, l'algorithme DFS va revenir à un point où il peut choisir un autre chemin à prendre. Il répétera le processus encore et encore jusqu'à ce que tous les sommets aient été visités [30]. La figure ci-dessous illustre l'algorithme DFS :

Algorithm : Depth First Search

```
1 : Function DFS (initial, ID, Action, Type)
2 :   maxDepth = root.depth
3 :   initial.level = 0
4 :   open = new Pile
5 :   visited = new Tableau
6 :   insert (open, initial)
7 :   Print (" Test Started ")
8 :   While (open is not Empty) do {
9 :     n = pop (open)
10 :    insert (visited, n)
11 :    Foreach Edge e at n do {
12 :      next = child from Edge e
13 :      If (visited doesn't contain next) Then {
14 :        next.level = n.level − 1
15 :        If (next.getName = ID) Then {
16 :          ID = get the data stored in the node next }
17 :        Else if (next.getName = Action) Then
18 :          { Action = get the data stored in the node next }
19 :        Else
20 :          { Type = get the data stored in the node next }
21 :        Print (ID, Action, Type)
22 :        FindElementBy(ID).Execute(Action)
23 :        If (next.depth < maxDepth) Then {
24 :          insert (open, next) }
25 :    } // If } // Foreach } // While
26 :  return " Test Finished "
```

Figure III.2 : Pseudocode de l'algorithme Depth First Search

Le fonctionnement de l'algorithme DFS est présenté dans la figure suivante :

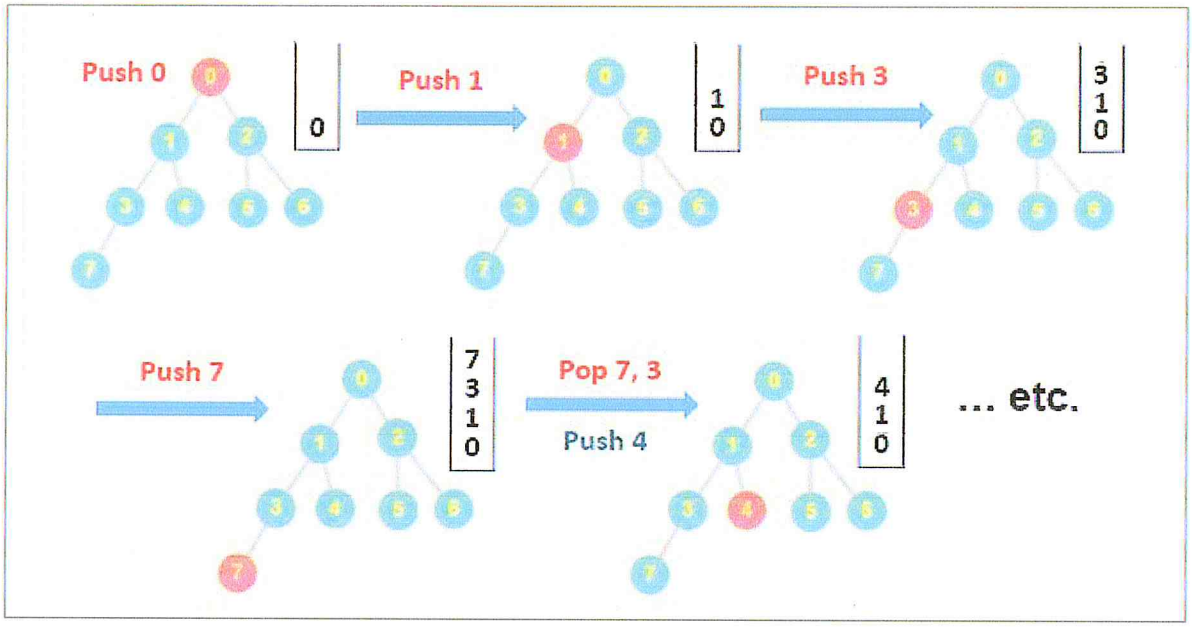

Figure III.3 : Le fonctionnement de l'algorithme DFS

Le parcours en profondeur des scénarios de test suit la stratégie préfixe qui traverse les nœuds comme suit :

- Visiter le nœud racine
- Visitez tous les nœuds dans le sous arbre gauche
- Visitez tous les nœuds dans le sous arbre droite

Le détail du déroulement de l'algorithme DFS est décrit dans les étapes suivantes :

1) Empiler le nœud racine (0) et mémoriser les chemins sortant de ce nœud.
2) Empiler le nœud (1) et mémoriser les chemins sortant de ce nœud.
3) Faire la même procédure pour les autres nœuds en profondeur (3,7).
4) Lorsque le nœud n'admet pas de chemins, alors désempiler le nœud de la pile et faire un retour en arrière pour parcourir les chemins mémoriser (non exploré), dans notre cas il va retourner vers le nœud (1) et empiler le nœud (4).
5) Faire les mêmes procédures pour le sous arbre droite.

Le même déroulement est appliqué sur le graphe de parcours des scénarios de test dans la Figure III.1, qui représente notre solution.

3.2.1. Complexité de temps et d'espace de l'algorithme DFS

La complexité de temps est la durée totale requise par un algorithme pour terminer son exécution. Généralement, le temps d'exécution d'un algorithme dépend de ce qui suit [36] :

- L'exécution est sur une machine monoprocesseur ou multiprocesseur.
- Un processeur 32 bits ou 64 bits
- La vitesse de lecture et écriture de la machine.
- Le temps nécessaire pour effectuer des opérations arithmétiques, des opérations logiques, des opérations de retour et d'affectation, etc.
- Les données d'entrée.

Lorsque nous calculons la complexité de temps d'un algorithme, nous considérons seulement les données d'entrée et nous ignorons les autres paramètres.

La complexité d'espace est la quantité totale de mémoire requise par un algorithme pour terminer son exécution. Généralement, lorsqu'un programme est en cours d'exécution, il utilise la mémoire de l'ordinateur pour les raisons suivant [36] :

- **Espace d'instruction :** C'est la quantité de mémoire utilisée pour stocker la version compilée des instructions.
- **Pile environnemental :** C'est la quantité de mémoire utilisée pour stocker des informations de fonctions partiellement exécutées au moment de l'appel de fonction.
- **Espace de données :** C'est la quantité de mémoire utilisée pour stocker toutes les variables et constantes.

La même chose avec la complexité de temps, lorsque nous voulons effectuer l'analyse d'un algorithme basé sur sa complexité d'espace, nous considérons uniquement l'espace de données et nous ignorons l'espace d'instruction ainsi que la pile environnementale.

Dans le pire des cas, l'algorithme DFS devra rechercher tous les nœuds dans l'arbre, plus un niveau supplémentaire de (n) nœuds pour prendre en compte la condition d'arrêt. Dans ce cas, la complexité de temps est égale à ($1 + n + n * n + \ldots = n^d = O(n^d)$), où (n) représente le facteur de branchement[18], et (d) représente la profondeur maximale de l'arbre [37].

En ce qui concerne la complexité d'espace, au pire des cas, l'espace de stockage maximal que l'algorithme pourrait utiliser à chaque itération doit être considéré. Cet espace est dominé par la taille de la pile (les nœuds stocké qui se trouvent dans le chemin de parcours). Intuitivement, ceci représente juste une ligne droite descendant vers les feuilles. Si on suppose que le but se trouve à un nœud feuille de l'arbre. Dans ce cas, à chaque itération de la recherche,

[18] **Facteur de branchement** est le nombre de fils de chaque nœud.

l'algorithme DFS ajoutera (n) nœuds dans la pile, qui se trouvent dans le chemin du parcours. Au plus, il peut le faire (d) fois, jusqu'à ce qu'il atteigne le bas (nœud-feuille) de l'arbre. À ce stade, la pile est de taille **(n*d)**. D'où la complexité d'espace du scénario le plus défavorable est égale à **O(n*d)**, où (n) représente le chemin le plus long possible et (d) le maximum des chemins alternatifs par nœud [37].

Le facteur de branchement des nœuds est calculé selon la formule suivante :

$$B = \frac{\textit{Nombre des noeuds d'un niveau}}{\textit{Nombre des noeuds du niveau précédent}}$$

Pour illustrer la complexité de temps de notre algorithme qui est basé sur l'algorithme DFS, nous allons utiliser le graph de Big-O Notation, qui est sert à décrire la performance ou la complexité d'un algorithme. Il décrit spécifiquement le scénario le plus défavorable, il peut être aussi utilisé pour décrire le temps d'exécution ou l'espace de stockage maximal requis par un algorithme [38]. La figure suivante présente le graph de Big-O Notation :

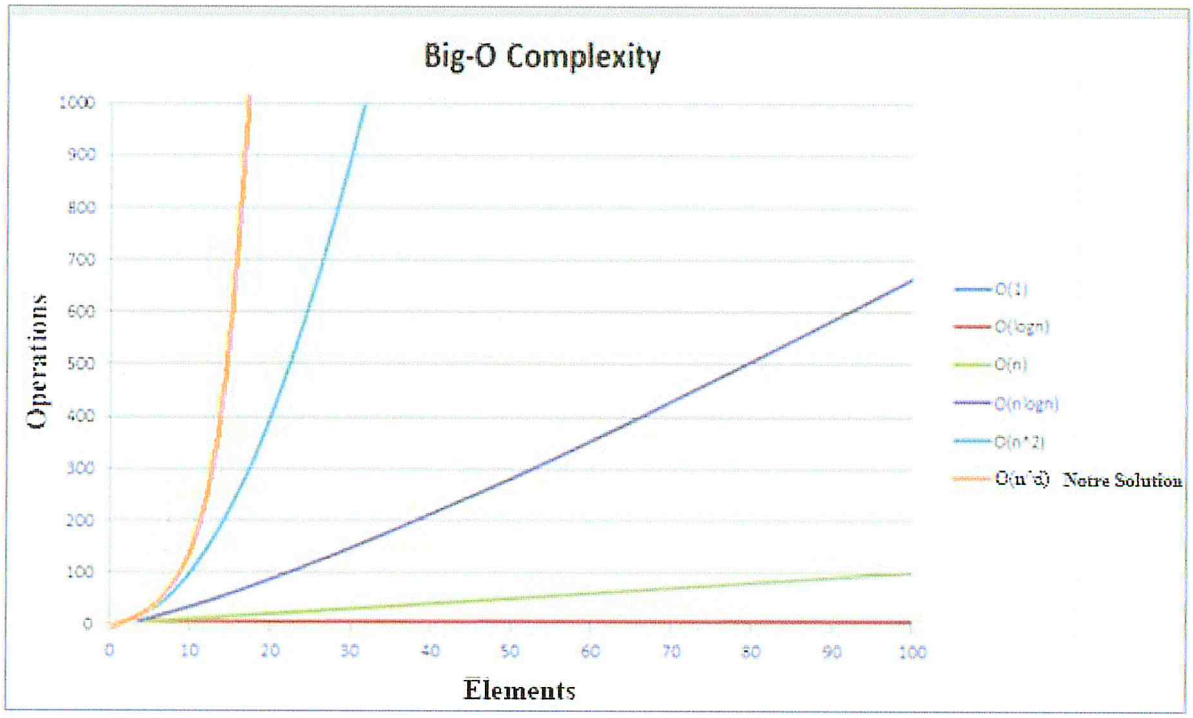

Figure III.4 : Graphe de complexité de temps (Big-O) d'un algorithme

Le graph dans la figure III.3, montre que l'algorithme que nous avons utilisé, a un temps d'exécution polynomial. La complexité de temps de notre algorithme est donc égale à $O(n^d)$, où (n) représente le facteur de branchement moyen de l'arbre et (d) représente la profondeur

maximale de l'arbre (fixé à 5 niveaux). Donc on peut conclure que la performance de notre algorithme est acceptable, et sa complexité peut accroitre suivant l'augmentation du nombre de nœuds de l'arbre.

4. Architecture du système

L'architecture désigne la structure générale inhérente à un système informatique, l'organisation des différents éléments du système et des relations entre ces éléments. Cette structure fait suite à un ensemble de décisions stratégiques prises durant la conception du système informatique. La figure suivante illustre l'architecture de notre système :

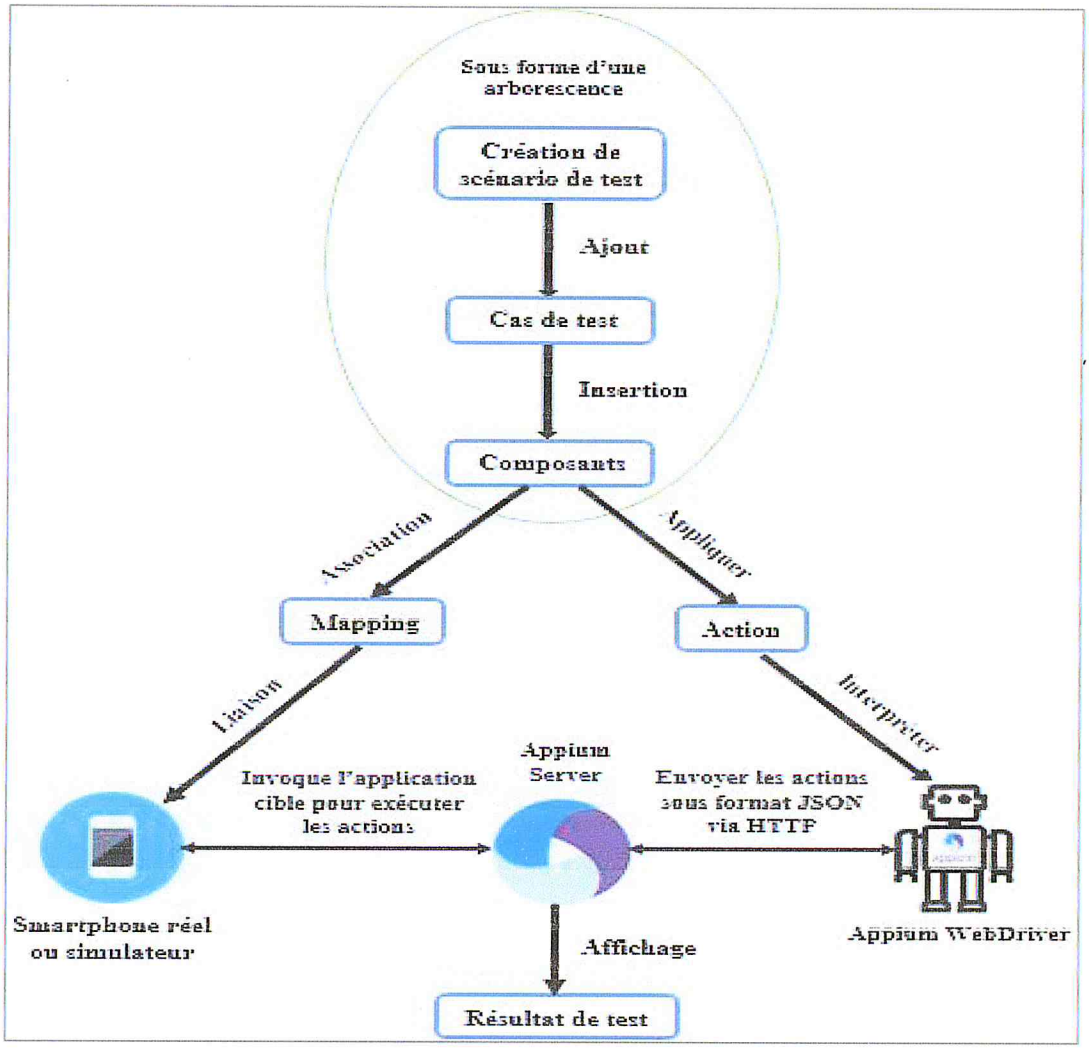

Figure III.5 : Schéma de l'architecture du système

L'architecture de notre système est basée sur trois phases :

1) **La phase de création des scénarios de test :** Dans cette phase, le testeur va créer des scénarios de test pour ensuite ajouter les cas de test qu'il va exécuter, ces scénarios sont créés sous forme d'un graphe (arbre pondéré). Chaque scénario admet des cas de test, et ces cas de test admettent des composants conceptuels de l'application cible.
2) **La phase de Mapping et actions :** La première partie de cette phase est appelé le Mapping qui désigne la transformation des composants conceptuels en composants réels, ceci est fait à travers une association d'un identifiant de chaque composant créé avec le composant réel de l'application cible. L'identifiant est récupéré depuis un outil d'inspection des éléments (Appium Inspector) sur l'application qui va être testé. La deuxième partie de cette phase est l'application des actions. Après la phase du Mapping, le testeur doit choisir une action pour chaque composant crée, ces actions vont être appliquées sur les composants réels de l'application cible.
3) **La phase de l'exécution des tests :** Dans cette phase les actions vont être envoyées vers une interface de contrôle à distance qui est Appium WebDriver[19], cette interface va interpréter les actions qui vont manipuler l'application cible pour ensuite les envoyer vers le serveur Appium via le protocol HTTP. Le serveur Appium invoque à son tour l'application pour exécuter les actions.

Le détail de fonctionnement des phases de l'architecture du système ainsi que le fonctionnement interne du système vont être présenté avec les diagrammes d'UML (activité, cas d'utilisation, séquence, classe).

5. Présentation du langage de modélisation UML

UML (Unified Modeling Language) est une notation permettant de modéliser un problème de façon standard. Ce langage est né de la fusion de plusieurs méthodes existant auparavant tel qu'OMT[20] (Object Modeling Technique) et OOSE[21] (Object Oriented Software Engineering), et il est devenu désormais la référence en termes de modélisation objet [27].

---

[19] **Appium WebDriver** est une interface de contrôle à distance qui permet le contrôle des agents utilisateurs (navigateurs). Il fournit un ensemble d'API permettant de découvrir et de manipuler des éléments DOM dans des documents Web et de contrôler le comportement d'un agent utilisateur.
[20] **OMT** est une technique de modélisation destinée à la conception et la modélisation pour le POO
[21] **OOSE** est langage de modélisation objet créé par Ivar Jacobson

5.1. Diagramme d'activité de l'architecture du système

Ce diagramme représente le flux de travail de l'architecture du système à travers les différentes actions qu'on peut appliquer et en mettant l'accent sur la séquence et les conditions du flux. La figure suivante illustre ce diagramme :

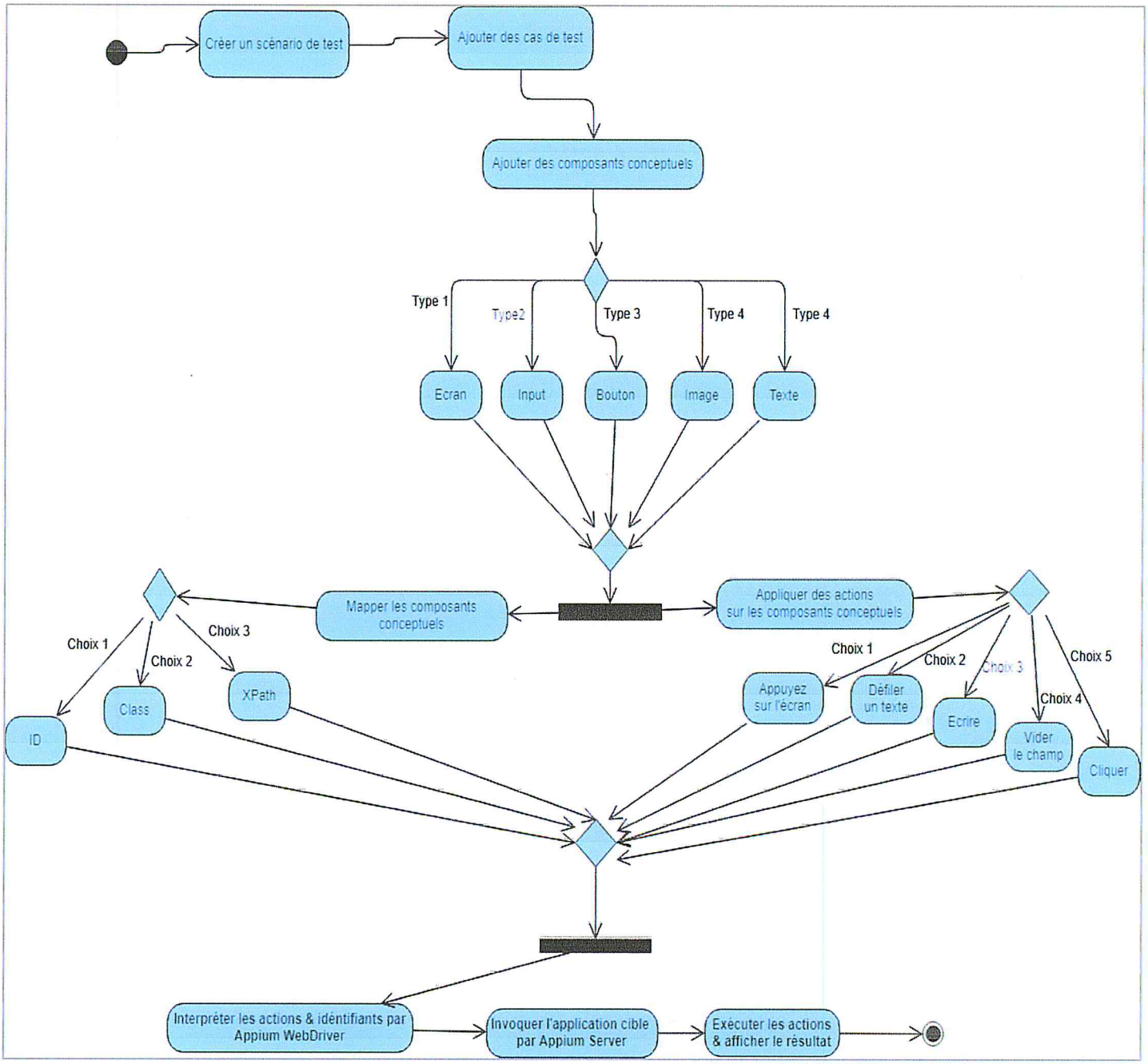

Figure III.6 : Diagramme d'activité de l'architecture du système

Ce diagramme d'activité nous a donné un aperçu global sur le mécanisme de travail de notre système ainsi que de faciliter sa compréhension.

5.2. Diagramme de cas d'utilisation

Les cas d'utilisation ont été définis initialement par Ivar Jacobson[22] en 1992 dans sa méthode OOSE (Object Oriented Software Engineering). Les cas d'utilisation constituent un moyen de recueillir et de décrire les besoins des acteurs du système. Ils peuvent être aussi utilisés ensuite comme moyen d'organisation du développement du logiciel, notamment pour la structuration et le déroulement des tests du logiciel [28].

5.2.1. Diagramme de cas d'utilisation globale

Ce diagramme représente les principales fonctionnalités de notre système, il schématise d'une façon globale les actions que le testeur peut exercer sur le système. La figure suivante décrit ce diagramme :

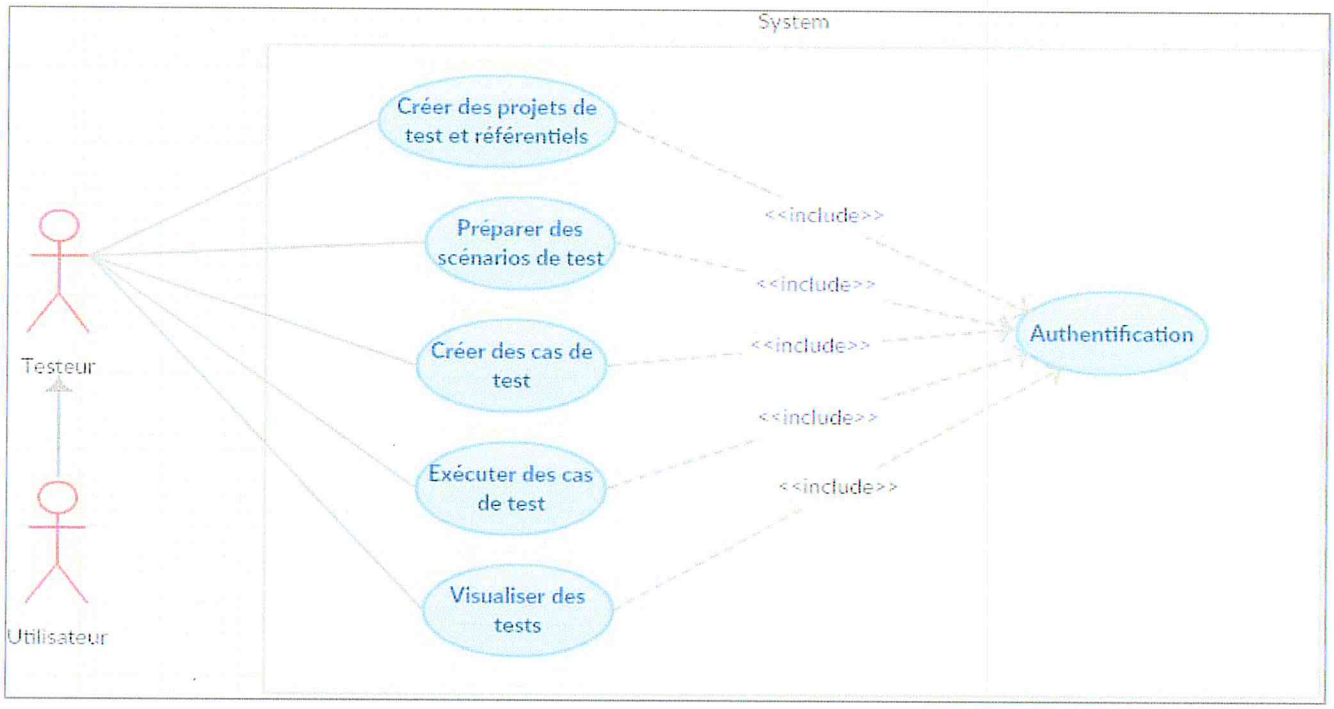

Figure III.7 : Diagramme de cas d'utilisation globale du système

5.2.2. Diagramme de cas d'utilisation de préparation des scénarios de test

Ce diagramme représente une fonctionnalité de base de notre système. L'action de préparation des scénarios de test permet au testeur de manipuler les scénarios de test (ajout / modification / suppression) et les enregistrer pour les exécuter autant de fois qu'il veut. Pour

[22] **Ivar Jacobson** est un informaticien. Il est principalement connu pour être l'un des concepteurs d'UML

pouvoir enregistre un scénario de test, le testeur doit créer le composant écran qui sert à décrire les interfaces de l'application cible. Ensuite il va créer les composants conceptuels de l'application cible. Et enfin il doit Mapper les composants conceptuels (Input, Bouton, Texte, Image…etc.) créés avec leurs ID réels dans l'application cible et choisir les actions qu'il veut appliquer sur ces composants. La figure suivante décrit ce diagramme :

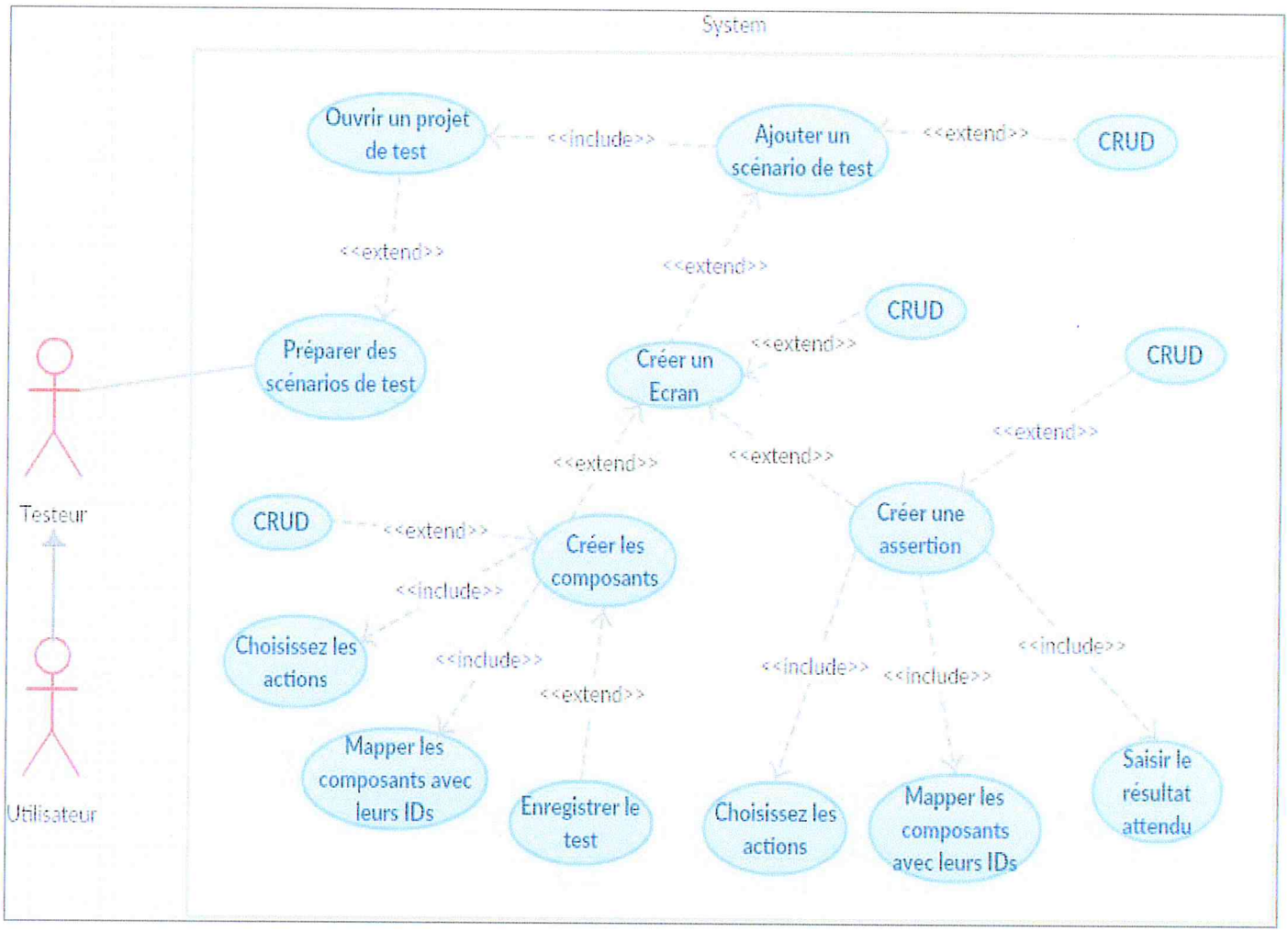

Figure III.8 : Diagramme de cas d'utilisation de préparation des scénarios de test

5.2.3. Diagramme de cas d'utilisation d'exécution des cas de test

Ce diagramme représente la fonctionnalité de l'exécution des tests. Il illustre les opérations que le testeur peut faire pour exécuter les différents scénarios de test. Le testeur peut manipuler l'ordre de l'exécution qui représente la priorité de test, comme il peut ajouter ou supprimer des scénarios / cas de test qu'il ne veut pas exécuter. Le testeur peut aussi choisir l'application qui va être sujet de test. La figure ci-dessous décrit ce diagramme :

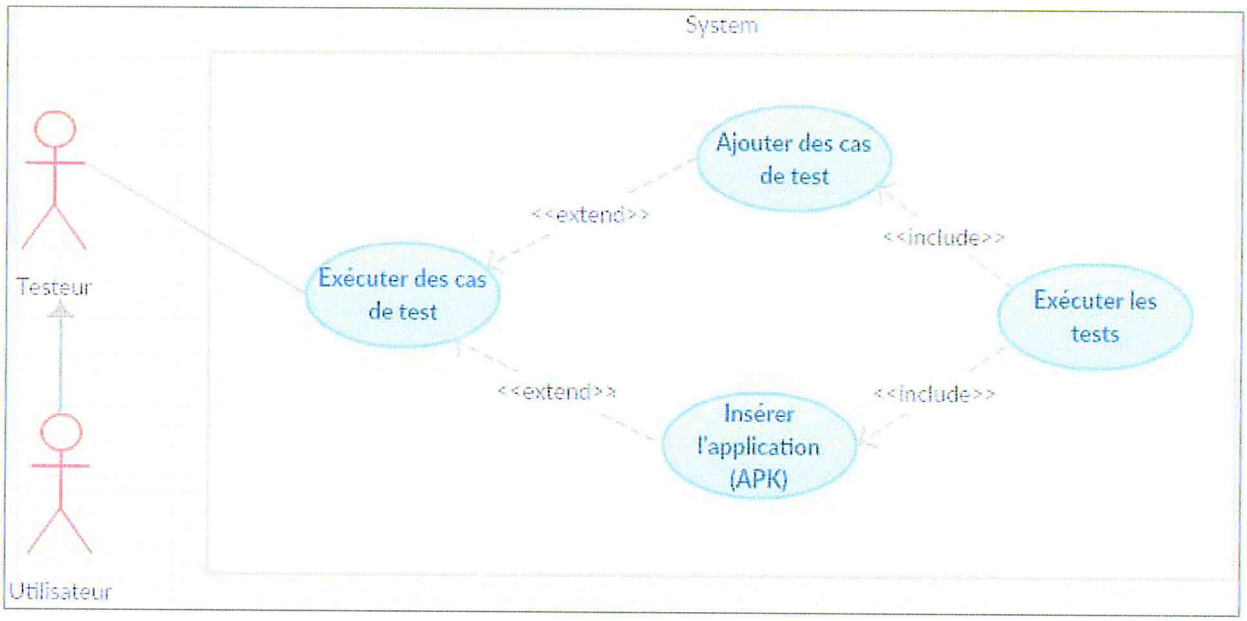

Figure III.9 : Diagramme de cas d'utilisation d'exécution des cas de test

5.3. Diagramme de séquence

L'objectif du diagramme de séquence est de représenter les interactions entre des objets en indiquant la chronologie des échanges. Cette représentation peut se réaliser par des cas d'utilisation en considérant les différents scénarios associés [28].

5.3.1. Diagramme de séquence de l'authentification

Ce diagramme représente le scénario d'authentification pour l'utilisateur qui veut accéder au système. La figure ci-dessous illustre ce diagramme :

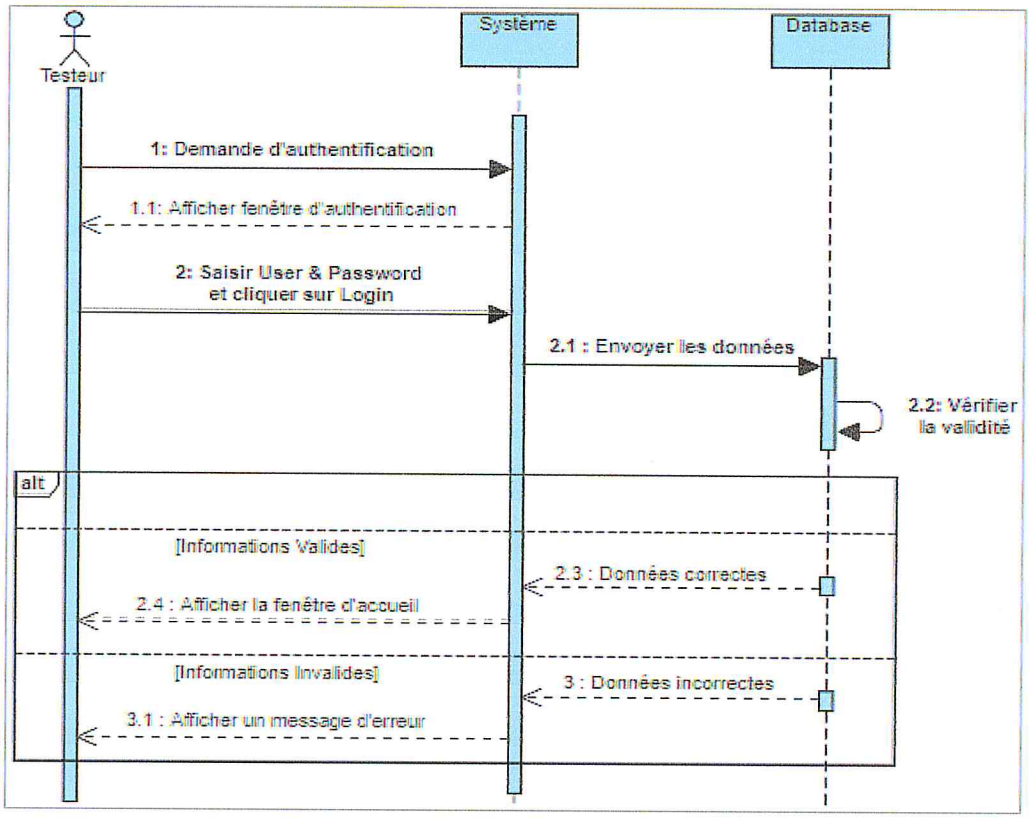

Figure III.10 : Diagramme de séquence de l'authentification d'un

5.3.2. Diagramme de séquence de la création des projets de test

Ce diagramme représente le scénario de création des projets de test. Le testeur peut créer plusieurs projets dont il associé des référentiels de test. La figure ci-dessous illustre ce diagramme :

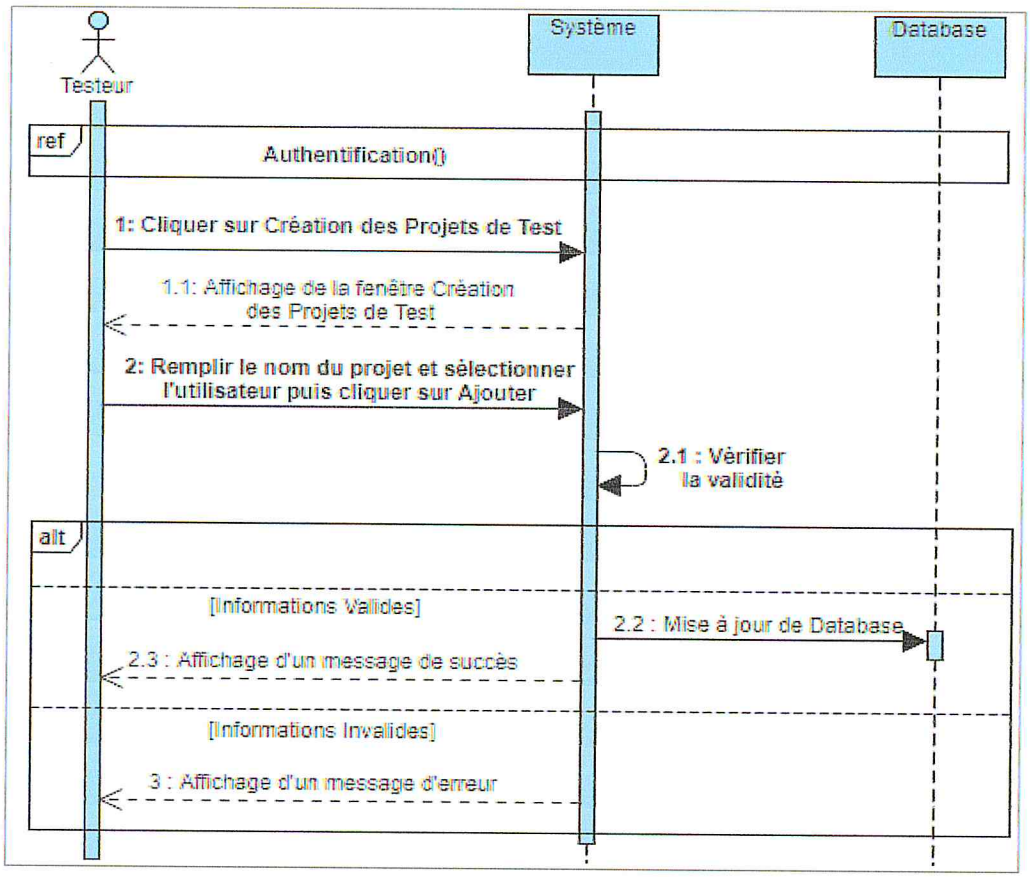

Figure III.11 : Diagramme de séquence de la création des projets de test

5.3.3. Diagramme de séquence de la création des cas de test

Ce diagramme représente le scénario de création des cas de test. Le testeur peut créer plusieurs cas de test pour chaque scénario. Les cas de test représentent les différents conditions ou perspectives qu'on peut appliquer sur chaque fonctionnalité. La figure ci-dessous illustre ce diagramme :

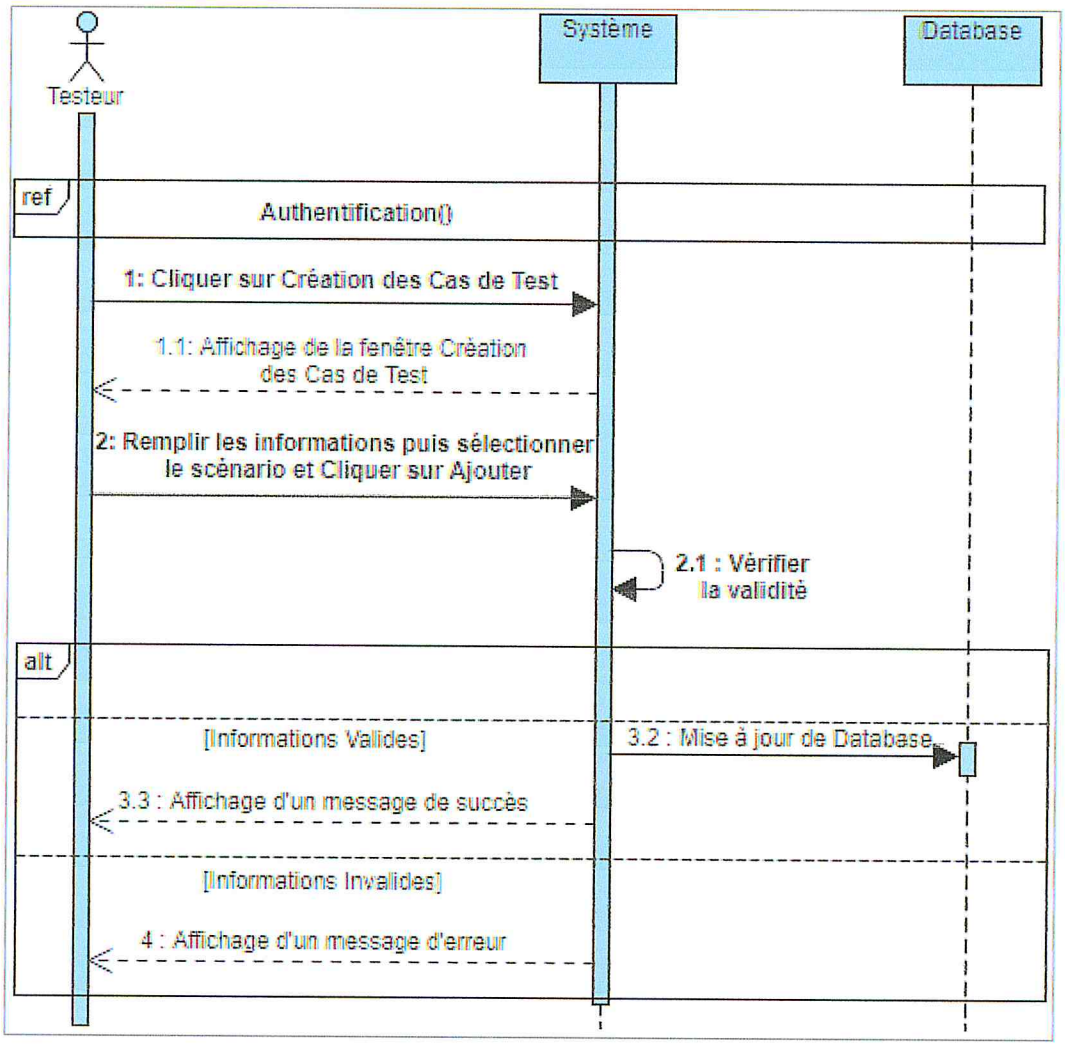

Figure III.12 : Diagramme de séquence de la création des cas de test

5.3.4. Diagramme de séquence de l'ajout des scénarios de test

Ce diagramme représente le scénario d'ajout des cas de test. Le testeur peut ajouter plusieurs scénarios de test. Chaque scénario admet un ou plusieurs cas de test. Le testeur doit insérer les informations qui représentent les composants conceptuels (Ecran, Input, Boutons …etc.) de l'application cible dans les cas de test. Enfin il doit enregistrer les scénarios de test pour pouvoir les exécuter ultérieurement. La figure ci-dessous illustre ce diagramme :

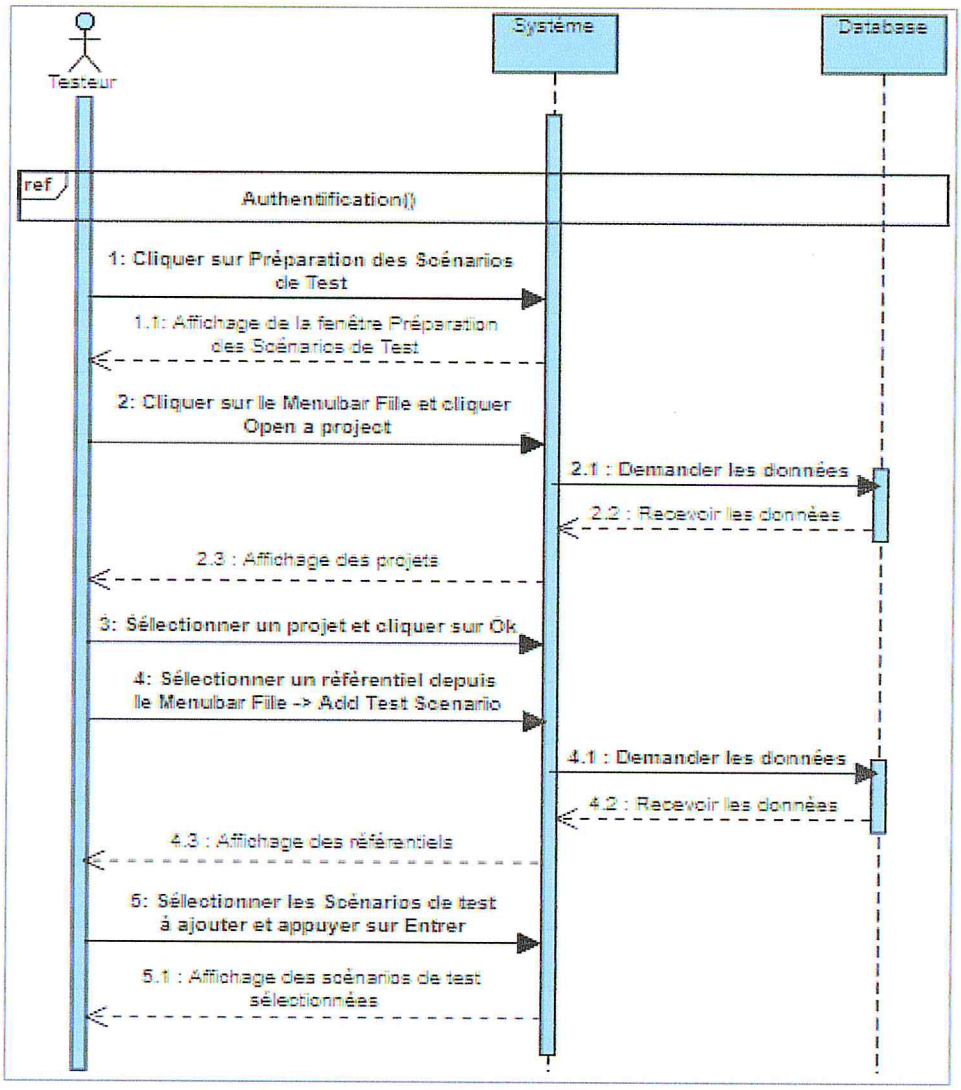

Figure III.13 : Diagramme de séquence de l'ajout des scénarios de test

5.3.5. Diagramme de séquence de la création des composants

Ce diagramme représente le scénario de création des composants. Puisque chaque cas de test contient plusieurs composants conceptuels qui représentent les interfaces / éléments de l'application qui va être testé. Le testeur doit mapper les composants avec leurs ID réel dans l'application cible. Puis, il doit choisir les actions qu'il veut appliquer sur ces composants. Les actions servent à manipuler et naviguer dans les différentes interfaces / éléments de l'application cible. La figure ci-dessous illustre ce diagramme :

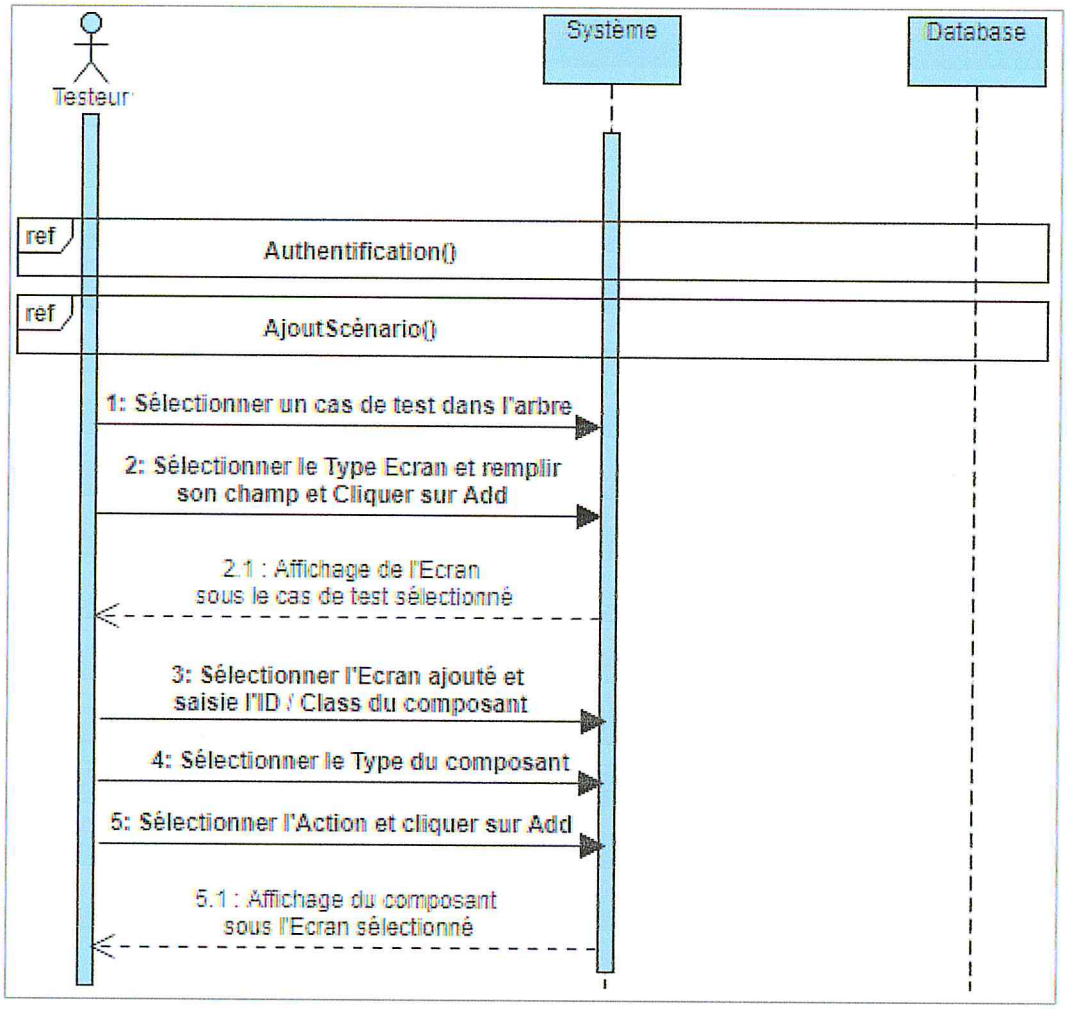

Figure III.14 : Diagramme de séquence de la création des composants

5.3.6. Diagramme de séquence de la modification des composants

Ce diagramme représente le scénario de modification des composants. Le testeur peut modifier les informations des composants (ID, Class, Action … etc.) ou les assertions en cas où il a commis des erreurs. La figure ci-dessous illustre ce diagramme :

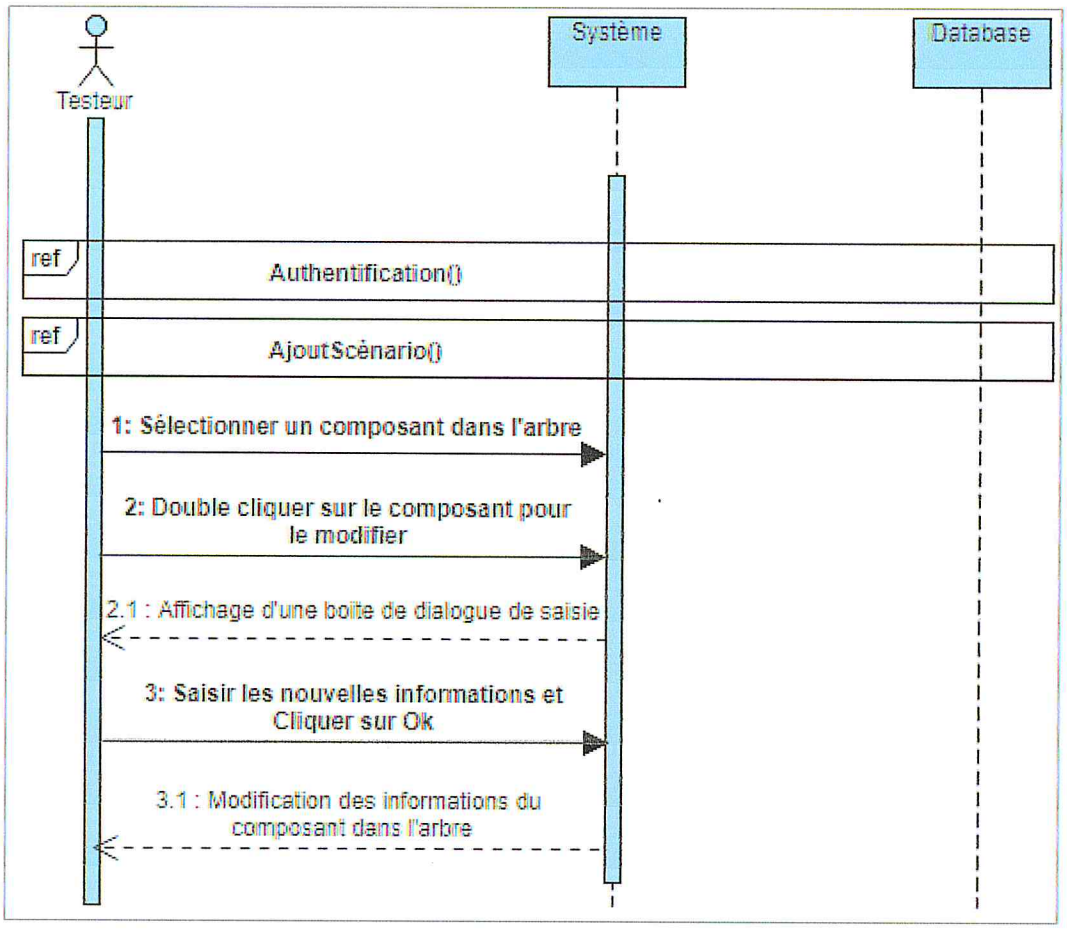

Figure III.15 : Diagramme de séquence de la modification des composants

5.3.7. Diagramme de séquence de la suppression des composants

Ce diagramme représente le scénario de suppression des composants. Le testeur peut supprimer les composants (Ecran, Composant) ou les assertions. Il peut aussi supprimer des cas de test ou des scénarios de test complètement. La figure ci-dessous illustre ce diagramme :

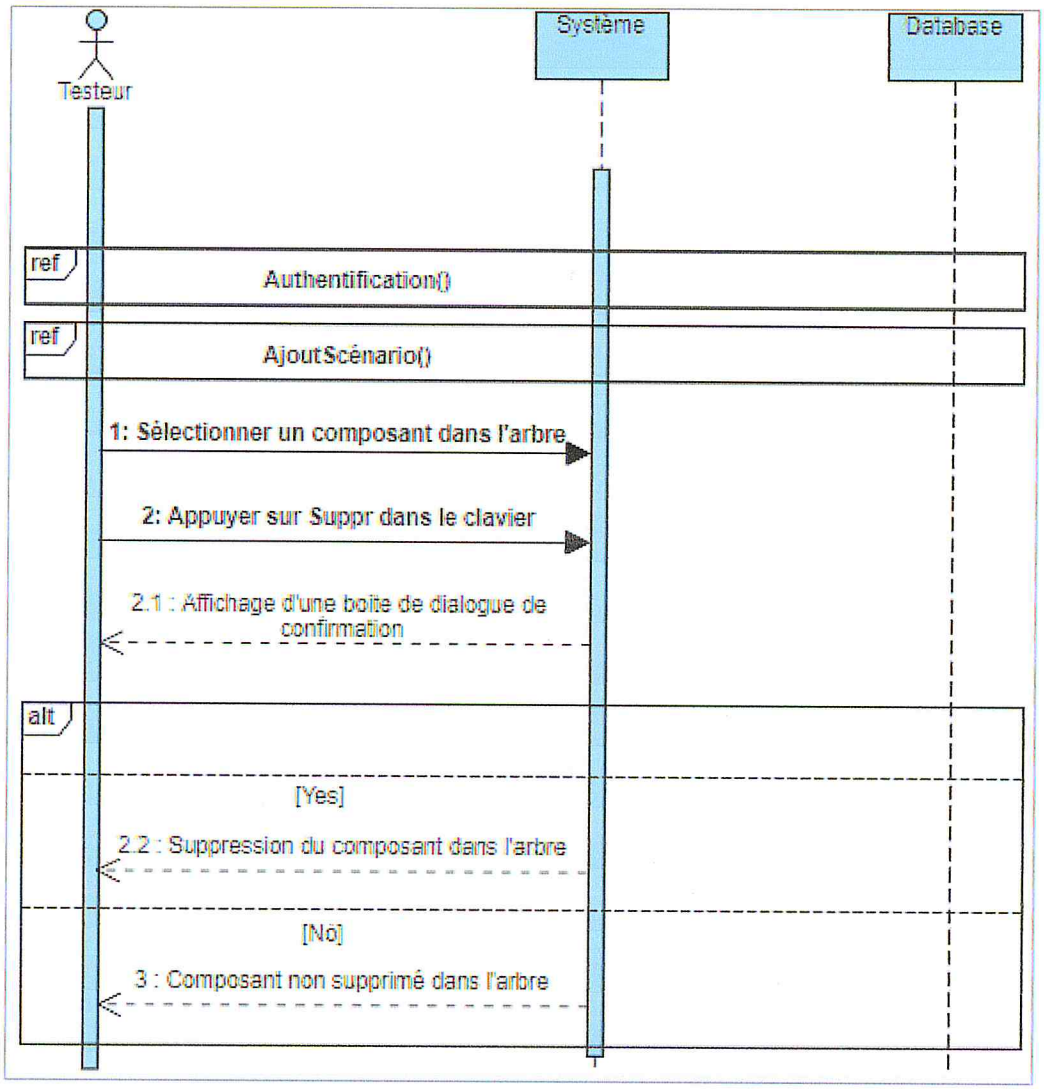

Figure III.16 : Diagramme de séquence de la suppression des composants

5.3.8. Diagramme de séquence de l'exécution des cas de test

Ce diagramme représente le scénario d'exécution des cas de test. Le testeur peut exécuter tous les cas de test d'un référentiel ou bien choisir des cas de test parmi qui existes. Il peut aussi choisir sur quelle application il va exécuter les cas de test. La figure ci-dessous illustre ce diagramme :

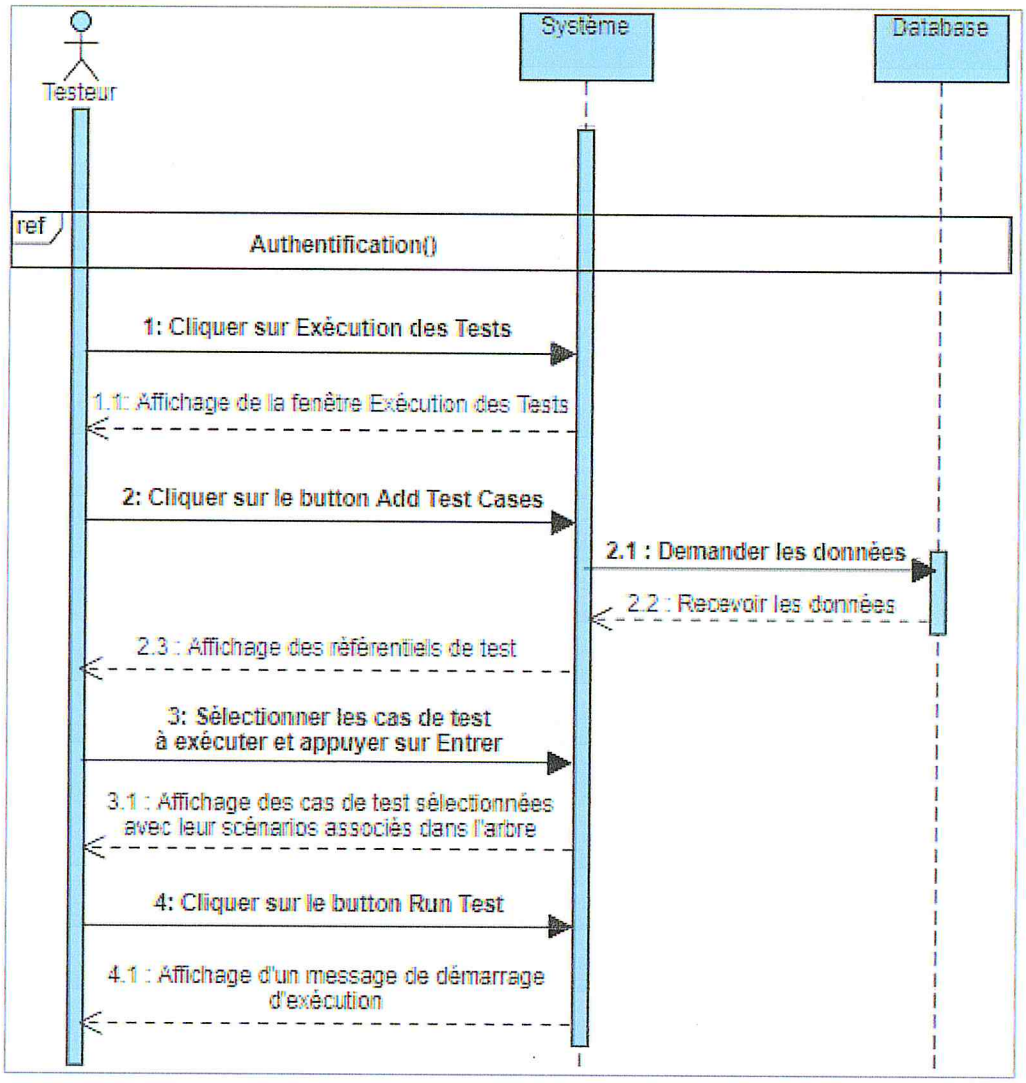

Figure III.17 : Diagramme de séquence de l'exécution des cas de test

5.4. Diagramme de classe

Le diagramme de classe constitue l'un des pivots essentiels de la modélisation avec UML. Ce diagramme permet de donner la représentation statique du système à développer. Cette représentation est centrée sur les concepts de classe et d'association. Chaque classe se décrit par les données et les traitements dont elle est responsable pour elle-même et vis-à-vis des autres classes. Les traitements sont matérialisés par des opérations [28].

5.4.1. Schéma du diagramme de classe

La figure ci-dessous illustre le diagramme de classe qui est une représentation statique de notre système, il décrit les données et les traitements entre les différentes classes de notre système :

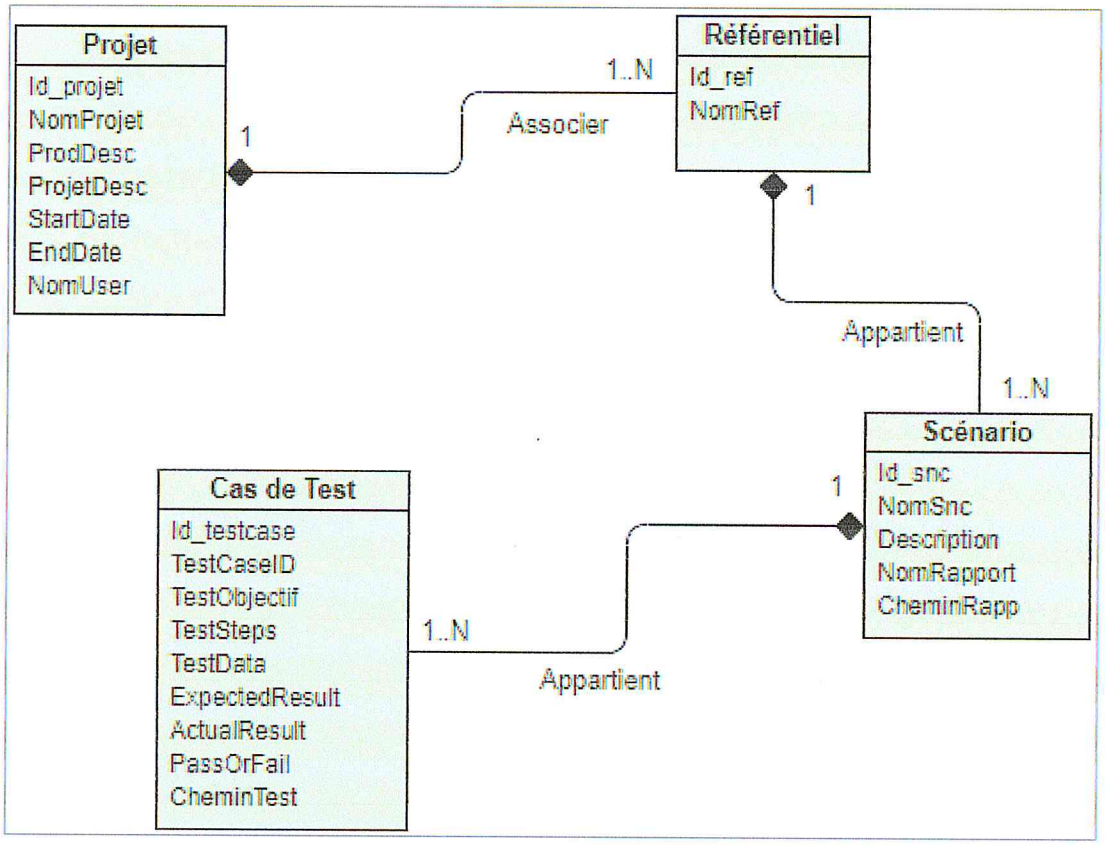

Figure III.18 : Schéma du diagramme de classe du système

5.4.2. Règles de gestion

Le diagramme de classe de notre système est basé sur les règles de gestion suivantes :

- Un projet est composé de un ou plusieurs référentiels.
- Un référentiel est associé à un seul projet.
- Un référentiel est composé de un ou plusieurs scénarios.
- Un scénario appartient à un seul référentiel.
- Un scénario est composé de un ou plusieurs cas de test.
- Un cas de test appartient à un ou plusieurs scénarios.

5.4.3. Dictionnaire de données

Le tableau suivant présente le dictionnaire de données de notre diagramme de classe :

Classe	Attributs	Type	Signification
Projet	Id_projet	Entier	Identificateur du projet
	NomProjet	Chaine de caractères	Nom du projet
	ProdDesc	Chaine de caractères	Description du produit
	ProjetDesc	Chaine de caractères	Description du projet
	StartDate	Date	Date de début
	EndDate	Date	Date de fin
	NomUser	Chaine de caractères	Nom de l'utilisateur
Référentiel	Id_ref	Entier	Identificateur de référentiel
	NomRef	Chaine de caractères	Nom de référentiel
Scénario	Id_snc	Entier	Identificateur de scénario
	NomSnc	Chaine de caractères	Nom de scénario
	Description	Chaine de caractères	Description de scénario
	NomRapport	Chaine de caractères	Nom du rapport de test
	CheminRapp	Chaine de caractères	Chemin du rapport de test
Cas de test	Id_testcase	Entier	Identificateur de cas de test
	TestCaseID	Chaine de caractères	Nom de cas de test
	TestObjectif	Chaine de caractères	L'objectif de cas de test
	TestSteps	Chaine de caractères	Les étapes de cas de test
	TestData	Chaine de caractères	Les données de cas de test
	ExpectedResult	Chaine de caractères	Le résultat attendu de cas de test
	ActualResult	Chaine de caractères	Le résultat actuel de cas de test
	PassOrFail	Chaine de caractères	Succès ou Échec
	CheminTest	Chaine de caractères	Chemin de cas de test

Tableau III.1 : Dictionnaire de données du diagramme de classe

5.4.4. Modèle relationnel du diagramme de classe

Le modèle relationnel est basé sur une organisation des données sous forme de tables. La manipulation des données se fait selon le concept mathématique de relation de la théorie des ensembles, c'est-à-dire l'algèbre relationnelle. Les opérations relationnelles permettent de créer une nouvelle relation (table) à partir d'opérations élémentaires sur d'autres tables (par exemple

l'union, l'intersection, ou encore la différence) [29]. Le modèle relationnel de notre diagramme de classe est comme suit :

- **Projet** (**Id_projet**, NomProjet, ProdDesc, ProjetDesc, StartDate, EndDate, NomUser).
- **Référentiel** (**Id_ref**, NomRef, **#Id_projet**).
- **Scénario** (**Id_snc**, NomSnc, Description, NomRapport, CheminRapp, **#Id_ref**).
- **Cas de test** (**Id_testcase**, TestCaseID, TestObjectif, TestSteps, TestData, ExpectedResult, ActualResult, PassOrFail, CheminTest, **#Id_snc**)

6. Conclusion

Dans ce chapitre, nous avons présenté notre méthode hybride de test ainsi que l'architecture de notre solution à travers une conception détaillée en utilisant les diagrammes d'UML. Nous avons commencé par démontrer notre méthode hybride de test, ensuite, nous avons définit l'architecture de notre système à travers un schéma global et un diagramme d'activité, puis, nous avons terminé par présenter la modélisation de notre système suivant les diagrammes du langage UML (diagramme de cas d'utilisation, séquence et classe).

Ce chapitre nous a donné une base solide pour passer à l'implémentation et la mise en œuvre de notre solution, qui sera présenté dans le prochain chapitre.

Chapitre IV

Implémentation et Test de la Solution

1. Introduction

Après avoir conçu notre système dans le chapitre précédent, nous allons passer maintenant à l'implémentation de notre outil de test. La phase de l'implémentation et de test est une phase importante car elle concrétise notre étude et permet d'exploiter le travail réalisé.

Ce chapitre sera divisé en deux parties. Dans la première partie, nous commencerons par présenter les différents choix techniques utilisés (plateforme, langage de programmation, SGBD …etc.). Puis, nous allons terminer cette première partie par présenter notre outil de test à travers des captures d'écrans des interfaces relatives aux fonctionnalités principales. Et pour la deuxième partie, nous allons présenter l'application démo (Eri-Bank) que nous allons utiliser pour les tests, ainsi que son plan de test. Ensuite nous allons montrer les différents cas de test que nous allons effectuer. Et enfin nous allons présenter le résultat des tests obtenus à travers des captures d'écrans.

2. Choix Techniques

Lors de développement de notre outil de test, nous avons utilisé les outils et technologies suivantes :

- **Plateforme :** Appium v1.8.0
- **Langage de programmation :** Python 3.6.5
- **SGBD :** MySQL
- **Outils :** PyCharm Edu 2018, Appium Studio, Microsoft Excel 2013, Notepad ++ 7.5.6, Xampp v3.2.2, Qt Designer v5.5.0, Mobizen Mirroring.

2.1. Présentation de la plateforme de test Appium

Appium est un outil Open-Source d'automatisation de test fonctionnel pour les applications mobiles. Il permet de tester les trois types d'applications mobiles : Web, native et hybride. Il permet également d'exécuter les tests automatisés sur des périphériques, des émulateurs ou des simulateurs.

Aujourd'hui, il existe deux plates-formes pour les applications mobiles (iOS et Android). Avoir deux Framework différents pour la même application augmente le coût du produit et le temps nécessaire pour le maintenir.

La philosophie de base d'Appium c'est qu'il est capable de réutiliser le code entre les deux plates-formes iOS et Android, et c'est pourquoi il utilise un API identique sur ces deux plates-

formes. L'outil Appium supporte une variété de langage de programmation pour écrire les tests. Il ne dicte pas le langage ou le Framework à utiliser [33].

2.1.1. Architecture de travail d'Appium

Appium est à la base un serveur web qui expose un API REST. Il reçoit les connexions d'un client, écoute les commandes, exécute ces commandes sur un périphérique mobile et répond par une réponse HTTP représentant le résultat de l'exécution de la commande. Le fait que Appium suit une architecture client / serveur, le testeur peut écrire un code de test dans n'importe quel langage qui utilise l'API client HTTP, mais il est plus facile d'utiliser une des bibliothèques clients d'Appium. De plus, le serveur Appium peut être placé sur une machine différente de celle que les tests sont exécutés [34]. La figure ci-dessous illustre l'architecture de travail du serveur Appium :

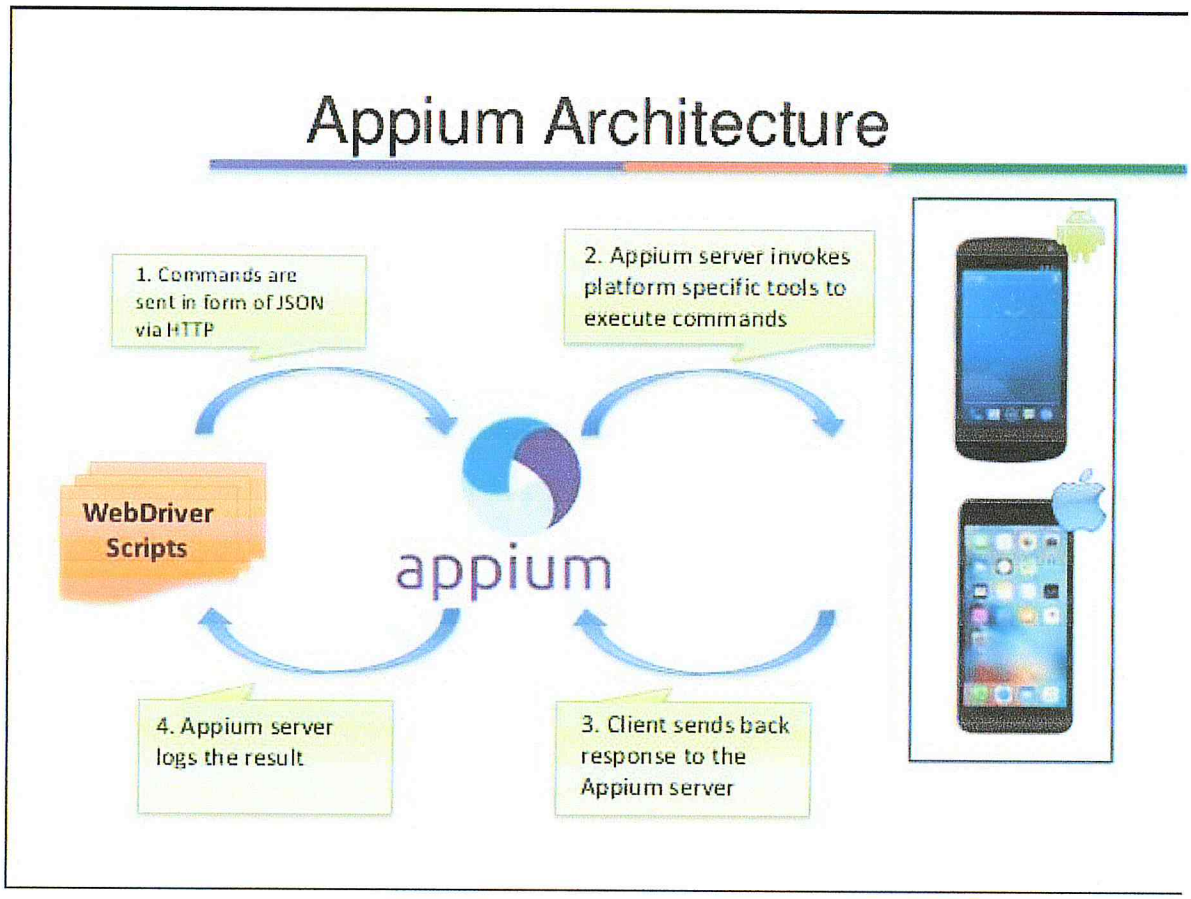

Figure IV.1 : Architecture de travail du serveur Appium [42]

Le déroulement de flux de travail du serveur Appium est décrit comme suit :

1. A partir du pilote Web, les commandes d'automatisation sont envoyées sous format de données JSON[23] via une requête HTTP au serveur Appium.
2. Le serveur Appium invoque à son tour des outils spécifiques à la plate-forme (Android / iOS) pour exécuter ces commandes sur le Smartphone.
3. Le client renvoie le message au serveur Appium.
4. Le serveur Appium enregistre et affiche le résultat dans la console du WebDriver.

Pour pouvoir exécuter les commandes, la phase de Mapping des composants conceptuels avec les composants réels de l'application cible est nécessaire. Pour cela le testeur a besoin d'utiliser un Framework appelé UI Automator.

2.1.2. Présentation de UI Automator Framework

UI Automator est un Framework de test adapté aux tests fonctionnels d'interface utilisateur. Il fournit un ensemble d'API pour créer des tests appliqués sur l'interface utilisateur. Ces API permettent également d'effectuer des opérations telles que l'ouverture du menu paramètres ou du lanceur d'applications dans un périphérique de test. Le Framework de test UI Automator est parfaitement adapté à l'écriture de tests automatisés de type Black-Box, dans lesquels le code de test n'est pas basé sur les détails de l'implémentation interne de l'application cible [35]. Les principales caractéristiques du Framework de test UI Automator sont les suivantes :

- Un API appelé (Viewer) pour inspecter les éléments de l'application à travers la hiérarchie de mise en page.
- Un API pour récupérer des informations d'état et effectuer des opérations sur le périphérique cible.
- Des APIs prenant en charge le test d'interface utilisateur de différentes applications.

L'inspection des composants réels de l'application cible représente une étape essentielle pour pouvoir exécuter les tests. À travers les API fournit par UI Automator Framework, le testeur peut faire le Mapping entre les composants conceptuels et les composants réels.

[23] **JavaScript Object Notation** est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il est permet de représenter de l'information structuré comme le permet XML.

2.2. Langage de programmation Python

Nous avons choisi le langage de programmation Python, car il est puissant et facile à apprendre. Il dispose de structures de données de haut niveau et d'une approche de la programmation orientée objet simple mais efficace. C'est à cause de sa syntaxe élégante, que son typage est dynamique[24] et qu'il est interprété[25]. Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications dans de nombreux domaines et sur de nombreuses plateformes [31].

2.3. SGBD MySQL

Nous avons utilisé le SGBD (Système de Gestion de Base de Données) MySQL car il est libre et gratuit, et aussi c'est un SGBD parmi les plus populaires au monde, de plus il assure un haut niveau de sécurité de la base de données ainsi que des accès concurrents tout en optimisant la taille, le temps d'accès et le temps d'exécution des requêtes.

De plus MySQL est un serveur de base de données relationnelles SQL qui fonctionne sur de nombreux systèmes d'exploitation (dont Linux, Mac OS X, Windows, Solaris, FreeBSD…) et qui est accessible en écriture par de nombreux langages de programmation, incluant notamment Python, Java, Ruby, C, C++, .NET, PHP …etc. [32].

2.4. Choix d'outils

Les outils que nous avons utilisés durant le développement de notre outil de test sont :

- **PyCharm Edu 2018 :** PyCharm Edu un EDI (Environnement de Développement Intégré) dédié au langage Python pour le développement d'applications. Il permet d'éditer le code source tout en bénéficiant de la coloration syntaxique, de suggestions, de débogage et plein d'autres fonctionnalités puissantes.
- **Appium Studio :** Appium Studio est un EDI (Environnement de Développement Intégré) conçu pour le développement et l'exécution des tests automatisés pour les applications mobiles.

[24] **Typage dynamique** consiste à associer à une variable son type dynamiquement sans le déclarer explicitement.
[25] **Langage interprété** veut dire que les instructions sont transcrites en langage machine au fur et à mesure de leur lecture lors de l'exécution.

- **Microsoft Excel 2013 :** Excel est un tableur et permet la création de tableaux, de calculs automatisés, de planning, de graphiques et de bases de données.
- **Notepad ++ :** Notepad est un éditeur de texte libre générique, fonctionnant sous Windows. Il intègre la coloration syntaxique de code source pour presque tous les langages de programmations. Il peut être aussi utilisé pour lire différents types de fichiers.
- **Xampp :** Xamp est un ensemble de logiciels permettant de mettre en place facilement un serveur Web et un serveur FTP. Il s'agit d'une distribution de logiciels libres (X Apache MySQL Perl PHP) offrant une bonne souplesse d'utilisation, réputée pour son installation simple et rapide.
- **Qt Designer :** Qt Designer est un outil pour la conception (design) et la construction d'interfaces graphiques (GUI). Il permet de personnaliser les fenêtres ou boîtes de dialogue d'une manière très simple pour l'utilisateur, et l'utilisateur peut les tester en utilisant différents styles et résolutions.
- **Mobizen Mirroring :** L'outil Mobizen Mirroring permet aux appareils mobiles de se connecter au PC via une application ou un navigateur Web. Une fois connectés, les utilisateurs peuvent visualiser et / ou contrôler l'appareil à partir du PC avec leur souris et leur clavier.

3. Présentation de l'outil de test

Nous allons présenter dans cette section quelques interfaces de notre outil de test à travers des captures d'écran où nous avons choisi les plus importantes. Ces interfaces représentent les fonctionnalités principales de notre outil de test.

3.1. Interface de l'authentification

L'interface de l'authentification fournit la fonctionnalité de l'accès sécurisée à l'outil, les utilisateurs doivent s'identifier avec un nom d'utilisateur et un mot de passe pour pouvoir accéder à l'outil.

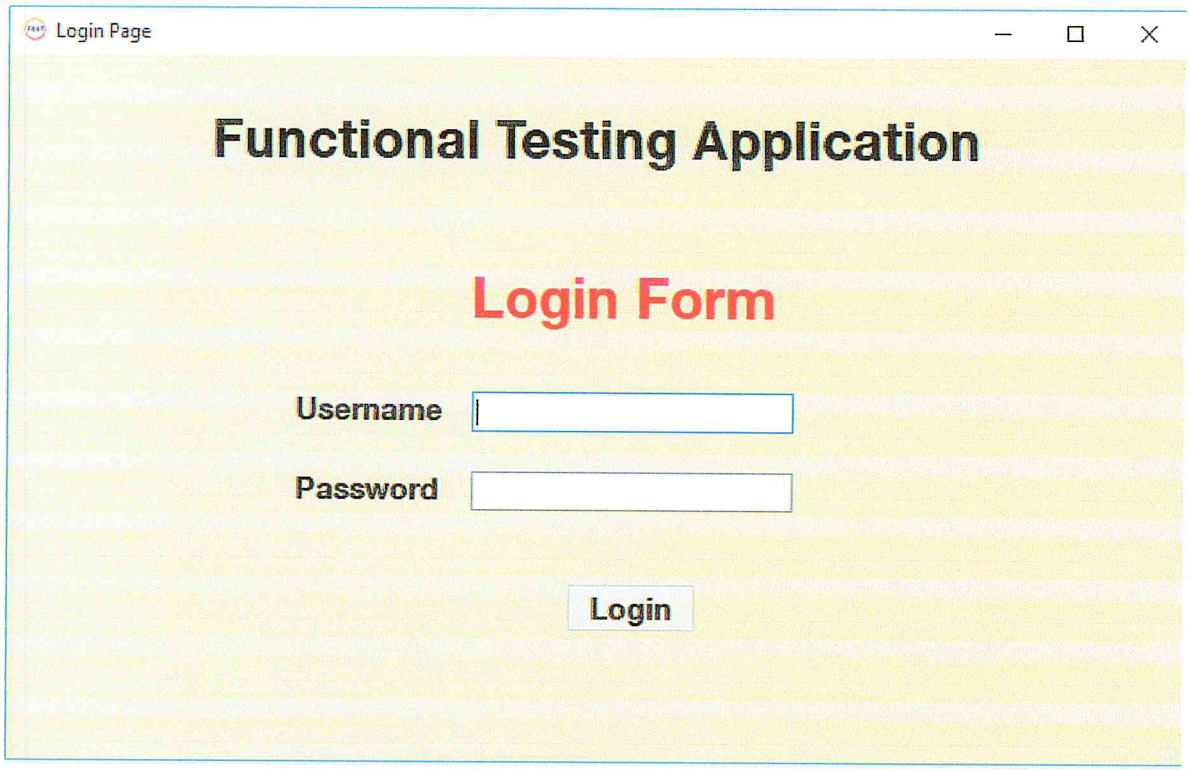

Figure IV.2 : L'interface de l'authentification

3.2. Interface d'accueil

L'interface d'accueil contient le menu principal qui montre les fonctionnalités de base de l'outil de test. La figure ci-dessous illustre cette interface :

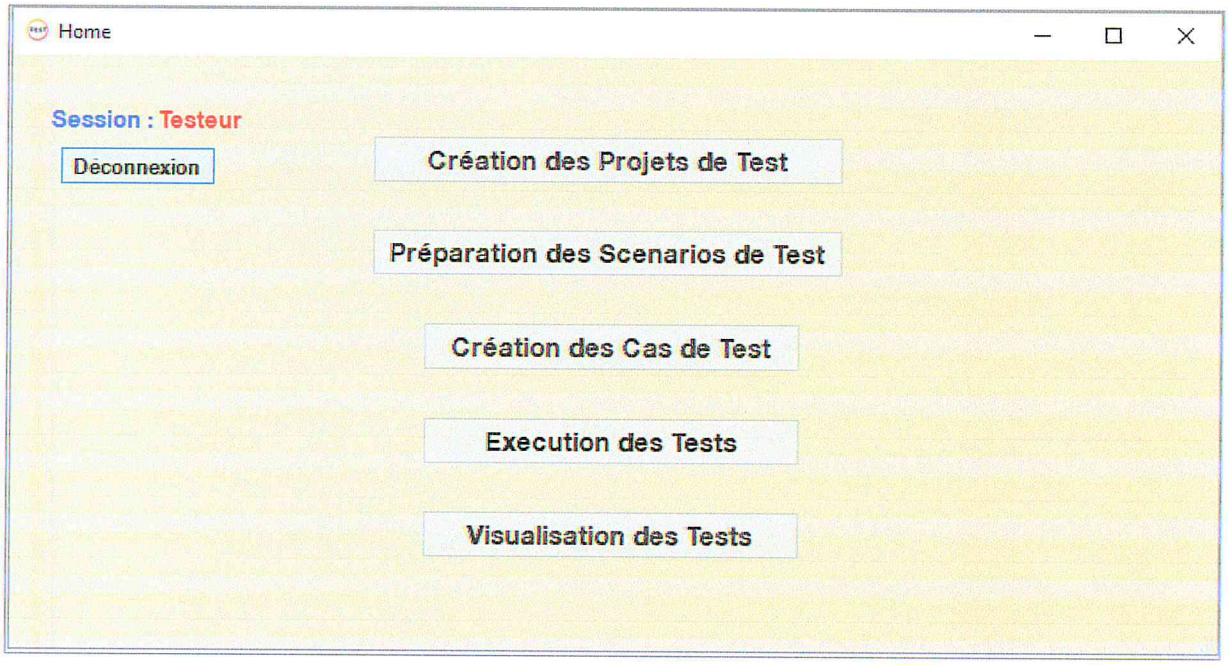

Figure IV.3 : L'interface d'accueil

3.3. Interface de la création des scénarios de test

Cette interface fournit au utilisateur la possibilité de la création / suppression des scénarios de test. La figure ci-dessous illustre cette interface :

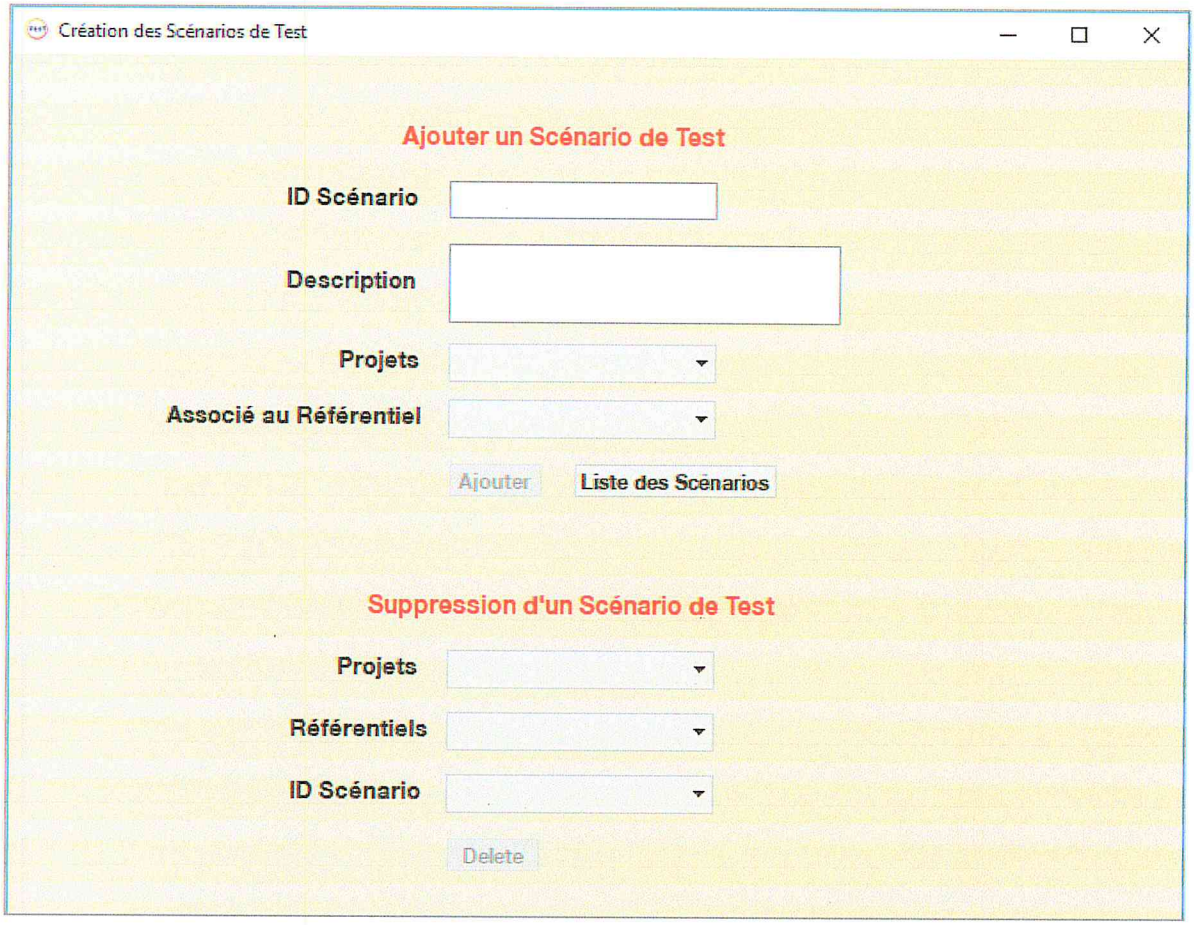

Figure IV.4 : L'interface de la création des scénarios de test

3.4. Interface de la création des cas de test

Cette interface fournit au utilisateur la possibilité de la création / modification / suppression des cas de test pour les associés ensuite aux scénarios de test. La figure ci-dessous illustre cette interface :

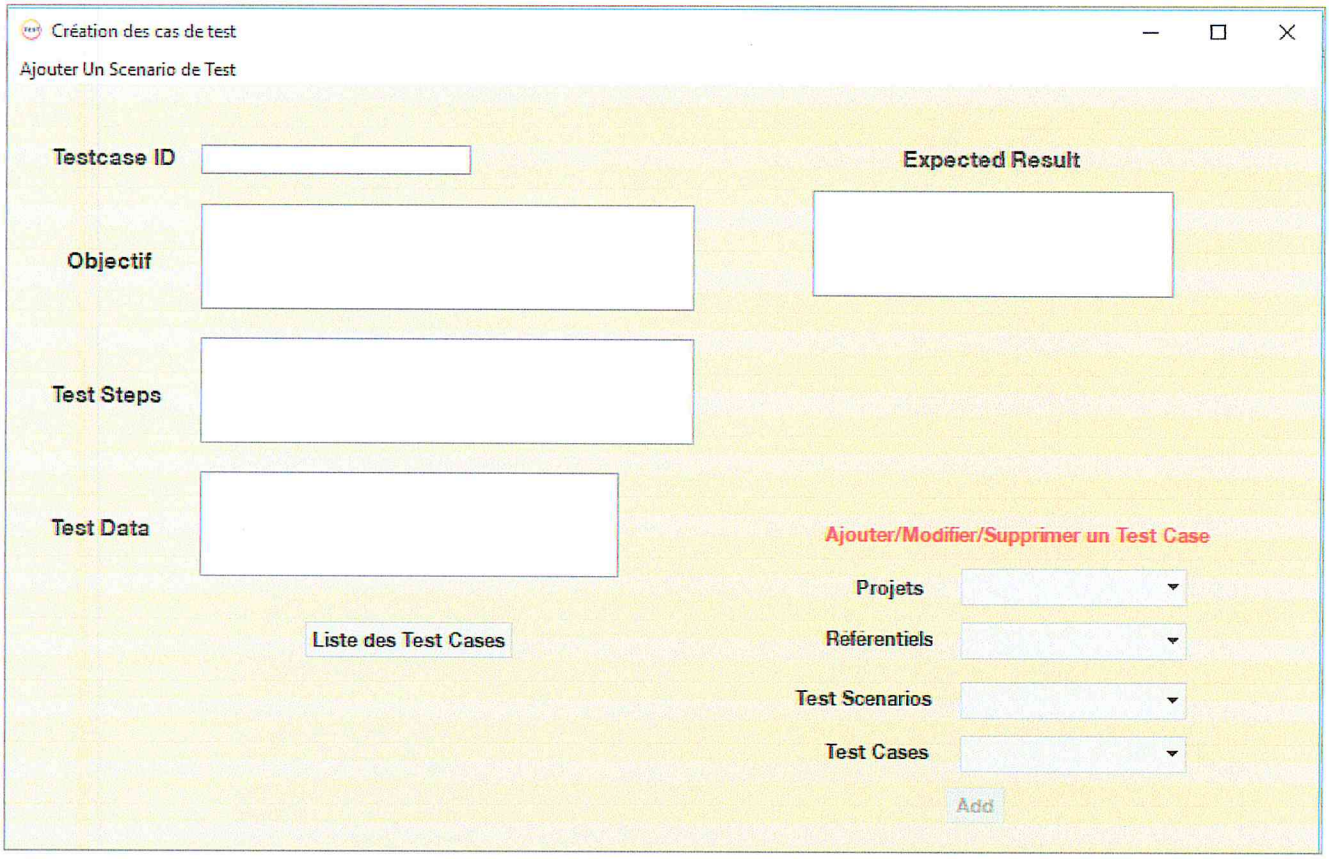

Figure IV.5 : L'interface de la création des cas de test

3.5. Interface de préparation des scénarios de test

Cette interface représente la phase de préparation des scénarios de test. C'est ici que le testeur va modeler ces tests en créant / modifiant / supprimant les composants conceptuels qui représentent les composants réels de l'application cible. Le testeur doit choisir les actions à effectuer sur l'application, et ensuite relier les composants conceptuels créés avec les composants réels de l'application à travers leurs ID / Class, ou la hiérarchie dans lesquels ils se trouvent (XPath[26]). La figure ci-dessous illustre cette interface :

[26] **XPath** est un langage qui décrit un moyen de localiser et de traiter des éléments dans des documents XML en utilisant une syntaxe d'adressage basée sur un chemin à travers la hiérarchie du document

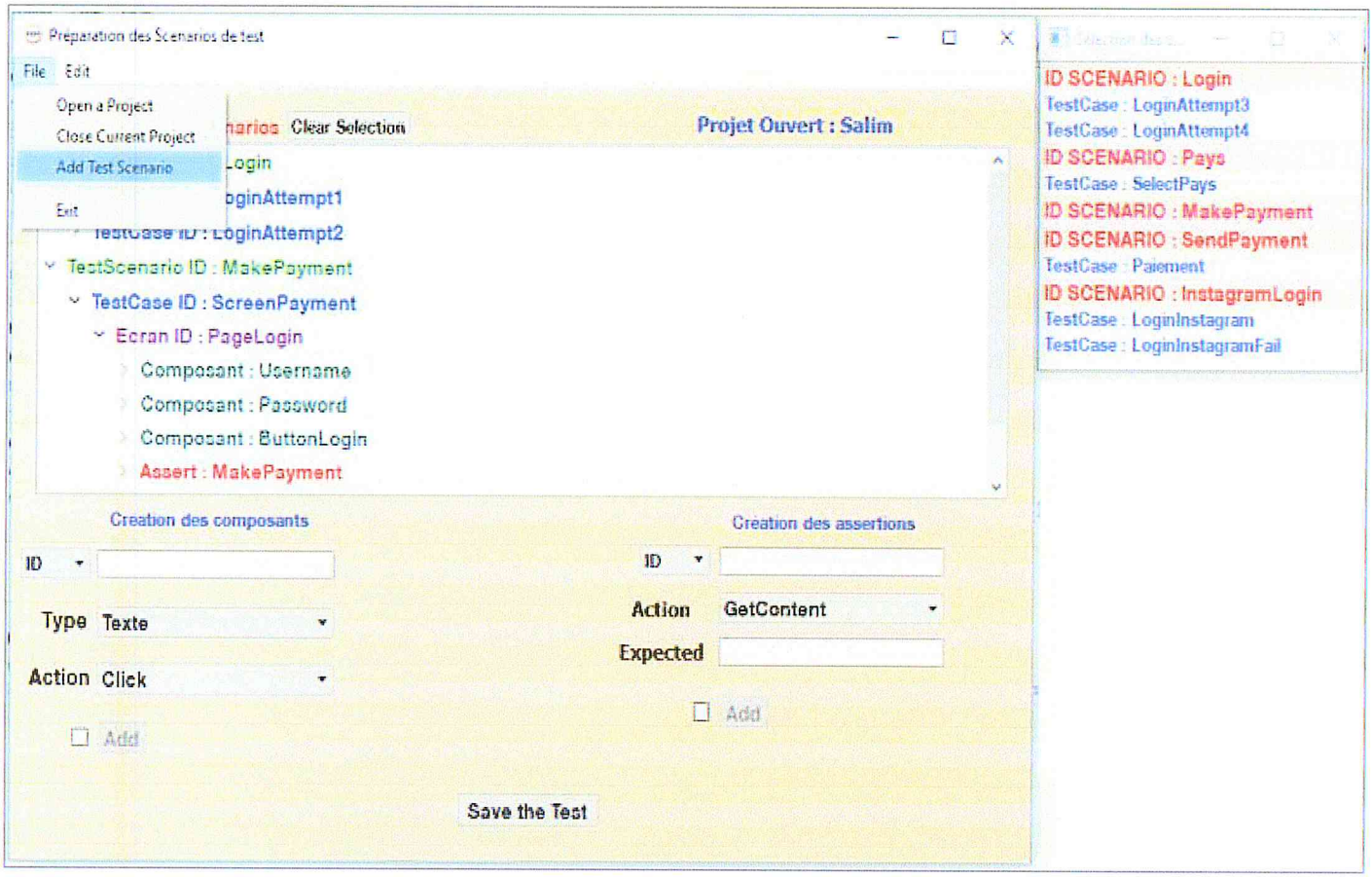

Figure IV.6 : L'interface de préparation des scénarios de test

3.6. Interface de l'exécution des tests

Cette interface représente la phase de l'exécution des tests. L'utilisateur dans cette interface doit ajouter en premier lieu les cas de test qu'il veut exécuter (il peut aussi supprimer les cas de test qu'il ne veut pas exécuter). Ensuite il doit choisir l'application qui va être sujet de test, à travers la précision du chemin du fichier APK de l'application, ou le nom du Package et l'Activity de l'application. Enfin il peut démarrer le test en cliquant sur le bouton Run Test. Et donc, le serveur Appium va exécuter le test sur le Smartphone en lançant l'application cible, puis retourne le résultat de l'exécution sur la console, et le rapport des tests sur un fichier Excel détaillé. La figure ci-dessous illustre l'interface de l'exécution des tests :

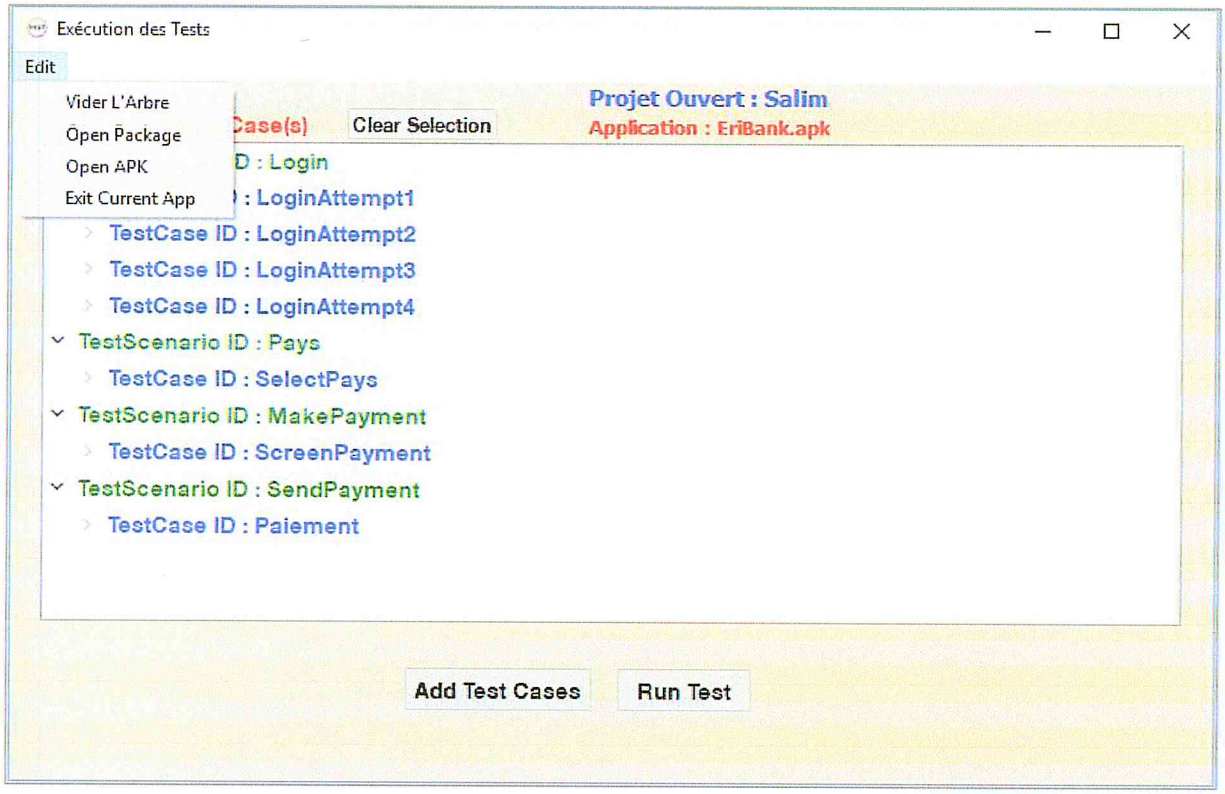

Figure IV.7 : L'interface de l'exécution des tests

Pour la visualisation / contrôle de notre appareil mobile depuis notre PC durant l'exécution des tests, nous avons utilisé l'outil Mobizen Mirroring. La figure ci-dessous présente l'écran de notre appareil visualisé sur notre PC :

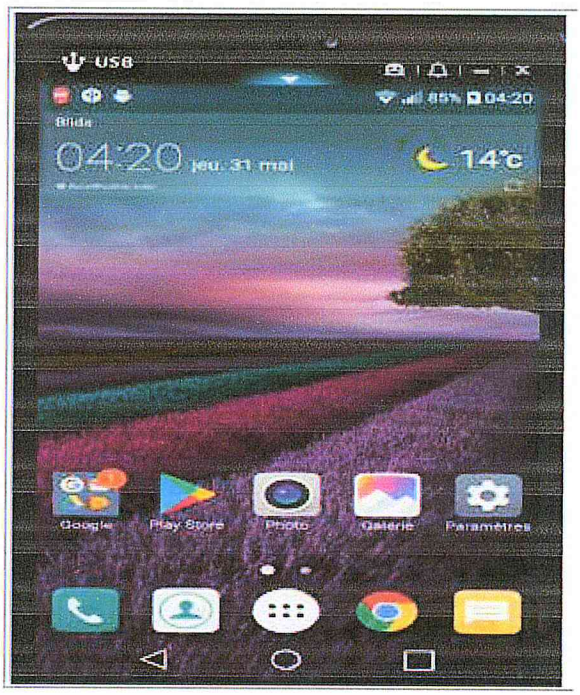

Figure IV.8 : L'écran de l'appareil mobile visualisé sur le PC

3.7. Interface du serveur Appium

Cette interface représente le serveur Appium dans un état d'écoute des commandes qui arrive depuis le WebDriver Script. Si le test démarre, le serveur Appium va recevoir les commandes envoyées depuis le WebDriver Script et lance l'application cible dans le Smartphone / émulateur / simulateur, pour commencer à les exécuter. Les commandes exécutées représentent les actions que l'utilisateur a choisies lors de la création des composants conceptuels. La figure suivante illustre cette interface :

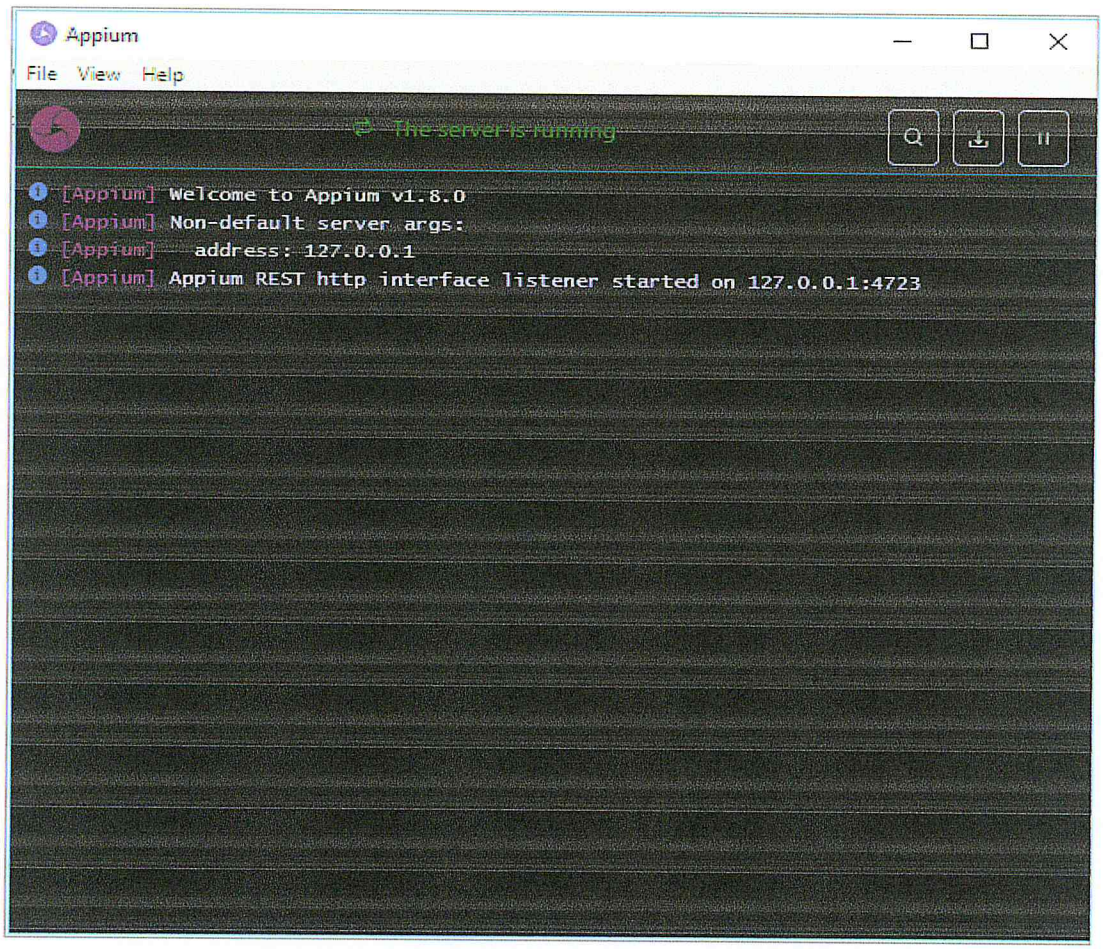

Figure IV.9 : L'interface du serveur Appium

3.8. Interface de l'API Viewer

Cette interface représente l'API Viewer, qui est chargé d'inspecter les composants réels de l'application cible. L'utilisateur ici, peut pour récupérer leurs identifiants (ID, XPath …etc.) ainsi que d'autres informations utiles, comme la position des composants dans l'écran ou les attributs des composants. La figure ci-dessous illustre cette interface :

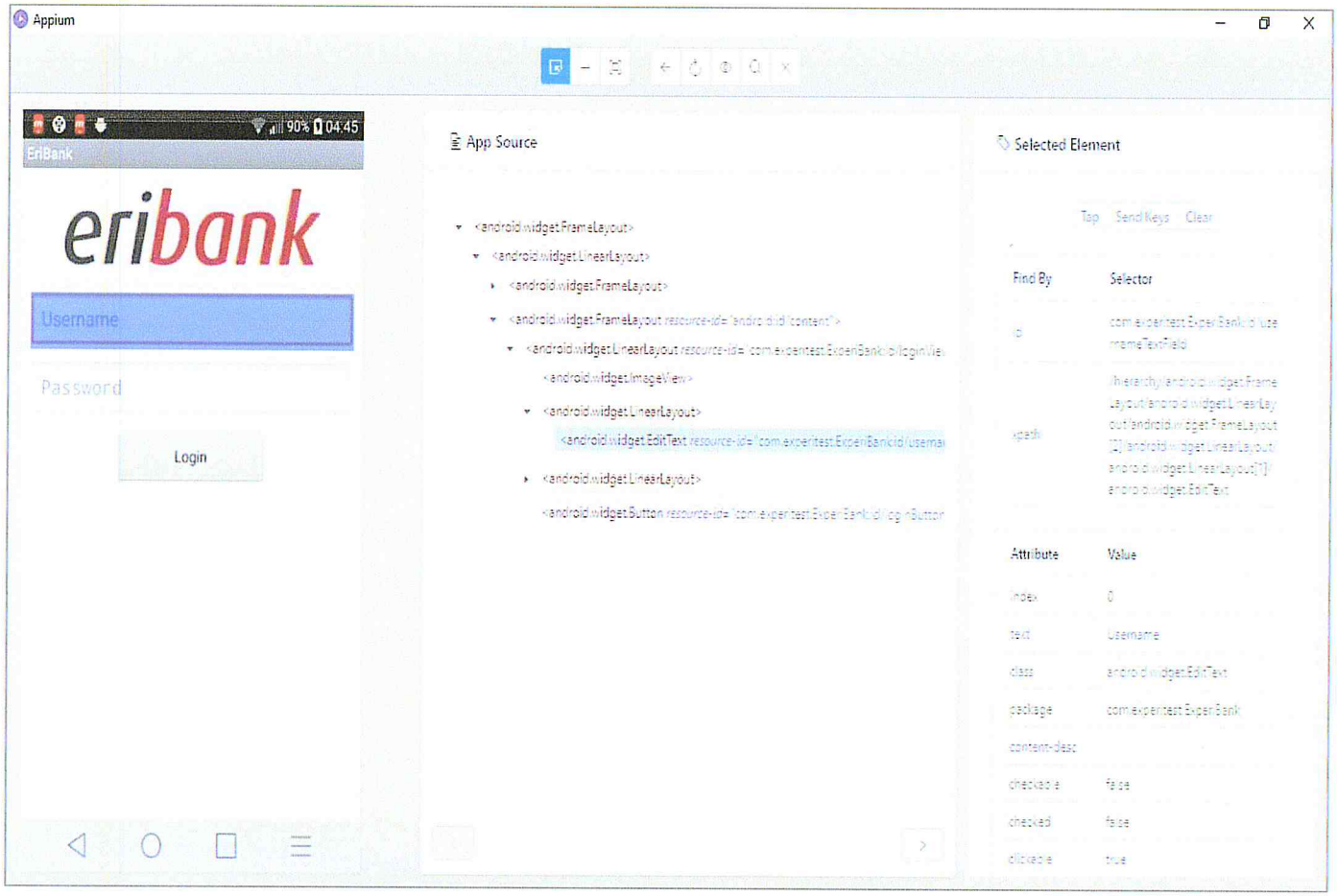

Figure IV.10 : L'interface de l'API Viewer

4. Phase de test et résultat

Dans cette phase, nous allons présenter l'application démo Eri-Bank, ainsi que son plan de test, et enfin nous allons montrer les résultats des cas de test exécutés sur les différentes fonctionnalités de cette application.

4.1. Présentation de l'application démo Eri-Bank

Eri-Bank est une application démo instrumentée à laquelle on peut conduire des tests fonctionnels. Elle contient des fonctionnalités relatives aux opérations bancaires telles que l'authentification, le paiement, le retrait de l'argent …etc. Le but de notre choix de cette application, est qu'elle offre des fonctionnalités basiques et elle convient pour conduire des tests avec notre outil, avant de généraliser l'utilisation de notre l'outil sur les différentes applications Android. La figure ci-dessous montre l'interface principale de cette application :

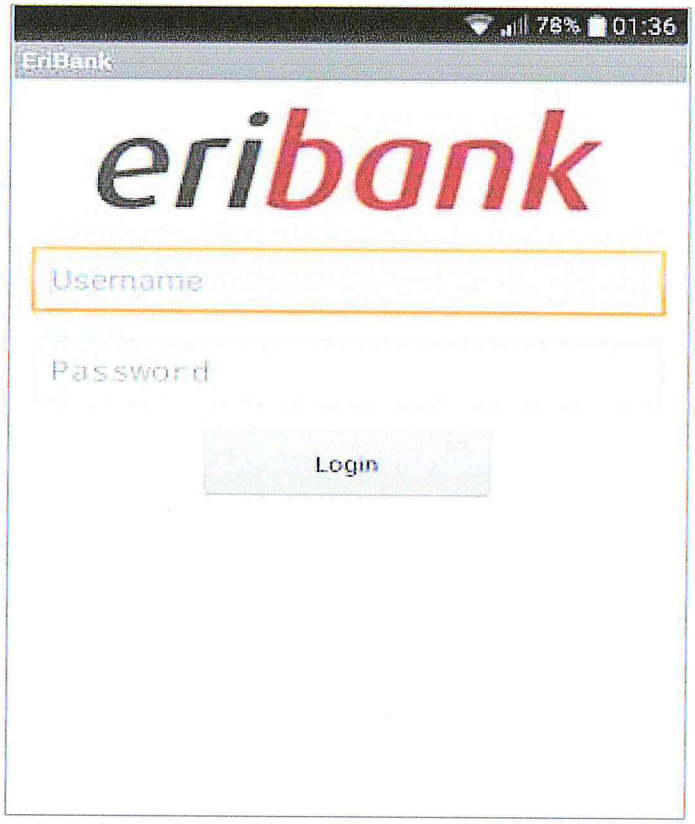

Figure IV.11 : L'interface de l'application de test démo Eri-Bank

4.1.1. Plan de test de l'application démo Eri-Bank

Le plan de test est conçu pour prescrire la portée, l'approche, les ressources et le calendrier de toutes les activités de test d'un projet. Il identifie les éléments à tester, les fonctionnalités à tester, les types de tests à effectuer, le personnel responsable des tests, les ressources et le calendrier requis pour effectuer les tests, ainsi que les risques associés au plan.

Les tests que nous avons effectués sur l'application démo Eri-Bank sont en fait basés sur le plan de test fonctionnel d'une application similaire appelé Guru99 Bank.

4.1.2. Stratégie de test

La stratégie de test est définie comme un ensemble de principes directeurs qui éclaircit la conception des tests et régulent la façon dont les tests doivent être effectués. L'élaboration de la stratégie dans un plan de test est nécessaire pour l'organisation des tests.

4.1.2.1. Fonctionnalités à tester

Toutes les fonctionnalités de l'application Eri-Bank doivent être testées. Le tableau suivant décrit les fonctionnalités principales de l'application :

Fonctionnalité	Description
Authentification	Un utilisateur peut s'authentifier à l'application en tapent le username & password correcte. Après l'authentification, Il doit voir la fenêtre d'accueil.
Erreur d'authentification	Un utilisateur ne peut pas accéder à l'application s'il insère un username & password invalide, il doit voir aussi un message d'erreur.
Logout	Un utilisateur peut revenir à la fenêtre d'authentification en cliquant sur le bouton Logout.
Paiement	Un utilisateur doit voir la balance actuelle après l'authentification, et il peut accéder à la fenêtre de paiement en cliquant sur le bouton Make Payment.
Confirmer un paiement	Un utilisateur peut faire un paiement après l'accès à la fenêtre de paiement en remplissant le n° de tél, le nom, la somme d'argent, le pays. À la fin Il clique sur Send Payment, puis Yes pour effectuer le paiement. Il doit être redirigé vers la fenêtre d'accueil.
Annuler un paiement	Un utilisateur peut annuler un paiement en cliquant sur le bouton No dans la boite de dialogue qui s'affiche après le clic sur le bouton Send Payment.
Quitter la fenêtre de paiement	Un utilisateur peut sortir de la fenêtre paiement en cliquant sur le bouton Cancel. Il doit être redirigé vers la fenêtre d'accueil.
Paiement interdit	Un utilisateur ne doit pas être capable de faire un paiement s'il ne remplit pas toutes les informations nécessaires.
Balance exacte	Un utilisateur doit voir une balance exacte après avoir effectué un paiement.
Choix des pays	Un utilisateur peut changer le pays qu'il a choisi en cliquant sur le bouton Select.

Tableau IV.1 : Les fonctionnalités à tester de l'application Eri-Bank

4.1.2.2. Tests à négliger

Les tests suivants ne vont pas être effectués, car ils n'ont pas une relation directe avec notre test fonctionnel sur l'application Eri-Bank :

- Sécurité.
- Performance.
- Interfaces matérielles.
- Base de données.

4.1.2.3. Risque et problèmes

Le risque est la probabilité d'occurrence d'un événement indésirable durant les tests associé au plan. Le tableau suivant décrit quelques risques ainsi que leur solution proposé :

Risque	Solution
L'utilisateur n'a pas les compétences requises pour les tests.	Planifiez une formation pour l'utilisateur.
Le calendrier du projet est trop serré, il est difficile de terminer les tests à temps.	Définir une priorité de test pour chacune des activités de test.
L'utilisateur n'est pas familier avec l'outil de test.	Définir un guide d'utilisation dans l'outil de test.

Tableau IV.2 : Les risques associés au plan de test

4.1.3. Objectifs de test

Les objectifs du test visent à vérifier les fonctionnalités de l'application démo Eri-Bank, le projet devrait se concentrer sur le test des opérations bancaires telles que, le retrait, le paiement ... etc. pour garantir que toutes ces opérations fonctionnent comme attendus.

4.1.4. Critère de test

Le critère de test spécifie les critères qui indiquent la réussite d'une phase de test. Tous les tests doivent être exécutés pour compléter ce critère. Généralement, si le rapport de test indique que 40 % des cas de test ont échoué, alors les tests doivent être suspendus jusqu'à ce que l'équipe de développement corrige tous les cas ayant échoué.

4.1.5. Plan de ressources

Le plan de ressources est un résumé détaillé de tous les types de ressources nécessaires pour accomplir la tâche du projet. Le tableau suivant résume les ressources qui ont été nécessaires pour tester l'application Eri-Bank :

Ressource	Description
Laptop	Nécessite un Laptop exécutant Windows 7 ou supérieur, Ram 4 Go ou plus, CPU i5 ou plus.
Smartphone	Besoin d'un Smartphone exécutant l'OS Android, avec des capacités (Ram 2 Go, CPU Snapdragon 820 octa-core ou équivalent).
Connexion	Nécessite une connexion stable (ADSL avec un débit minimum de 4 Mb/s, ou la 4G)

Tableau IV.3 : Le plan de ressources

4.2. Présentation de résultat final d'exécution des tests

Après avoir définit le plan de test de l'application Eri-Bank, nous allons maintenant passer à l'exécution des tests pour but de voir si les fonctionnalités de l'application fonctionnent comme prévues. Les figures suivantes présentent le résultat de l'exécution des tests sur l'application Eri-Bank :

```
Exécution de Scénario :  Login


Exécution de TestCase :  Login_Attempt1

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
TestCase : Login_Attempt1 Pass

Exécution de TestCase :  Login_Attempt2

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
TestCase : Login_Attempt2 Pass

Exécution de TestCase :  Logout

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
TestCase : Logout Pass
```

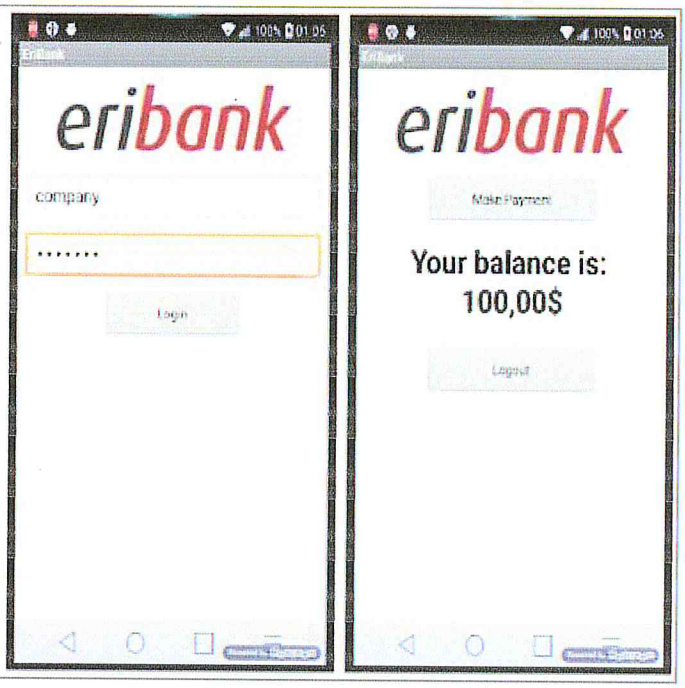

Figure IV.12 : Résultat d'exécution des tests (partie 1)

```
Exécution de TestCase :  Screen_Payment

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Payment ***
TestCase : Screen_Payment Pass

Exécution de TestCase :  Cancel_Screen_Payment

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Payment ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
TestCase : Cancel_Screen_Payment Pass
```

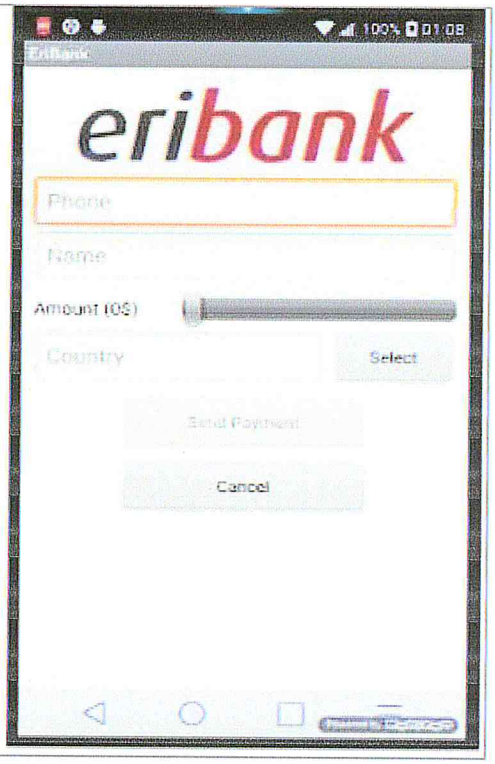

Figure IV.13 : Résultat d'exécution des tests (partie 2)

```
Exécution de TestCase :  Paiement

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Payment ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
TestCase : Paiement Pass

 Exécution de Scénario :  Pays


Exécution de TestCase :  SelectPays

*** Vous êtes maintenant dans l'écran : Ecran ID : PageLogin ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Home ***
*** Vous êtes maintenant dans l'écran : Ecran ID : Payment ***
Element  com.experitest.ExperiBank:id/countryTextField  Not Found

Assertion On Element  com.experitest.ExperiBank:id/cancelButton  Failed

TestCase : SelectPays Fail
```

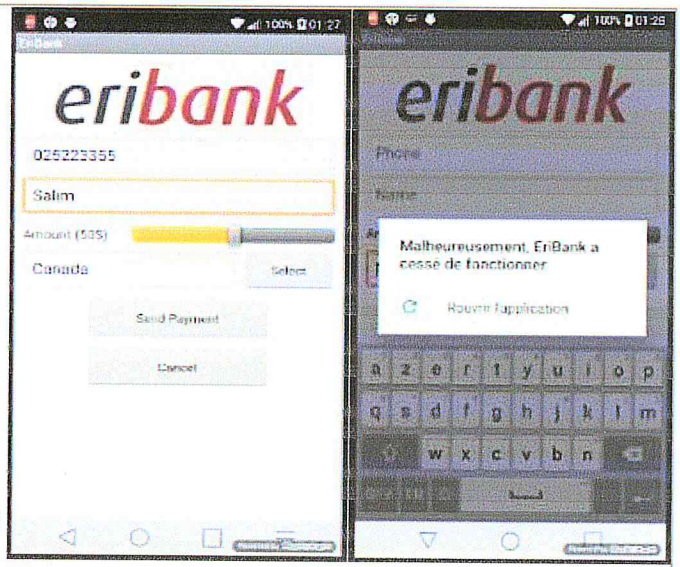

Figure IV.14 : Résultat d'exécution des tests (fin)

Le rapport du résultat final des tests est enregistré dans un fichier sous format Excel. Les figures ci-dessous illustrent ce rapport :

Test Scenario ID : Login		Description : Tester la fonctionnalité d'authentification				
TestCase ID	**TestOjbectif**	**TestSteps**	**TestData**	**ExpectedResult**	**ActualResult**	**Pass / Fail**
LoginAttempt1	Test the login	Insert Username Insert Password Click Button Login	User = company Password = company	Login successful		Pass
LoginAttempt2	Test the login	Insert Username Insert Password Click Button Login	User = company Password = salim	Login Fail		Pass
Logout	Tester la déconnexion de l'application	1- Insérer User & Password 2- Cliquer sur le bouton Login 3- Cliquer sur le bouton Logout	User = company Pass = company	Logout avec succès		Pass
Test Scenario ID : PaymentWindow		**Description : Tester l'affichage de l'écran de payment**				
TestCase ID	**TestOjbectif**	**TestSteps**	**TestData**	**ExpectedResult**	**ActualResult**	**Pass / Fail**
ScreenPayment	Avoir accès a l'écran payment	Insert Username & Pass Click Login Click MakePayment	User = company Pass = company	Affichage de l'écran Payment		Pass
CancelScreenPayment	Tester si le client peut sortir de la fenêtre de paiement	1- Insérer User & Pass et clic Login 2- Cliquer sur le bouton Make Payment 3- Cliquer sur le bouton Cancel	User = company Pass = company	Affichage de la fenêtre d'accueil		Pass

Figure IV.15 : Rapport du résultat final des tests (partie 1)

Test Scenario ID : SendPayment		Description : Tester la fonctionnalité de paiement				
TestCase ID	TestObjectif	TestSteps	TestData	ExpectedResult	ActualResult	Pass / Fail
Paiement	Faire un paiement	Insert Username & Password Click Login , Click Make Payment Remplir les informations Click Send Payment	Username = company Password = company Amount $ = 50	Un paiement et une nouvelle Balance correcte		Pass
Test Scenario ID : Pays		**Description : Tester la fonctionnalité de sélection des pays**				
TestCase ID	TestObjectif	TestSteps	TestData	ExpectedResult	ActualResult	Pass / Fail
SelectPays	Sélectionner différents pays	Login & Click Make Payment Click Select & Choisir un Pays	User = company Pass = company Pays = Quelconque	Affichage du pays sélectionné dans l'input (pays)		Fail

Figure IV.16 : Rapport du résultat final des tests (fin)

4.3. Diagnostique du résultat du test

L'exécution du test été réussie avec un résultat qui indique que les cas de test ont aboutis, à l'exception d'un seul cas de test qui a échoué, il représente la fonctionnalité de sélection des pays. L'échec du cas de test signifie la découverte d'un comportement non désiré (bug) dans l'application Eri-Bank.

Nous avons aussi effectué un test sur la fonctionnalité de l'authentification d'une autre application de réseau social appelé Instagram. L'exécution du test sur cette application a été réussie sans erreurs, ce qui prouve la validité de notre outil de test, et qu'elle peut être généralisée sur les différentes applications sur la plate-forme Android.

5. Conclusion

Tout au long de ce chapitre qui clôture notre mémoire nous avons présenté les outils que nous avons utilisés pour la réalisation de notre système, ainsi que la présentation de notre solution via des captures d'écran des interfaces les plus significatives, et enfin nous avons présenté le résultat de l'exécution des tests et le rapport final des tests.

Mais au-delà de cela, il est évident que la réalisation n'était et ne sera jamais l'étape finale du processus de développement, car il faut continuer à suivre notre système et le superviser, et cela par essayer toutes ces fonctionnalités avec plusieurs jeux de test, dans le but de détecter les éventuels bugs et anomalies et les rectifier pour assurer sa stabilité et sa fiabilité.

CONCLUSION GENERALE

Dans ce mémoire nous avons présenté le travail réalisé dans le cadre de notre projet de fin d'études, et dont l'objectif était « Une approche d'automatisation de test fonctionnel pour les applications mobiles ».

Le concept de test fonctionnel est devenu de plus en plus utilisé pour répondre à la complexité croissante des situations rencontrées par les entreprises. Ce concept s'avère le plus efficace pour tester et évaluer un logiciel, dont il résulte à une meilleure qualité de ce dernier et un retour sur investissement valable.

Cependant, les entreprises font face toujours à des problèmes résultant des tests manuels notamment les tests fonctionnels. Ils s'avèrent fastidieux en matière de perte de temps, de la mauvaise couverture de test et les erreurs humaines.

Ainsi, l'outil de test que nous avons proposé va permettre une amélioration tangible de la qualité en automatisant le processus du test fonctionnel, de l'élaboration et l'organisation du plan de test jusqu'à l'exécution des tests.

Afin de répondre à cette problématique, notre travail est passé par plusieurs étapes.

En premier lieu, nous avons présenté une étude théorique sur le processus de test et qualité du logiciel. Ensuite, nous avons fait une étude comparative des outils d'automatisation de test et mis en évidence les recherches académiques sur les méthodes d'automatisation de test.

Partant de la dernière étape, nous avons présenté notre méthode hybride de test qui été basée sur les méthodes de génération de test automatique que nous avons détaillés dans la partie des recherches académiques. Notre méthode nous a offert une bonne organisation des tests, et une grande flexibilité dans leurs manipulations ainsi que la facilité d'automatiser l'exécution de ces tests. De plus, l'avantage de la représentation des scénarios de test sous forme d'une arborescence symbolique (textuelle) dans notre méthode par rapport aux autres méthodes de génération de test est qu'elle est compréhensible par n'importe quel utilisateur (testeur ou autre).

Enfin, nous avons présenté les différents tests effectués par notre outil pour valider notre solution. Bien que les résultats des tests étaient satisfaisantes, nous avons rencontré des difficultés qui nous ont permis de fournir d'avantage d'efforts afin d'achever le travail dans les délais impartis et selon la qualité demandée. Parmi eux, le manque de documentation, la difficulté de maitriser les outils et technologies que nous avons utilisés, la difficulté de visualiser les tests graphiquement et en temps réel, la difficulté de générer des données de test automatiquement ainsi que le manque des applications (démo) pour conduire nos tests avant de passer à des applications (Android) réelles.

Ce projet a fait l'objet d'une expérience intéressante, qui nous a permis non seulement d'améliorer nos connaissances et nos compétences dans le domaine de test et qualité du logiciel, mais aussi de concrétiser nos connaissances théoriques acquises pendant le cursus universitaire.

Bien que notre solution soit opérationnelle, le travail est toujours appelé à être amélioré et enrichi. De ce fait, nous avons pensés à quelques perspectives :

- Généraliser notre outil pour prendre en charge les tests sur les différentes applications, y compris les applications de la plateforme iOS.
- Implémenter d'autres fonctionnalités, comme par exemple, la fonction de Record/Playback qui consiste à enregistrer les actions performé sur l'application cible par l'utilisateur automatiquement.
- Améliorer la conception visuelle des tests, qui est sous format d'un arbre textuel, en un format graphique et dynamique.
- Améliorer notre méthode hybride pour générer des données de test automatiquement.

BIBLIOGRAPHIE

[1] Janssen, D. and Janssen, C. (2010). *What is a Mobile Application? - Definition from Techopedia.* [online] Techopedia.com. Disponible à : https://www.techopedia.com/definition/2953/mobile-application-mobile-app [Accédé le 25 Mar. 2018].

[2] Communication, T. (2016). *Choisissez une application mobile plus ergonomique et attractive.* [online] Taktilcommunication.com. Disponible à : https://www.taktilcommunication.com/blog/applications-mobile/definition-typologie-applications-mobiles.html [Accédé le 25 Mar. 2018].

[3] IBM. (n.d.). *Types d'application mobile.* [online] Disponible à : https://www.ibm.com/support-knowledgecenter/fr/SS8H2S/com.ibm.mc.doc/dev_source/references/dev_about_app_types.htm [Accédé le 25 Mar. 2018].

[4] Radatz, J. (1990). *610.12-1990 IEEE Standard Glossary of Software Engineering Terminology.* New York : The Institute of Electrical and Electronics Engineers, p.74.

[5] Myers, G., Badgett, T., Thomas, T. and Sandler, C. (2004). *The art of software testing.* 2nd ed. Hoboken, N.J. : John Wiley & Sons, p.8.

[6] Janssen, D. and Janssen, C. (2016). *What is Software Testing ?* - Definition from Techopedia. [online] Techopedia.com. Disponible à : https://www.techopedia.com/definition/17681/software-testing [Accédé le 27 Mar. 2018].

[7] Coudert, O. (2011). *What is software quality ?.* [online] Olivier Coudert's Blog. Disponible à : http://www.ocoudert.com/blog/2011/04/09/what-is-software-quality/ [Accédé le 28 Mar. 2018].

[8] ISTQB. (2017). *What is fundamental test process in software testing ?.* [online] Disponible à : http://istqbexamcertification.com/what-is-fundamental-test-process-in-software-testing/ [Accédé le 28 Mar. 2018].

[9] Black, R., Veenendaal, E. and Graham, D. (2017). *Foundations of software testing.* Andover : Cengage Learning EMEA, pp.24-49.

[10] Bradford, L. (2017). *Everything You Need to Know About Software Testing Methods.* [online] The Balance Careers. Disponible à : https://www.thebalance.com/all-you-need-to-know-about-software-testing-methods-4019921 [Accédé le 29 Mar. 2018].

[11] Hooda, I. and Singh Chhillar, R. (2015). Software Test Process, Testing Types and Techniques. *International Journal of Computer Applications*, 111(13), pp.2-3.

[12] Rätzmann, M. and De Young, C. (2003). *Software testing and internationalization.* Salt Lake City : Lemoine International, Inc., pp.49-55.

[13] Mohtashim, M. (2016). *Software Testing - software system evaluation.* [ebook] Telangana : Tutorials Point Company, pp.19-20. Disponible à : https://www.tutorialspoint.com/software_testing/software_testing_tutorial.pdf [Accédé le 2 Apr. 2018].

[14] Thévenod-Fosse, P., & Waeselynck, H. (1993). On functional statistical testing designed from software behavior models. In *Dependable Computing for Critical Applications 3* (pp. 1-3). Springer, Vienna.

[15] Hcltech. (n.d.). *What is Automation? | HCL Technologies.* [online] Disponible à : https://www.hcltech.com/technology-qa/what-is-automation [Accédé le 4 Apr. 2018].

[16] Janssen, D. and Janssen, C. (n.d.). *What is Automated Testing? - Definition from Techopedia.* [online] Techopedia.com. Disponible à : https://www.techopedia.com/definition/17785/automated-testing [Accédé le 4 Apr. 2018].

[17] Sharma, L. (2016). *Manual Testing Process Life Cycle in Software Testing.* [online] Toolsqa.com. Disponible à : http://toolsqa.com/software-testing/manual-testing/ [Accédé le 5 Apr. 2018].

[18] Software Testing Class. (2014). *What is Automation Testing? - Software Testing Class.* [online] Disponible à : http://www.softwaretestingclass.com/what-is-automation-testing/ [Accédé le 7 Apr. 2018].

[19] Donaldson, W. (2017). *What's the Difference Between Automated Testing and Manual Testing?.* [online] dzone.com. Disponible à : https://dzone.com/articles/automated-testing-vs-manual-testing [Accédé le 10 Apr. 2018].

[20] Dustin, E., Rashka, J., & Paul, J. (1999). *Automated software testing: introduction, management, and performance.* Addison-Wesley Professional.

[21] Islam, Nazia, "A Comparative Study of Automated Software Testing Tools" (2016). *Culminating Projects in Computer Science and Information Technology.* 12., pp.20.

[22] Rouse, M. (2016). *Que signifie Open Source ?.* [online] LeMagIT. Disponible à : http://www.lemagit.fr/definition/Open-Source [Accédé le 13 Apr. 2018].

[23] Halpanet.org. (n.d.). *Logiciels libres et propriétaires*. [online] Disponible à : https://www.halpanet-.org/content/logiciels-libres-propri-taires [Accédé le 14 Apr. 2018].

[24] Kangoye, S. (2016). *Elaboration d'une approche de vérification et de validation de logiciel embarqué automobile, basée sur la génération automatique de cas de test* (Doctoral dissertation, Angers)., pp.41-65.

[25] Păsăreanu, C. S., & Visser, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4), 339.

[26] Schneider, G. (n.d.). *Model Checking: A Complement to Test and Simulation*. [online] Cse.chalmers-.se. Disponible à : http://www.cse.chalmers.se/~gersch/model-checking/What-is-Model-Checking-.html [Accédé le 20 Apr. 2018].

[27] CommentCaMarche. (2018). *Introduction à UML*. [online] Disponible à : https://www.commentcama-rche.com/contents/1141-introduction-a-uml [Accédé le 23 Apr. 2018].

[28] Gabay, J. and Gabay, D. (2008). *UML 2*. Paris: Dunod, pp.32-34 & 76-78 & 105-107.

[29] CommentCaMarche. (2018). *Le modèle relationnel*. [online] Disponible à : https://www.commentca-marche.com/contents/1013-le-modele-relationnel [Accédé le 5 May 2018].

[30] De Ridder, A. (2018). *Depth First Search Algorithm: What it is and How it Works*. [online] Edgy Labs. Disponible à : https://edgylabs.com/depth-first-search-algorithm-what-it-is-and-how-it-works [Accédé le 14 May 2018].

[31] Docs.python.org. (2017). *Le tutoriel python - documentation Python 3.5.4*. [online] Disponible à : https://docs.python.org/fr/3.5/tutorial/ [Accédé 20 May 2018].

[32] Brouard, F. (n.d.). *MySQL*. [online] SQL. Disponible à : http://sql.sh/sgbd/mysql [Accédé le 25 May 2018].

[33] Verma, N. (n.d.). *What is Appium · Appium for Android*. [online] Nishantverma.gitbooks.io. Disponible à : https://nishantverma.gitbooks.io/appium-for-android/appium/why_appium.html [Accédé le 27 May 2018].

[34] Appium.io. (n.d.). *Introduction - Appium*. [online] Disponible à : http://appium.io/docs/en/about-appium/intro/ [Accédé le 27 May 2018].

[35] Android Developers. (2018). *UI Automator*. [online] Disponible à : https://developer.android.com-/training/testing/ui-automator [Accédé le 28 May 2018].

[36] B, R. (n.d.). *Time / Space Complexity*. [online] Btechsmartclass.com. Disponible à : http://btechsmartclass.com/DS/U1_T4.html [Accédé le 6 Jun. 2018].

[37] Notes on the Complexity of Search. (2003). [ebook] Cambridge: MASSACHUSETTS INSTITUTE OF TECHNOLOGY. Disponible à : http://www.ai.mit.edu/courses/6.034b/searchcomplex.pdf [Accédé le 7 Jun. 2018].

[38] Bell, R. (2009). A beginner's guide to Big O notation. [online] Rob-bell.net. Disponible à : https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/ [Accédé le 7 Jun. 2018].

[39] Hanna, M., El-Haggar, N., & Sami, M. (2014). A review of scripting techniques used in automated software testing. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *5*(1).

[40] Grolleau, E. (2016). *Tests logiciels*. [ebook] Paris: LESIA, pp.49 - 66. Disponible à : https://sites.lesia.obspm.fr/emmanuel-grolleau/files/2016/07/Master_OSAE_Cours_Tests_Grolleau.pdf [Accédé le 21 Apr. 2018].

[41] Cousin, N. (2012). *Test Fonctionnel vs Test Structurel?*. [online] nicolas-cousin.com. Disponible à : https://nicolas-cousin.com/2012/07/06/test-fonctionnel-vs-test-structurel/ [Accédé le 23 Apr. 2018].

[42] Bhasin, J. (2015). *Cross-platform test automation using Appium*. [ebook] p.24. Disponible à : https://www.slideshare.net/thinkexist/cross-platform-test-automation-using-appium [Accédé le 10 Jun. 2018].