

L'ENFER QUOTIDIEN DU TESTEUR FONCTIONNEL

Invarture
Beyond your ERP

LIVRE BLANC





SOMMAIRE

Sommaire	1
L'enfer quotidien du testeur fonctionnel	2
Le concepteur/développeur : ses attributs et ses problèmes	3
1. Le dilemme des compétences croisées	3
2. A la recherche des critères parfaits	3
S'adapter au multiples environnements et comportements	6
1. Savoir appréhender le problème des environnements multiples	6
2. Echec ou pas echec, telle est la question	7
3. Le problème des «flaky tests»	7
Vers la voie du coordinateur/analyste	9
1. L'homme et la machine	9
2. Simple ne veut pas dire simpliste	9
3. Une accélération du processus selon trois axes	10
4. La 4ème révolution industrielle sera « intelligente »	11
A propos	12



L'enfer quotidien du testeur fonctionnel

Cet article va aborder le domaine de la validation des logiciels sur des aspects fonctionnels. Nous ne parlerons dans ce cas que de [tests fonctionnels d'IHM](#) et non pas de tests fonctionnels (de type Web Services par exemple) car ils se focalisent sur ce que voit un expert métier du domaine du logiciel et qui est censé le valider sur des aspects fonctionnels/métier.

Depuis quelques années un nouveau métier est apparu dans l'industrie informatique : celui que nous qualifierons de [concepteur/développeur](#) de tests fonctionnels d'IHM automatisés. Ce métier devient de plus en plus incontournable dans une chaîne de production de logiciels sérieuse et tend même à devenir irremplaçable.

Dans les parties suivantes nous n'allons pas plus nous étendre sur la partie fonctionnelle du métier : en effet la rédaction et la création de scénarios fonctionnels sont en effet très spécifiques à chaque projet et varient selon les objectifs à atteindre.

Nous allons donc nous focaliser sur les difficultés techniques que va devoir surmonter notre concepteur/développeur pour arriver à ses fins et nous détaillerons les principes et méthodes à mettre en place pour y remédier. Les principaux exemples sont basés sur des technologies Web s'appuyant sur le DOM, mais il est tout à fait possible de transposer le fonctionnement à d'autres technologies.



Le concepteur/développeur : ses attributs et ses problèmes

1. Le dilemme des compétences croisées

Le rôle de concepteur/développeur de tests fonctionnels d'IHM automatisés demande une double compétence fonctionnelle et technique : une [expertise fonctionnelle du logiciel](#) à valider pour maîtriser les diverses fonctions mais également de solides [compétences techniques](#) afin de faire fonctionner de manière stable et reproductible l'ensemble des tests dont il a la charge. La problématique actuelle est la suivante : cette double compétence rare et chère à acquérir est une condition sine qua non au bon développement des tests automatisés fonctionnels.

S'il est évident qu'avoir des compétences fonctionnelles sur le logiciel à tester est incontournable pour assurer l'exhaustivité des tests fonctionnels, posséder en plus de cela des compétences en développement se révèle être bien moins classique. Malgré cet état des lieux, ces connaissances techniques en développement sont actuellement [systématiquement requises](#) pour pouvoir utiliser convenablement les outils disponibles sur le marché.

On retrouve actuellement cette double compétence dans plusieurs configurations :

- Lorsqu'un développeur devient expert fonctionnel et s'approprie complètement le domaine fonctionnel du logiciel
- Lorsque qu'un expert fonctionnel va se former « sur le tas » au développement informatique et à l'administration technique des environnements
- Lorsque des équipes mixtes testeur-développeur utilisant des outils comme Cucumber (cette option est d'ailleurs la plus optimale même si elle génère une charge de travail double pour les équipes).

2. A la recherche des critères parfaits

La première difficulté est la définition des critères utiles et nécessaires à la recherche d'un élément de l'interface graphique de manière unique, reproductible, fiable et performante. Ce n'est qu'une fois l'élément trouvé qu'on pourra ensuite effectuer une action sur celui-ci : action de souris, de clavier, ou de vérification. Les trois principales techniques utilisées vont être une [recherche purement graphique](#) d'un élément, une [recherche par identifiant unique](#) positionné sur tous les éléments susceptibles de recevoir une action utilisateur, ou encore une [recherche de type 'XPath'](#) basée sur une arborescence d'éléments à rechercher dans l'arborescence principale de l'application à automatiser.



La **recherche graphique** est devenue, avec l'évolution de la puissance des PC d'aujourd'hui, une solution très intéressante en termes de performance, avec des délais de recherche de zone graphique inférieur à 100 ms pour une application plein écran en 1920x1080. Pour être suffisamment « compétitive » par rapport aux autres solutions de recherche d'éléments, cette recherche demande certaines fonctionnalités telles que la définition de « pattern » de reconnaissance, de « masque » de captures graphiques voire d'une intelligence artificielle (IA) permettant des reconnaissances d'images selon plusieurs résolutions ou définition de palette de couleurs. Malgré toutes ces fonctionnalités cela reste une solution « alternative » qui n'est pas forcément très fiable pour l'automatisation de tests fonctionnels.

La **recherche par identifiant unique** se rapproche à première vue de la solution idéale car cette solution garantit un très bon niveau de reconnaissance des éléments mais également un excellent rendement en termes de performance d'exécution des scénarios de tests. Cette solution est pourtant à éviter car même si elle va permettre au développeur de tests une certaine tranquillité, ce sera un vrai cauchemar pour les équipes de développeurs pour la mise en place et la maintenance de ce système. En plus d'un moyen permettant de définir des identifiants partagés à mettre en place entre les développeurs, la maintenance et l'évolution des tests automatisés seront impactés par de nombreux allers-retours entre les développeurs de tests et les équipes de développement afin de se 'caler' en permanence sur des identifiants nouveaux ou ayant évolués.

La **recherche par arborescence** a été largement popularisée grâce notamment à l'utilisation de la technologie « XPath » qui permet des recherches complexes répondant aux besoins de la plupart des applications récentes. La technologie XPath possède néanmoins deux inconvénients majeurs :

- L'impossibilité d'utiliser un système d'**expressions régulières complexes** (autre que « contient » ou « commence par » ou « finit par ») qui permettrait de traiter un plus grand nombre de cas particuliers.
- La plupart des outils d'automatisation utilisant XPath vont se contenter de capturer une arborescence de l'application en utilisant des **heuristiques internes**. Cela permet, certes, une création quasi automatique du test mais engendre par la suite des problèmes de maintenance et d'évolutions des tests fonctionnels automatisés quand l'application évolue : ces XPath ne sont pas optimaux.



Par exemple, avec la technologie XPath, la recherche d'un élément à partir d'un parent plus ou moins «lointain » sera capturée comme suit par la plupart des outils d'automatisation :



Ce type de recherche risque d'être très sensible aux évolutions de l'application à tester car l'insertion d'une div intermédiaire mettra la recherche en échec, et donc le test aussi, suite à une évolution mineure de développement. Il faudra donc modifier la chaîne de recherche XPath comme cela pour rendre la recherche plus robuste et fiable dans le temps :



La convention de [définition des identifiants d'éléments](#) entre les équipes de développement et de tests fonctionnels automatisés tout comme la [maîtrise des techniques de recherches](#) sont donc les bases indispensables d'un bon projet de validation fonctionnelle automatisée.

Alors quelle est la bonne solution pour la recherche et la reconnaissance des éléments d'une application à automatiser ?

La réponse serait peut-être un mélange judicieux des trois solutions ci-dessus et une répartition des tâches plus intéressantes entre le développeur de tests et les équipes de développement.



S'adapter aux multiples environnements et comportements

1. Savoir appréhender le problème des environnements multiples

Les utilisateurs sont habitués à pouvoir utiliser leurs applications favorites sur tous les environnements, sur tous les navigateurs, et dans des configurations techniques différentes (tailles d'écran, résolution). L'arrivée du « [responsive design](#) » n'a d'ailleurs pas simplifié la donne. Cela a multiplié les configurations possibles et rendu quasi impossible la validation manuelle sous peine de voir exploser la charge de travail.

Pour les technologies Web, [les compétences et l'expérience](#) à acquérir pour définir les bons critères et les particularités par navigateur sont des difficultés importantes à surmonter pour un concepteur de tests fonctionnels automatisés. Les cas particuliers sont nombreux et doivent être tous adressés pour permettre d'exécuter un test fonctionnel automatisé donné sur la plupart des navigateurs du marché (Chrome, Edge, Firefox et Opera).

Par exemple, des recherches sur des éléments ayant la propriété 'text' égale à « Menu » ne fonctionnera sur certain navigateur qu'à condition de rechercher la valeur « MENU » : problème de sensibilité à la casse. De même avec des espaces qui ne seront pas forcément interprétés de la même manière par tous les navigateurs. Autre exemple, lors d'affichage de liste d'éléments dans une balise DIV, chaque navigateur aura son propre comportement de chargement des éléments, obligeant le testeur à, selon le navigateur, plus ou moins scroller dans la liste afin de rendre les éléments accessibles.

Il est donc indispensable de [centraliser](#) ces comportements différenciés dans des api utilisables par tous les développeurs de tests. Cela permet un partage des connaissances de manière transparente et efficace et évite que chaque développeur de tests ne mette en place sa propre solution.

C'est pour cela qu'un [langage d'abstraction](#) de haut niveau, permettant la définition d'action standard à exécuter sur des applications à automatiser, est indispensable pour garantir le succès de projets d'automatisation d'envergure.

L'utilisation de technologies telles que Sélénium est un des éléments permettant d'obtenir ce type de comportement, même si le niveau d'abstraction de Sélénium n'est pas suffisant. C'est la voie à suivre pour permettre des exécutions différenciées par navigateurs.

L'objectif est de donner aux développeurs de tests la possibilité de ne se concentrer que sur les actions fonctionnelles à réaliser sans le « surcharger » par des considérations techniques lourdes et pénalisantes.



2. Échec ou pas échec, telle est la question ?

Une autre difficulté quotidienne du développeur de tests automatisés sera d'analyser les résultats des tests, en particulier le « [pourquoi](#) » des tests en échec. Dans le cas d'une application web plusieurs raisons sont possibles : problème du réseau pendant l'exécution en intégration continue, problème du navigateur, problèmes sur le serveur d'applications, problème du Framework de test, et finalement le plus intéressant, problème fonctionnel, celui que l'on recherche désespérément et qui justifie tout ce travail.

Comment arriver à rapidement qualifier un test en échec ?

Plusieurs solutions sont possibles mais ne permettent pas tout le temps d'avoir la réponse juste :

- Un rapport et des logs techniques ne permettront pas à un expert fonctionnel de facilement comprendre les raisons d'un échec.
- Une capture d'écran à la fin du test est indispensable même si la plupart du temps les raisons de l'échec se sont produites 4 ou 5 actions avant la dernière action qui échoue.
- Une vidéo de l'exécution du test permet de mieux visualiser l'exécution et permet de mieux se rendre compte de ce qui s'est passé mais ne permet pas forcément de bien comprendre les actions réalisées.
- Des logs fonctionnels sont à mon avis indispensables afin de mieux comprendre les problèmes sur un test.

La solution idéale est probablement un mix de tout cela. Le plus important est sans doute qu'elle soit tirée de la [réelle exécution](#) qui a conduit à l'échec afin d'éviter d'avoir à reproduire cet échec sur un environnement dédié, ce qui peut s'avérer quelquefois difficile.

3. Le problème des « flaky tests »

Tous les concepteurs de tests fonctionnels automatisés ont été un jour confrontés à ce problème : le test ne fonctionne pas, on le relance juste une deuxième fois « pour être sûr » et là il se remet à fonctionner normalement... relativement énervant.

Il y a même des équipes de développement qui ont créé des modules sur [Jenkins](#) pour contourner ce problème, en définissant une estimation d'échec de test sur les dernières exécutions afin de ne s'occuper que des tests ayant une estimation d'échec anormale. Le comble pour un informaticien qui ne comprend normalement que 0 ou 1, il se retrouve à gérer des résultats « aléatoires » et se baser sur des suppositions pour trouver des solutions.



Les raisons d'un « flaky » test sont multiples et difficiles à prévoir. La **généralisation et la diffusion** de plus en plus importante de composants et de processus asynchrones en sont une explication sans en être la seule.

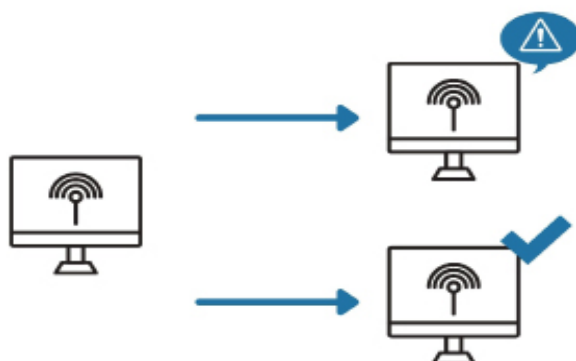
Des composants peuvent apparaître, ou être modifiés par du code côté client (navigateur) bien après le chargement d'une page html par exemple. Si ce type de comportement n'est pas pris en compte, l'exécution d'une action sur ce composant va échouer alors qu'en temps normal l'exécution du code ne prend que quelques millisecondes. Les raisons de ce temps d'attente supplémentaire sont multiples : réseaux, serveur d'application un peu surchargé, le navigateur qui sollicite un peu trop la mémoire du système d'exploitation... En d'autres termes **impossibles à anticiper**.

La seule solution fiable est de mettre en place un système d'exécution d'actions à **tentatives multiples**. Si une action « clic » échoue car l'élément n'est pas apparu ou parce que sa propriété « enabled » est à « false », la réexécution de cette même action à peine 100 ms plus tard fonctionnera parfaitement.

Cela justifie-t-il d'ajouter systématiquement 500 ms entre chaque action pour être tranquille ? Evidemment que non car cela augmente artificiellement le temps d'exécution global des tests mais ne nous met pas plus à l'abri d'un élément qui ne sera finalement cliquable qu'au bout de 501 ms.

La correction des « flaky tests », malgré les apparences d'un problème marginal, est bien un des **problèmes majeurs** du concepteur de tests, car non seulement ce dernier perd du temps à relancer les tests mais il risque de mettre en place un sentiment de défiance des développeurs d'application envers les tests automatisés, voire envers le concepteur des tests.

Ainsi le premier réflexe des développeurs d'applications ne sera donc pas de vérifier si les tests automatisés ont bien détecté une défaillance applicative, mais malheureusement de dire que les tests automatisés ne sont pas fiables et finalement demander au concepteur de tests de revérifier son travail... Pour au final beaucoup de temps et d'énergie perdus.





Vers la voie du coordinateur/analyste

1. L'homme et la machine

Ne jamais laisser faire le travail d'un programme par un humain - Agent SMITH, Matrix

Pour tester au mieux une application et ses IHM (Interfaces Homme Machine), même si aucun humain ne rivalisera jamais avec un programme en termes de fiabilité et de performance d'exécution, il est impératif de **simuler au maximum** les actions telles que le ferait un humain sur une application, et pas telle que le ferait un programme, qui sera efficace mais qui ne permettra pas forcément en évidence des problèmes liés à une utilisation « normale » de l'interface.

Avec l'évolution des interfaces de plus en plus **graphiques**, complexes et « riches » il sera important d'exécuter une séquence d'action permettant de solliciter de manière réaliste l'interface, par exemple en réalisant systématiquement une action « hover » avant de faire une action clic sur un élément, ou en entrant du texte par des actions clavier et pas par une affectation directe de valeur, un « swipe » sur un élément « slider » ou un clic sur une « combo » avant de sélectionner un élément de la liste.

Encore une fois des **api spécialisées** et faciles d'utilisation sont indispensables pour gérer ce type de comportement. Dans le cas du « swipe » le concepteur de test ne doit ni gérer la taille des composants ni la position de la souris mais doit « juste » par exemple créer une action « swipe » vers la gauche ou vers la droite.

2. Simple ne veut pas dire simpliste

La plupart des logiciels ou systèmes de test que j'ai été amené à utiliser ou à évaluer ont presque tous les mêmes caractéristiques : une grande **complexité dans l'organisation** des fichiers, des composants ou des outils nécessaires à l'utilisation de leur Framework de test.

Pour certains, un seul script de test va générer 3 autres fichiers sources, pour d'autres une structure de fichier Xml ou Json tellement complexe qu'il est impossible de les lire facilement. Il y en a même qui génèrent des fichiers au format binaire.



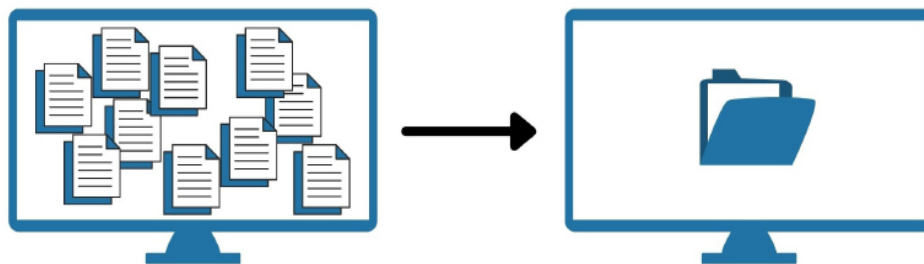
D'autres solutions vont présenter des interfaces avec une présentation très graphique des actions à réaliser avec tellement de choix et de paramètres possibles qu'après avoir créé 10 actions il devient vite difficile de comprendre et d'appréhender le script qu'on a créé.

Peut-être faut-il retourner à des fondamentaux **simples** qui ont prouvé leur efficacité :

un script de test = un script d'exécution généré.

Pour permettre l'utilisation d'**outils collaboratifs** comme Svn, Git ou autres, un script doit être facilement « lisible » afin de permettre une meilleure compréhension des modifications des camarades (la vérité est dans le code).

Dernier point mais pas des moindres, proposer un système simple et logique d'organisation par sousscripts dans des répertoires définis : cela peut paraître évident dit comme ça mais ce n'est pas systématiquement le cas actuellement.



3. Une accélération du processus selon trois axes

L'ensemble des problèmes remontés ci-dessus s'intègre dans un processus qui est en train de fortement s'accélérer. En effet, la mise en place des **méthodes Agiles** impose aux équipes de validation de pouvoir valider une application à chaque sprints, à savoir en une ou deux semaines suivant les méthodologies en place. La nécessité d'optimiser **la communication et la collaboration** entre les intervenants des projets s'en trouvent d'autant plus importante.

La mise en œuvre de plus en plus fréquente des solutions logicielles dans le cloud impose de mettre en place des processus de modifications/corrections Dev/Ops complètement **automatisées** : nous ne pouvons plus nous permettre d'intégrer des éléments de validation manuelle dans le cycle de déploiement lorsque potentiellement tous nos clients peuvent être impactés par un problème.



La multiplicité des environnements va aussi conduire les concepteurs d'outil d'automatisation à développer des solutions permettant de rejouer des tests sur plusieurs environnements **simultanément**, se limiter aux technologies Web ou Mobiles n'est plus une option.

Sous peine d'explosion, l'industrie de la validation automatisée va devoir **se réinventer** pour permettre les accélérations exigées par les clients tout en trouvant des solutions pragmatiques et simples aux problèmes auxquels elle fait face et qui exigent une charge de travail en validation toujours plus importante.

4. La 4ème révolution industrielle sera « intelligente »

Nous pensons que le métier traditionnel du test logiciel est sur le point d'**évoluer fondamentalement**. Si nous avons abordé ici les problématiques techniques auxquelles sont soumis les intervenants en charge des tests automatisés fonctionnels tout comme les problématiques humaines de communication et de travail collaboratif, nous prévoyons que l'avènement de l'**IA** va ouvrir de nouvelles perspectives majeures.

Le rôle de concepteur/développeur de tests automatisés fonctionnels va être transformé par la mise en application des fonctions avancées ouvertes par le « Machine Learning » et finalement tendre plus vers un rôle de **coordinateur/analyste**.

C'est dans cette optique que la solution **Agilitest** a été pensée et conçue, autant pour répondre aux problématiques que posent le métier de développeur de tests automatisés fonctionnels que pour préparer le terrain à la mise en application de l'intelligence artificielle.

Dans l'inconscient collectif, l'identification d'un composant dans une page ne devrait se faire qu'une seule fois pour que la machine s'en souvienne, l'analyse des « flaky tests » devrait être automatique, et au-delà du « monkey-testing » qui est actuellement pratiqué, la machine devrait même être en mesure de générer des tests de plus en plus intelligents pour laisser les humains prendre plus de recul sur ce qui compte vraiment : comment permettre à leurs clients de mieux exercer leur métier en utilisant la solution testée.

Nous pensons que le succès de l'intelligence artificielle sera de permettre d'alléger les phases de tests en vue de consacrer en amont plus de temps aux phases à forte valeur ajoutée comme le sont par exemple l'écoute du besoin ou encore la conception du logiciel.



A PROPOS

L'auteur



Pierre Huber, 50 ans et 30 ans d'expérience en développement logiciel, 12 ans d'expérience en tests fonctionnels automatisés et développement d'outils et plateforme de tests. Pierre est fondateur de CAIPTURE S.A.S, l'éditeur d'Agilitest, et Directeur Recherche et Développement. Depuis 4 ans maintenant, Pierre et son équipe travaillent sur l'intégration et le développement d'Agilitest.

En savoir plus sur Agilitest

Agilitest est une suite intuitive d'automatisation des tests fonctionnels conçue pour que tout le monde puisse automatiser les tests. Ce programme s'oriente vers une conviction que l'innovation doit être au service d'une expérience utilisateur simple et intuitive.

Agilitest est une solution proposée par Invarture, société française spécialisée dans la fourniture de logiciels dédiés à l'optimisation de SAP. Invarture et Agilitest sont devenus partenaires en 2021 pour porter la solution d'automatisation des tests fonctionnels Agilitest sur les environnements SAP.

Rendez-vous sur notre site internet pour plus d'informations sur la solution Agilitest :

www.invarture.com

Pour nous contacter :
info@invarture.com

Invarture
Beyond your ERP