Stage de fin d'études

Conception d'une infrastructure de test automatisé d'applications web

Auteur	**Matti Schneider** (hi@mattischneider.fr)
Entreprise	**Eurogiciel Ingénierie** — Agence de Sophia-Antipolis (06), France
Encadrants	**Fabien Massol** **Laurent Basset** (laurent.basset@eurogiciel.fr)
Établissement	**École d'ingénieurs Polytech'Nice-Sophia** Parcours Ingénieur, filière KIS (Knowledge and Information Systems)
Tuteur	**Peter Sander** (sander@polytech.unice.fr)
Volume	48 pages dont 28 de contenu. Durée de lecture estimée : environ 50 min.

Remerciements

Merci à Fabien Massol pour deux mois très enthousiasmants, pour avoir insufflé l'énergie nécessaire au démarrage de ce projet, et pour avoir tout fait pour que mon stage soit excellent.

Merci à Laurent Basset et à Muriel Janel d'avoir pris de leur temps pour me présenter leur poste.

Merci à Caroline Desmoulins pour sa bonne humeur et son accompagnement au quotidien.

Merci à Hillel Nabedrick pour les pauses, et à Yann Tassy pour ses retours d'expérience.

Merci à Christophe Desclaux pour tous ces joyeux midis.

Merci à tous ceux qui ont fait et font le mouvement open-source, sans lesquels ce projet n'aurait jamais pu être ce qu'il est devenu.

Table des matières

Tout au long de ce rapport, les mots définis dans le glossaire sont indiqués par un astérisque (*) lors de leur première apparition.
La présence de nombres entre crochets en exposant ([1]) indique une référence bibliographique.

Contexte

Établissement d'enseignement

Polytech'Nice-Sophia est une école publique d'ingénieurs rattachée à l'Université de Nice-Sophia-Antipolis. Le stage décrit dans ce document a lieu à temps plein pendant le dernier semestre d'études, validant ainsi une formation de cinq ans dont trois en cycle ingénieur.

Entreprise

Eurogiciel se définit comme une SSAP, Société de Services en Accompagnement de Projet.

Bien que société de services, c'est-à-dire ayant pour activité principale la prise en charge de projets logiciels externalisés, ses valeurs la rapprochent plus de la firme de consultants que de la SSII*. En privilégiant le placement de collaborateurs à long terme et la maîtrise des processus de gestion, Eurogiciel vise une haute qualité logicielle.

Sujet

Au vu de l'importance prise par les applications web* dans l'industrie, dans le cadre d'une volonté d'internalisation de projets, et en accord avec ses valeurs, Eurogiciel souhaite développer une offre complète de gestion et d'amélioration de la qualité logicielle de telles applications.

Plus spécifiquement, il s'agit de :

- étudier les outils de test automatisé existants ;
- développer une structure logicielle corrigeant les limitations rencontrées ;
- concevoir une offre de vente de l'expertise correspondante.

L'objectif est d'accompagner les clients souhaitant améliorer leurs pratiques de gestion de la qualité de leurs applications web.

Que contient ce document ?

Ce document est le **rapport final** rédigé au terme du stage. Il contient :

- l'examen en profondeur de l'espace du problème ;
- un état de l'art aussi exhaustif que possible des solutions existantes ;
- une description du travail réalisé pour répondre aux problèmes non résolus aujourd'hui ;
- ses limitations et les méthodes employées pour le mener à bien.

En un clin d'œil

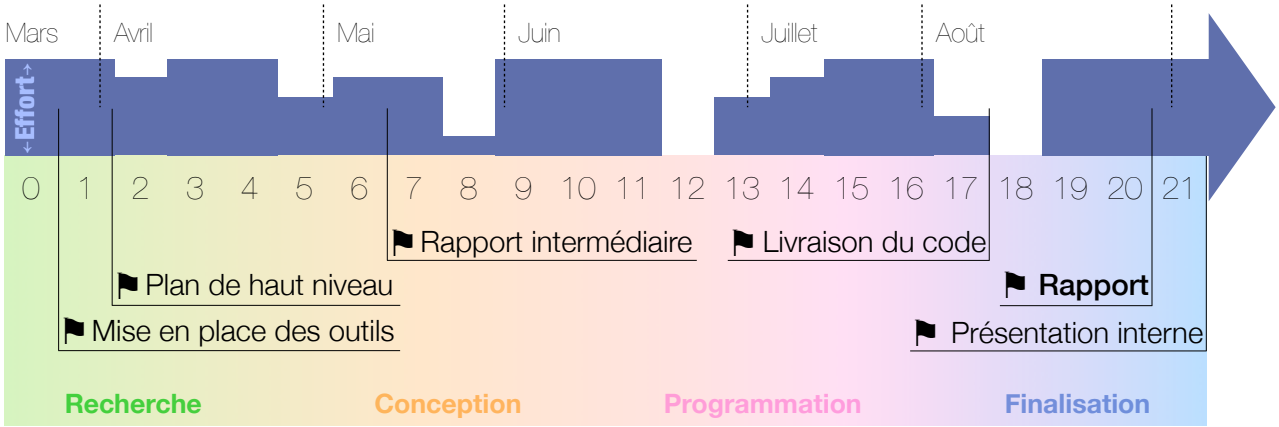
Cette section est une version simplifiée du plan de haut niveau prévisionnel disponible en annexe.

Objectifs

- Réduire les coûts associés à la gestion de la qualité.
- Évaluer la qualité d'un site ou d'une application web.
- Évaluer les risques de régression.

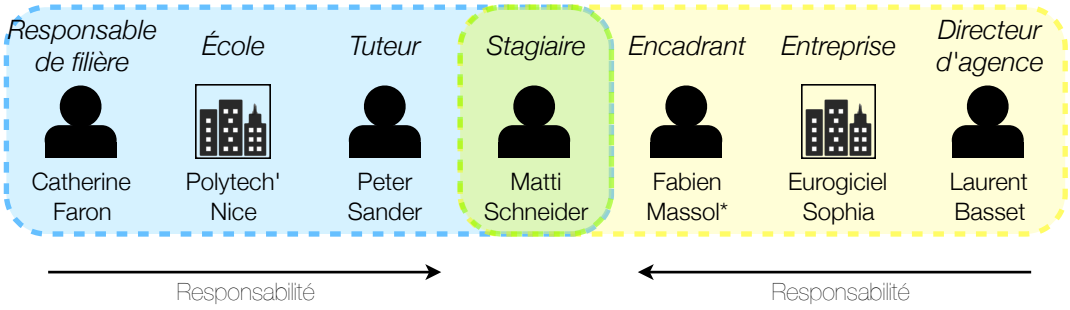
Feuille de route

Nous travaillons avec la méthodologie agile* Scrum, sur des itérations d'une semaine.



Parties prenantes

Internes



* Fabien Massol a quitté Eurogiciel fin juin. Mon encadrement est depuis assuré par Laurent Basset.

Externes

- Clients d'Eurogiciel.
- Partenaires et concurrents.
- Ingénieurs d'Eurogiciel.

Problématique

Tests automatisés

L'ingénierie logicielle est un domaine complexe, dans lequel les erreurs ne sont pas rares. En tant que telles, ces anomalies font partie du processus créatif de raffinement et de découverte de solutions. En revanche, il convient d'en minimiser le nombre passant la mise en production.

Une solution efficace et communément utilisée est l'usage de tests automatisés, c'est-à-dire du code exécutable exerçant le code de l'application livrée et validant les résultats renvoyés à partir d'une base des résultats attendus. Toute différence entre la valeur concrète et l'étalon est considérée comme un échec, et bloque la mise en production jusqu'à sa résolution.

Nous nous intéresserons ici à la validation de fonctionnalité directe, et non aux contrôles annexes tels que résistance à la charge, au volume ou encore aux attaques. Ces éléments, bien que tout aussi cruciaux à la qualité finale de l'application et dont la validation peut être en partie automatisée, sont transversaux et sortent de la hiérarchie que nous allons présenter.

Niveaux de test

Tout comme un programme peut être décomposé en différents domaines (l'exemple classique est modèle de donnée / code métier / interface utilisateur), le code de test peut se restreindre à un sous-ensemble de ces domaines, ou encore à une approche spécifique.

Il est important de comprendre les différences entre ces types de validation, ou *niveaux*, car les enjeux qui leur sont associés varient fortement.

Tests unitaires

Visant le niveau le plus proche du code, cette approche consiste à tester séparément les *unités* élémentaires du code, à savoir des fonctions et méthodes individuelles.

Un jeu de tests unitaires permet de déterminer très précisément l'origine d'une erreur dans l'application, tout en restant dans un environnement d'exécution très contrôlé puisque l'exercice n'est qu'appels internes. Il s'agit donc d'un système de validation puissant lors du développement, mais qui ne peut confirmer le fonctionnement de l'application complète.

Tests fonctionnels

Agissant à un niveau intermédiaire, les tests fonctionnels sanctionnent un bloc de fonctionnalités, soit généralement un module d'un point de vue de l'architecture logicielle.

Ces tests sont plus abstraits par rapport à l'organisation interne du code, bien que leurs points d'entrée soient encore des éléments de programmation : ils ne connaissent que les entrées et sorties des modules. Ce n'est cependant toujours pas ce avec quoi l'utilisateur final interagira, à moins que le livrable ne soit destiné à des développeurs.

Tests d'intégration

Les tests d'intégration, ou tests d'acceptation utilisateur, sont les seuls s'étendant à la totalité du programme, dans son environnement final, et avec les moyens d'interaction de l'utilisateur final. Pour cette raison, ils sont parfois également nommés *end-to-end testing (bout à bout)*.

De tels tests peuvent sembler les plus importants. En effet, ce sont les seuls qui valident à proprement parler les fonctionnalités que le client souhaite fournir, indépendamment de l'implémentation sous-jacente. Cependant, leur coût est rapidement prohibitif, comme nous allons le voir.

Coûts des tests d'intégration

Réalisation manuelle

Aujourd'hui, seule une minorité des tests d'intégration effectués au sein de l'industrie logicielle est automatisée. Cette pratique contraste fortement avec les autres niveaux. Il paraîtrait inimaginable, par exemple, d'effectuer des tests unitaires "à la main" en dehors des toutes premières phases d'expérimentation.

Cette absence d'industrialisation a un coût très élevé. Bien évidemment, trivialement, par le temps passé par les chargés de qualité à effectuer les tests, qu'ils soient développeurs ou non (dans le cas d'un département *QA** distinct).

Mais également, moins directement, la durée de ces tests rallonge, potentiellement fortement, le délai entre réalisation et livraison. De fait, dans le cas d'une anomalie, sa découverte apparaît tard dans le processus, et son impact peut s'en trouver fortement augmenté[1].

Enfin, en poussant souvent à un découplage entre responsables du développement et de la qualité pour des raisons économiques[1], les deux activités se retrouvent mises en concurrence, diminuant de fait la collaboration entre les personnes et donc la productivité globale de la structure.

Industrialisation

La solution appliquée aux autres niveaux de validation, à savoir l'industrialisation du processus de validation, paraît viable et efficace. Elle consiste en :

- l'automatisation des tests par l'écriture de code spécifique par les développeurs ;
- la mise en place de moyens d'exécuter ces tests automatisés de manière fiable et reproductible ;
- l'incorporation de cette exécution dans le processus de développement habituel, par exemple au moment de l'intégration de nouveau code dans le livrable.

Bien évidemment, si cette industrialisation est aujourd'hui peu répandue, c'est qu'elle pose des difficultés importantes. Les tests d'acceptation sont en effet plus complexes à mettre en œuvre que les systèmes appelant directement le code du programme, et ce pour chacun des points

[1] Le poste de testeur est généralement moins rémunéré qu'un poste de développeur. Les activités de validation peuvent aussi être totalement délocalisées, voire externalisées.

donnés ci-dessus. Il paraît néanmoins important de souligner que cette complexité est avant tout due à un manque d'outils et de pratiques adaptés.

Origine possible du retard

Je ne prétendrai pas expliquer la totalité des difficultés rencontrées, mais tenterai de présenter plus précisément leur contexte et la raison du décalage observé avec les autres niveaux de test. Il s'agit avant tout d'un postulat, non étayé par des recherches exhaustives. L'hypothèse paraît cependant, tant par l'absence de conséquences pratiques sur le sujet étudié que par la similarité avec d'autres problèmes récurrents dans l'industrie logicielle (standards, bibliothèques d'abstraction graphique…), suffisamment crédible pour être avancée.

L'utilisation d'interfaces homme-machine évoluées, et la conscience de l'importance de cette partie dans le succès ou non d'un logiciel, par opposition à la prédominance absolue précédemment accordée aux parties métier, paraît relativement récente dans l'industrie informatique (début des années 90). Cette préoccupation s'est vue fortement appuyée par la montée en puissance des applications web (milieu des années 2000), où l'utilisateur final n'a plus à gérer les questions de compatibilité et d'installation.

L'importance prise par la validation de l'interface utilisateur final est donc très récente par rapport aux autres niveaux de test, rencontrés dès les débuts de l'informatique pour les tests unitaires, ou dès la modularisation en composants réutilisables pour les tests fonctionnels.

Ce placement historique pourrait expliquer, au moins en partie, l'absence des tests d'intégration de la plupart des cursus d'enseignement et le manque d'outils et pratiques adaptés, comme nous allons l'explorer. Il y a en effet aujourd'hui moins d'outils libres*, simples et fiables sur le marché pour automatiser efficacement les interfaces utilisateur qu'il n'y en a pour les autres niveaux de test, où de clairs standards sont établis, tels que le jeu de bibliothèques xUnit.

Environnement d'exécution

La fragilité principale des outils d'automatisation d'interfaces utilisateur est leur sensibilité à l'environnement. L'entrée utilisateur simulée n'est en effet pas isolée des actions simultanées d'un utilisateur réel, et la machine utilisée pour les tests doit donc leur être intégralement dédiée si l'on veut que les tests soient fiables.

Dans le contexte plus spécifique des applications web, la question de l'environnement s'étend également à deux paramètres techniques. D'une part, la vitesse et la qualité de l'accès au réseau peuvent fortement impacter le déroulement des tests, et les actions automatiques doivent donc prendre en compte la dimension temporelle dans le chargement des différents éléments. D'autre part, le choix des navigateurs employés pour le rendu des pages web en modifie l'aspect et les fonctionnalités. Qui plus est, tous ne sont pas égaux en termes de capacités d'automatisation.

Intégration dans le processus de développement

De par les difficultés énoncées précédemment, à savoir tant la disponibilité que la qualité de solutions techniques, il n'existe aujourd'hui pas de solution “clés en main” pour l'automatisation de validation d'interface utilisateur, a contrario des autres niveaux de tests.

De plus, la nécessité de ne pas interférer avec l'ordinateur à travers les dispositifs d'entrée classiques pendant la durée de tests d'interface pose un problème matériel. Soit le développeur vérifiant ses modifications doit être immobilisé le temps des épreuves, dont la durée est loin d'être négligeable (généralement un minimum de 10 secondes par test) ; soit des serveurs dédiés à la validation asynchrone doivent être intégrés à l'infrastructure.

Dans tous les cas, la question de l'infrastructure s'impose encore plus dans le cas d'une application web, où l'on retrouve le problème des différents navigateurs… et où tous ne sont pas disponibles sur tous les systèmes d'exploitation. L'intégration de tels tests dans un processus de vérification multi-plateformes nécessite donc une infrastructure relativement complexe, que ce soit sur la base de machines physiques ou virtuelles, pour distribuer correctement la validation et en compiler les résultats.

Un examen approfondi des différentes bibliothèques existantes, de leurs différences et avantages respectifs, ainsi que des limitations qui leur sont aujourd'hui communes a constitué la première partie du travail décrit par ce document, que nous allons aborder immédiatement.

Solutions actuelles

Recherche

Il existe à première vue dans le domaine de l'automatisation de tests d'intégration d'applications web, comme dans tout autre domaine logiciel, une myriade de solutions[9]. Et, comme dans tout autre domaine logiciel, la majorité n'est plus maintenue, ou obsolète, ou non documentée…

J'ai effectué de nombreuses recherches sur l'offre existante, et exploré plus particulièrement environ 80 outils, sur lesquels 47 m'ont semblé potentiellement envisageables, à divers degrés de confiance. Je les ai ensuite catégorisés et classés selon différents critères tels que taux d'adoption, qualité de la documentation, date de dernière mise à jour…

Après un examen plus poussé et une sélection basée avant tout sur le taux d'adoption et de soutien dans la communauté, sept outils ont fait l'objet d'une évaluation complète.

Évaluation

Trois applications ou sites ont été choisis, et un ensemble de scénarios représentatifs de situations de test récurrentes leur ont été associés. Chacun de ces scénarios contenait un "piège" classique pour les validations automatisées, afin d'évaluer leur résistance. Le choix des sites visait à couvrir le champ technologique le plus large, afin d'être aussi représentatif que possible, tout en restant suffisamment peu nombreux pour ne pas complexifier outre mesure l'évaluation.

Sites ciblés

checkmytrip.com
CheckMyTrip est une application web fournie par Amadeus, un des clients d'Eurogiciel.

D'un point de vue technologique, CheckMyTrip est intégralement basé sur AJAX*, c'est-à-dire que les pages transmises ne contiennent aucune information ; celles-ci seront chargées quelques temps plus tard, de manière dynamique. Cette spécificité permet d'évaluer la capacité du système de test à gérer des délais variables selon la qualité de la connexion réseau.

pdc.refedd.org
L'outil Plans de Déplacements Campus est une application web fournie par le REFEDD et développée par l'auteur de ce document[10]. Cette application n'a pas de lien avec Eurogiciel, mais présente l'avantage important de pouvoir être examinée de l'intérieur, potentiellement modifiée, et de permettre l'évaluation de la facilité à transposer des tests locaux sur une application déployée.

PDC repose sur les technologies HTML5, sans être aussi dépendant d'AJAX que CheckMyTrip.

gsf-fr.net
Le site web de GSF, prospect d'Eurogiciel, est un simple portail d'informations basé sur Flex*, une technologie un peu plus exotique et posant des problèmes techniques spécifiques.

Outils classiques évalués

La dénomination d'outil “classique” s'applique aux bibliothèques bien connues, utilisées par une partie de l'industrie, maintenues et possédant une certaine communauté.

- Selenium IDE ;
- Watij ;
- Cucumber ;
- Mocha.

Outils innovants évalués

L'adjectif “innovant” désigne des bibliothèques fournissant une approche en rupture avec les outils “classiques”. Aujourd'hui, il s'agit majoritairement de preuves de concept, fonctionnelles mais n'ayant pas nécessairement une forte communauté associée. Un minimum de reconnaissance publique a néanmoins été une condition nécessaire à l'évaluation, afin d'éliminer les bibliothèques personnelles non génériques.

- Madcow ;
- Test::Right.

Conclusions

Chacun des outils ci-dessus a été utilisé pour implémenter des tests d'intégration sur les trois sites listés plus haut, selon une procédure fournie en annexe. Les conclusions ci-dessous se basent sur la rédaction de ces tests et le résultat de leur exécution.

Visibilité

Le domaine de la validation d'applications web, comme on l'a vu, souffre probablement de sa jeunesse quant aux taux d'utilisation des outils.

Néanmoins, un autre problème important s'est fait jour lors de la recherche des outils en question : il est aujourd'hui très difficile de se repérer dans l'offre existante. Il n'existe aucun comparatif facilement accessible des outils, ni de bonne pratique satisfaisante. À vrai dire, le consensus semble plutôt être sur l'impossibilité de tester efficacement l'interface de sites web[3].

En conséquence, les communautés d'utilisateurs des différents outils sont souvent fragmentées, et ne connaissent pas nécessairement les interdépendances entre les logiciels utilisés.

Outils

Le résultat a le mérite d'être clair : **aucun des outils évalués n'a aujourd'hui la fiabilité et l'efficacité nécessaires à l'industrialisation**, du moins sans méthodes additionnelles[2,11,12,13].

Outils classiques

Tous les outils “classiques” sont confrontés aux limitations énumérées dans la première partie de ce document, notamment le manque d'isolation envers l'environnement.

Mais le problème principal est celui de la maintenabilité. Écrire un test fonctionnel avec l'un de ces outils est tout à fait possible ; mais la première modification du système testé (SUT*) l'invalidera facilement[13], et le coût d'adaptation du test en question sera particulièrement élevé.

En effet, toutes les bibliothèques fournies par les outils “classiques” permettent de commander des navigateurs et d'évaluer des assertions sur les résultats renvoyés, mais la forme des tests est impérative : toutes les instructions sont listées en séquence, sans considération pour les bonnes pratiques d'ingénierie logicielle recommandant d’éviter les répétitions ou encore de décomposer le code en blocs atomiques[12].

En réalité, on se rend compte après un tel examen que la fragilité dénoncée par les ténors du développement et de la qualité provient certes de la sensibilité à l'environnement extérieur, mais également, pour une part non négligeable, de l'absence de fondation architecturale solide[11], un problème reconnu et résolu dans beaucoup d'autres domaines. C'est d’ailleurs sur ce constat que s'appuie la rupture proposée par les outils “innovants” évalués.

Outils innovants

Tant les outils “classiques” souffrent de leur lourdeur, tant les outils “innovants” souffrent de la fragilité de leur propre code. Jeunes, peu revus, rarement utilisés, souvent présentés comme libres mais en réalité développés en interne par une entreprise n'ayant pas nécessairement pris la mesure de l'ouverture au public, leur utilisation nécessite un important travail d'exploration. Les exemples donnés semblent alléchants, mais la rédaction de tests fonctionnels sans autre documentation relève plus du tour de force que de la partie de plaisir.

Néanmoins, bien que leur utilisation en l'état soit trop peu productive, les idées présentées ont un potentiel remarquable. La proposition architecturale de *Test::Right*, rappelant fortement les meilleures pratiques du développement web, paraît particulièrement fiable. À chaque test, la majeure partie du temps fut perdue à relire le code interne non commenté pour compenser l'absence de documentation ; mais une fois les appels nécessaires élucidés, l'écriture du test en elle-même est d'une simplicité tranchant net avec les autres solutions, résultant en une quantité de code plus faible, une lisibilité largement meilleure, et une fiabilité accrue.

Solutions envisagées

Après cette évaluation, trois approches apparaissent pour répondre aux besoins de ce stage :

1. Utiliser des outils fortement soutenus, reconnaître leurs limitations et définir une série de bonnes pratiques permettant de diminuer les risques et d'éviter les tests trop fragiles.

2. Améliorer l'un des outils innovants en le documentant par rétro-ingénierie, en lui ajoutant des jeux de tests, et en en améliorant le code pour l’industrialiser.

3. Créer un nouvel outil inspiré des philosophies des outils innovants, en le pensant dès sa conception pour l'industrialisation et l'ouverture au public.

Solution choisie

Après présentation des résultats de mon analyse et évaluation interne, le consensus s'est fait sur la conception d'une nouvelle solution. J'ai donc formulé une proposition prenant en compte les trois aspects actuellement problématiques.

Automatisation

Contrôle des navigateurs

Les actions sur les navigateurs seront commandées par Selenium WebDriver, bibliothèque commune à la majorité des structures de test examinées.

L'avantage principal de ce choix est le nombre de plateformes supportées. Les alternatives comme WatiX supportent moins de navigateurs, et d'autres en utilisent un faux n'implémentant qu'un sous-ensemble des opérations accessibles sur un véritable navigateur.

Modélisation des tests

C'est ici que le travail le plus important sera effectué, en tâchant de trouver la manière la plus appropriée, maintenable et efficace d'écrire des tests. L'inspiration sera prise avant tout dans les outils "innovants", en particulier *Test::Right* ; mais également dans les patrons de conception* documentés[2] par la communauté d'utilisateurs de Selenium.

L'idée principale qui ressort de la lecture de ces sources est celle de l'abstraction. Solution connue et reconnue à la majorité des problèmes de maintenabilité rencontrés par l'industrie logicielle, abstraire le plus possible les éléments modélisés permet de fournir des solutions les plus génériques possibles, et d'orienter les implémentations spécifiques vers un langage beaucoup plus déclaratif qu'impératif. La maintenance s'en trouve très largement facilitée, car la logique de validation est découplée de la déclaration des données étalon et des éléments à valider.

Il s'agira donc de trouver, de manière abstraite, la manière la plus simple de représenter et de définir un jeu de données de tests.

Environnement d'exécution

Pour diminuer la sensibilité à l'environnement tout en conservant l'aspect multi-plateformes indispensable à une application web, je propose de distinguer deux modes de test.

Validation déportée

Il s'agit ici d'utiliser la technique habituelle de contrôle des navigateurs, sensible à l'environnement. Néanmoins, elle sera déportée et exécutée uniquement sur un serveur d'intégration. Ainsi, les machines dédiées au développement ne seront pas touchées par la durée de la validation.

Validation sur machine de développement

Pour permettre aux développeurs de ne pas avoir à systématiquement propager leur code et attendre le bon vouloir du serveur d'intégration avant de pouvoir valider, l'utilisation d'un navigateur en mode *headless*, c'est-à-dire sans interface graphique, semble attirante.

Jusqu'ici, la majorité des navigateurs de ce type ont été écrits spécifiquement pour ce mode de rendu, et n'étaient donc pas nécessairement représentatifs du rendu de navigateurs habituels, remettant dès lors en question le bien-fondé de ces tests.

Je prévois ici d'utiliser PhantomJS, une implémentation *headless* de WebKit, moteur de rendu très répandu (utilisé notamment par les navigateurs Chrome et Safari, en versions fixe et mobile).

PhantomJS est contrôlable directement par JavaScript ; il paraît cependant plus intéressant de le rendre compatible avec Selenium WebDriver que d'utiliser une API* spécifique, pour les raisons de large adoption soulignées dans la partie précédente. Cela reviendrait donc à rejoindre le développement de GhostDriver, projet dédié à l'abstraction de PhantomJS par WebDriver.

Usage

Pour que la validation soit intégrée dans le processus de développement quotidien, il faudra assurer une facilité d'exécution des tests dans les deux environnements décrits plus haut. La validation devra pouvoir être exécutée en une commande.

Idéalement, sur machine de développement, tous les tests seront lancés avant propagation des modifications. De même, aucune intégration ne devra être faite sans validation multi-plateforme par le serveur dédié.

Si cette simplicité d'utilisation devient réalité, l'utilisation d'un serveur d'intégration continue* offrira un filet de sécurité suffisamment crédible pour protéger réellement des régressions.

Technologies

Au vu de l'évolution technologique dans le domaine ciblé, à savoir les applications web, le seul langage commun à tous les environnements de développement est celui utilisé pour la programmation côté client : JavaScript. Ce langage paraît donc indiqué pour automatiser des navigateurs dont il représente la seule capacité d'exécution. Ainsi, les développeurs habitués au web connaîtront forcément le langage utilisé. Qui plus est, possédant personnellement une forte expertise dans cette technologie, il paraît bienvenu d'utiliser cet atout.

Réalisation

Environnement

Une fois le langage de programmation fixé, encore faut-il trouver un environnement d'exécution permettant de transformer le code source en actions.

L'environnement d'exécution classique du JavaScript est le moteur de rendu des navigateurs web. Cependant, étant ici les systèmes testés, il paraissait absurde et risqué d'utiliser un tel environnement à la fois en tant que pilote et cible.

Node.js

Heureusement, un environnement d'exécution jeune mais déjà très largement soutenu affranchit JavaScript du navigateur web et permet de l'exécuter comme tout autre langage de script, directement à la ligne de commande. Cet environnement, nommé Node.js (on y fera par la suite référence sous le nom "*Node*"), ajoute aussi le support du système du fichiers, ainsi qu'un gestionnaire de paquets et de modules.

Node.js étant multi-plateformes, et JavaScript étant interprété, c'est-à-dire indépendant de la machine qui exécute le code, la question de la compatibilité entre systèmes d'exploitation cibles est très largement résolue. Cela résout également la question du découplage entre validation déportée et locale : il suffit que le serveur d'intégration utilisé pour la validation déportée (voir chapitre précédent) soit équipé de Node.js pour pouvoir exécuter un test.

Modules

Gestionnaire de paquets

Il faut ici que le lecteur peu habitué au développement logiciel comprenne le fonctionnement d'un gestionnaire de paquets.

Les logiciels sont généralement installés par un utilisateur final sous la forme d'applications téléchargées depuis internet, ou d'installateurs lancés depuis un support physique (CD-ROM, clé USB…). Néanmoins, cela devient rapidement limitant pour un développeur.

En effet, aujourd'hui, la grande majorité des modules (de petits programmes logiciels effectuant une tâche spécifique et couramment nécessaire à la réalisation de plus grands programmes) est publiée sous licence libre, c'est-à-dire qu'ils sont librement réutilisables par tous les autres développeurs. En conséquence de quoi, le développement d'un logiciel consiste en très grande partie en la recherche et l'intégration de modules déjà existants, et en la rédaction de code assurant le lien entre ces différents blocs.

Vu le nombre de modules nécessaires et leur variété, les installer manuellement et en gérer les dépendances (puisqu'eux-mêmes peuvent avoir besoin d'autres blocs pour fonctionner) devient rapidement extrêmement fastidieux. Pour circonvenir à cette difficulté, on utilise un *gestionnaire*

de paquets, où les "paquets" sont les modules, et où leur recherche s'effectue dans un dépôt public recensant tous les modules publiés.

Registre public NPM

Comme tout écosystème logiciel un tant soit peu populaire, le gestionnaire de paquets de Node, NPM (pour *Node Package Manager*), référence un grand nombre de modules libres. Ces modules sont recensés dans le registre public NPM, et la recherche se fait donc dans cet espace.

Cependant, Node est très jeune (à peine trois ans), et la majorité des modules publiés est de piètre qualité, ou obsolète (l'environnement, de par sa jeunesse, évoluant très rapidement), ou encore trop spécifiques et publiés sans prise en compte des besoins des autres développeurs.

Une partie non négligeable de l'implémentation de l'outil décrit dans ce rapport a donc consisté en la recherche de modules fiables et efficaces, et en la détermination de critères de recherche permettant de répondre à ces besoins, ce qui n'est aujourd'hui pas des plus simples. Ce souci a d'ailleurs donné lieu à une discussion animée[1] avec les responsables du projet NPM, au sujet des contraintes de nommage à appliquer sur les modules publics.

Contraintes

En conséquence des points ci-dessus, voici les principales contraintes techniques.

Asynchronicité

La programmation avec Node.js se fait selon un modèle de programmation asynchrone, c'est-à-dire sans garantie sur la durée ni l'ordre d'exécution des opérations. Ce modèle vient contraster avec le mode impératif séquentiel des langages plus répandus. Il implique une plus grande complexité lors de la rédaction du code, en contrepartie de quoi il offre une vitesse d'exécution et une capacité de parallélisation des opérations supérieures.

Dans le domaine visé, voilà qui paraît tout à fait adapté, étant donné qu'une page web a une latence de chargement inconnue et variable, mais que l'on peut vouloir exécuter les tests sur différents navigateurs, voire différents sites, en parallèle. On souhaite également que la validation soit la plus rapide possible. Bien que les temps de chargement et d'exécution des tests soient négligeables face aux temps de chargement cumulés, toute accélération reste bonne à prendre, notamment dans les délais d'initialisation.

Une des contraintes importantes a donc été de se plier à ce modèle de programmation, et de le comprendre suffisamment bien pour en tirer les bénéfices d'un usage avancé.

Suivi des dépendances

Comme souligné plus haut, la jeunesse de Node se ressent tant dans les évolutions très rapides du cœur (environ une mise à jour par semaine) que des modules. Bien que tous soient versionnés et que l'on ne souffre donc pas directement d'instabilités extérieures, le suivi des modifications

[1] Voir https://github.com/isaacs/npm/issues/798.

est coûteux. Pire, il arrive régulièrement que des mises à jour cassent la compatibilité avec les versions précédentes, forçant à réécrire des parties du code.

Il a donc fallu, en plus de passer du temps à chercher les modules les plus indiqués, continuer à suivre leurs évolutions au quotidien, les mettre à jour régulièrement, gérer leurs dépendances et vérifier leurs compatibilités respectives.

Fiabilité

Même dans cet environnement mouvant, au vu de l'objectif, il fallait garantir une haute qualité de code. En effet, un outil de validation ne peut être vu comme fiable que si lui-même est validé, et si ses détections d'erreurs sont systématiques et non aléatoires.

La contrainte correspondante aura donc été de définir une méthode de certification efficace (c'est-à-dire peu coûteuse, pour qu'elle ne ralentisse pas le développement) et fiable, puis de l'appliquer systématiquement tout au long du projet.

Facilité d'utilisation

Comme toute nouvelle technologie, l'ajout d'un outil supplémentaire comporte un coût d'apprentissage non négligeable. Si l'on veut que l'outil développé soit un succès, il faut impérativement en faciliter l'approche et le rendre aussi agréable que possible à utiliser pour les usagers visés, à savoir des développeurs, mais aussi des responsables opérationnels, peut-être moins techniques mais plus encore concernés par les failles fonctionnelles de l'application validée.

Maintenabilité des tests

Comme on l'a largement souligné plus haut, le seul intérêt à créer un nouvel outil au lieu d'adapter un existant était de changer la manière dont les tests sont écrits, pour qu'ils soient enfin maintenables et que l'investissement en devienne rentable sur le long terme.

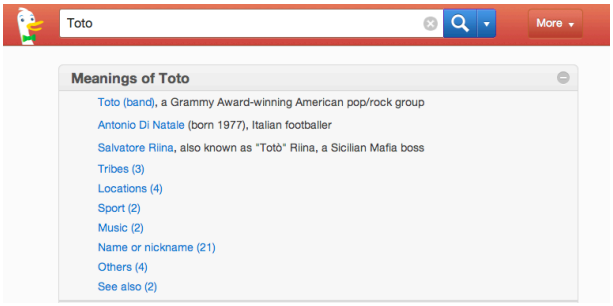
En conséquence, une contrainte doit être explicitement donnée sur une capacité de maintenabilité sensiblement supérieure aux solutions existantes.

Exemple

Avant de décrire le format des tests, présentons un cas d'usage qui servira de support à toutes les explications ultérieures. L'exemple que nous utiliserons sera DuckDuckGo, un moteur de recherche du même type que Google.

Page d'accueil de DuckDuckGo →

Une des fonctionnalités intéressantes de ce moteur est sa capacité à proposer des interprétations alternatives pour les termes ambigus.



Désambiguation de la requête "Toto" →

La présentation d'une telle désambiguation est une boîte appelée "boîte zero-click" dont le titre est "*Meanings of <terme ambigu>*".

Imaginons qu'en tant que personnel de DuckDuckGo, nous souhaitions valider le fonctionnement de ce système de manière automatique. Concrètement, la séquence d'actions sera la suivante :

1. Accéder à la page `https://duckduckgo.com`.
2. Entrer "Toto" dans le champ de recherche.
3. Envoyer la requête.
4. Attendre le chargement de la page de résultats.
5. Vérifier que la boîte zero-click est bien présente, et que son titre est "*Meanings of Toto*".

Nous allons nous baser sur ce scénario pour en présenter sa modélisation avec Watai.

Rédaction d'un jeu de tests

Pour répondre à la contrainte de maintenabilité, j'ai introduit des concepts utilisés depuis longtemps en programmation dans le format proposé.

Tout d'abord, un **découplage** entre les différentes composantes d'un test est imposé. En effet, au lieu d'une suite d'instructions séquentielles ciblant explicitement des éléments d'une page web (bouton, lien…) les uns après les autres pour les manipuler (clic, entrée de texte…), un test écrit avec Watai est séparé en trois types de descriptions.

Widgets

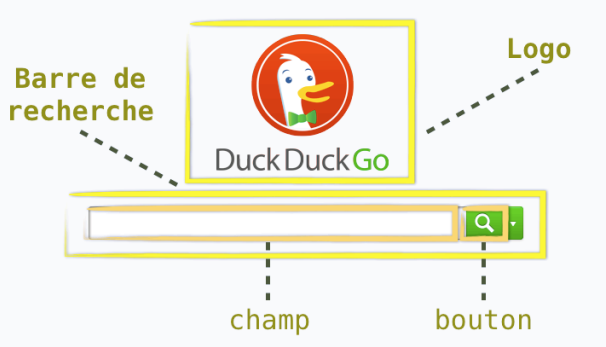
Un *widget* est un **composant** d'une page web. Par métonymie, c'est le nom donné à la description d'un tel composant dans Watai.

Pour comprendre cette notion, décomposons une page web selon ses composants.

Exemple

La page d'accueil précédente est constituée de deux composants : le logo du moteur et la barre de recherche. Cette dernière contient plusieurs **éléments**, dont en particulier :

- un champ de texte, pour entrer une requête ;
- un bouton permettant d'envoyer la requête.



La rédaction d'un test avec Watai va donc consister, entre autres, à décrire des composants.

Granularité

Bien évidemment, une description exhaustive des composants serait très coûteuse. Comme dans tout système de description, la question se pose du niveau de granularité à choisir pour maximiser le retour sur le temps investi à décrire.

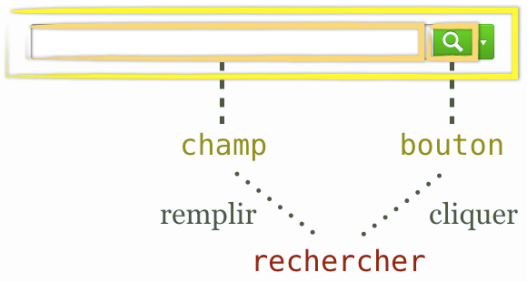
La réponse est simple : il faut décrire, et il suffit de décrire, les composants et éléments qui seront utilisés dans le scénario.

Dans notre exemple, le logo n'a aucun intérêt. On peut donc se contenter, pour cette page, de décrire le champ et le bouton de la barre de recherche. Il faudra également décrire la boîte zero-click et son titre, pour pouvoir en valider le contenu.

Actions

En plus des éléments, les composants peuvent également décrire des **actions** appliquées sur un ou plusieurs de leurs éléments.

Par exemple, à proprement parler, effectuer une recherche est une action en deux temps : le `champ` est d'abord rempli par un terme, puis le `bouton` est cliqué.



Avec Watai, le composant `Barre de recherche` se contente d'exposer l'action `rechercher`, en en masquant les détails d'implémentation. Cela signifie que, lors d'un test, on peut `rechercher` sans savoir comment se nomme le champ, s'il a un bouton…

Qualité logicielle

On retrouve ici les notions classiques dans la qualité logicielle de **cohésion forte** et d'**encapsulation**. Ces notions sont critiques à la maintenabilité des tests. Ainsi, si l'on décide de changer l'interface web et de remplacer le bouton par un lien, par exemple, il suffira de modifier un élément dans le composant pour que le test soit à nouveau opérationnel, sans avoir à réécrire la moindre logique de validation.

Features

La logique de validation en question est contenue dans un autre type de description : une *feature* décrit un **comportement** attendu et le **scénario** permettant d'en valider le fonctionnement.

Exemple

Notre exemple précédent pourrait avoir le contenu suivant :

- description : "Chercher un mot ambigu devrait ouvrir une boîte zero-click" ;
- scénario : *voir le scénario donné dans le chapitre précédent*.

Délais de chargement

Notons avant toute chose que tous les problèmes de temps de chargement de composants, que ce soit d'une page à l'autre, ou au sein d'une page dans le cas d'opérations asynchrones*, sont totalement prises en charge par Watai. Toutes les attentes sont implicites : un scénario est mis en attente jusqu'à ce que les composants sur lesquels il travaille soient disponibles.

Cela résout un problème récurrent et particulièrement pernicieux lors de l'écriture de tests de composants asynchrones, où l'utilisateur devait tenter de deviner le délai de chargement, au risque de bloquer le test pendant plus longtemps que nécessaire, ou inversement d'en rater l'exécution en tentant d'effectuer des actions sur un composant inaccessible[4].

Forme d'un scénario

La question des délais étant résolue, reste celle de la formalisation du scénario. Tous les outils étudiés les rédigent sous la forme d'une suite d'*actions* et d'*assertions*. Les *actions* agissent sur les composants d'une page, et les *assertions* comparent le contenu de composants à des valeurs prédéfinies.

En ce qui concerne les actions, comme la plupart des autres outils "innovants", Watai utilise bien entendu les widgets. C'est ce découplage qui permet la résilience soulignée plus haut aux changements d'interface. En revanche, pour les assertions, Watai se sépare de toutes les autres propositions en les rendant implicites, comme les attentes.

En effet, au lieu de comparaisons explicites, un scénario Watai se contente de définir un *état attendu*, et le système se charge d'effectuer les comparaisons nécessaires. Dans l'exemple précédent, on indiquerait donc simplement que le `titre` du composant `zero-click` attendu est "*Meanings of Toto*", au lieu d'en obtenir le contenu pour le comparer à cette valeur.

Qualité logicielle

Ce format permet d'augmenter la **lisibilité** des tests, notamment pour les personnes n'étant pas développeurs. En effet, le scénario ressemble bien plus à un *scénario* au sens premier du terme, c'est-à-dire le déroulement d'une histoire prédéfinie, plutôt qu'à une suite d'instructions entremêlées de code. Il est également beaucoup plus aisé d'isoler les définitions d'états attendus des actions, par opposition à une série d'instructions où actions et assertions se mêlent.

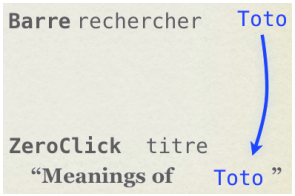
Enfin, l'abstraction des délais et des assertions permet d'augmenter la cohésion des scénarios via une **inversion de contrôle** : ceux-ci sont purement descriptifs, et ne contrôlent jamais directement le navigateur.

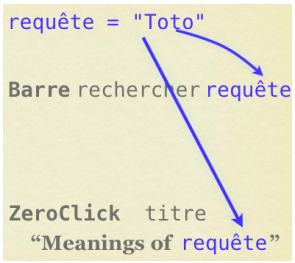
Data

Le dernier type de composant d'un jeu de tests Watai définit des **constantes** rendues accessibles tout au long du test.

Exemple

Nous avons utilisé comme terme ambigu "Toto". En conséquence, nous avons défini le titre attendu de la boîte zero-click comme "*Meanings of Toto*". Qu'adviendra-t-il si l'on souhaite utiliser un autre terme supposé ambigu pour vérifier le comportement de la désambiguation ? Ou encore si l'on souhaite effectuer encore d'autres contrôles sur la présence du terme de recherche dans d'autres composants ? Il faudra systématiquement recopier le terme "Toto", et la probabilité d'oublier de changer l'une des occurrences lors d'une modification augmente, diminuant ainsi la fiabilité du processus de validation.





Le terme "Toto" ainsi utilisé est en réalité une constante, et devrait être défini comme tel. On écrira donc un fichier de type *data* contenant la définition de `requête` comme "Toto", et on utilisera systématiquement la constante `requête` au sein des features.

Importation

L'utilisation de ce système n'est bien évidemment pas obligatoire, et l'utilisateur est libre de ne pas s'en servir. Néanmoins, il s'agit d'un point important pour la maintenabilité des tests.

Qui plus est, vu la simplicité du format, il paraît tout à fait envisageable d'écrire un programme automatisant l'extraction de constantes depuis d'autres sources, comme par exemple une base de données, un tableur, ou encore des annotations dans le code. On aboutit ainsi à un test totalement intégré dans un processus de suivi de la qualité plus complet.

Qualité logicielle

Le dicton **DRY** (*Don't Repeat Yourself*, ne vous répétez pas) résume la majorité de l'intérêt de cette fonctionnalité. On sait en effet que la duplication mène systématiquement à des oublis lors de modifications ultérieures, et à du code moins compréhensible ("Toto" est-il présent en tant qu'élément recherché ? S'agit-il d'une coïncidence ?). La minimiser par l'usage de constantes est une technique de base de la programmation.

Configuration

Le dernier élément manquant à l'exécution d'un jeu de tests est un espace de configuration définissant les moyens d'accès au site évalué. Notamment, l'adresse à laquelle il est accessible, et les caractéristiques du navigateur qui devra exécuter le test.

Ces points sont définis dans un fichier de configuration situé à proximité des composants du jeu de test (widget, feature, data). Ils peuvent donc être versionnés et partagés comme tout autre élément de code.

Néanmoins, étant donné que certains éléments de configuration peuvent changer d'un environnement à l'autre (navigateurs disponibles, vitesse d'accès au réseau…), j'y ai ajouté un système de chargement en cascade. Ainsi, tout élément non défini explicitement par un fichier de configuration spécifique à un jeu de tests peut être défini par des préférences spécifiques à l'utilisateur, ou encore à la machine sur laquelle le test est exécuté. De cette manière, il devient envisageable d'avoir une configuration différente pour les machines de développement, pour un serveur d'intégration continue et pour un système validant le site web en production, tout en continuant à partager le plus d'informations de configuration possible.

Stockage

Un jeu de tests est donc constitué d'un dossier contenant un fichier de configuration, et autant de fichiers de description de widgets, de features et de data que souhaité. Tous ces fichiers sont rédigés en JavaScript, et doivent respecter une syntaxe spécifique dépendant du type décrit.

Exécution d'un jeu de tests

La forme d'un jeu de tests ayant été correctement définie, voyons à présent comment l'exécuter.

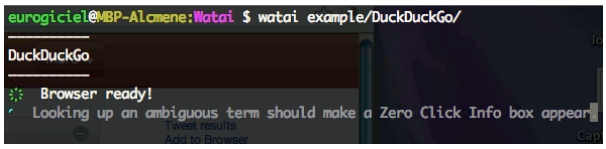
Dépendances

Pour pouvoir exécuter un test, un serveur Selenium doit être accessible pour commander les navigateurs. Un tel serveur peut être soit lancé sur la machine du développeur validant ses modifications, soit par exemple rendu accessible sur une machine dédiée à cet effet pour toute une équipe. Il existe également des entreprises offrant un service de mise à disposition d'infrastructures de validation. Cette diversité est appréciable pour le passage à l'échelle qu'elle offre.

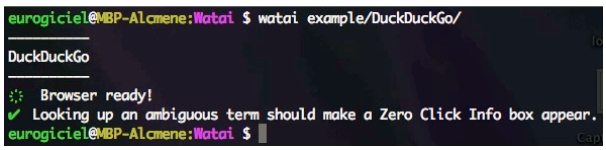
Apparence

L'application finale se présente sous la forme d'un utilitaire exécuté à la ligne de commande. Un jeu de tests est lancé en passant en paramètre du programme le chemin d'un dossier en contenant la description, comme indiqué plus haut.

Une fois le navigateur démarré, les fonctionnalités sont évaluées les unes après les autres, et les résultats sont affichés au fur et à mesure qu'ils sont obtenus.



Lorsque toutes les fonctionnalités ont été évaluées, une notification est affichée sur la machine ayant lancé le test. En cas d'erreur, les fonctionnalités ayant échoué sont mises en valeur.



Distribution

Nous avons vu comment fonctionne l'application réalisée. Néanmoins, comme nous l'avons souligné tantôt, pour pouvoir être utilisée efficacement, encore faut-il qu'elle soit facile à utiliser. Le premier pas vers la facilité d'utilisation est la facilité d'installation, et c'est pourquoi la question du mode de distribution de l'application est cruciale à son succès.

Page d'accueil

Le code source de l'application et sa documentation sont visibles publiquement sur le site d'hébergement de code GitHub[1]. Les manuels de référence, d'installation et une aide sont également disponibles sous la forme d'un wiki*.

Cette page d'accueil sert de point d'entrée à la présentation du projet comme à son installation, et son ouverture et ses capacités d'édition collaborative assurent qu'elle pourra être corrigée, en cas de besoin, par tout utilisateur.

Installation

Procédure

L'application est distribuée sous la forme d'un paquet visible sur le dépôt public NPM. En conséquence, une fois l'environnement Node installé via un assistant, l'ajout de Watai est l'affaire d'une ligne de commande : `npm install watai`. Pour utiliser Watai sur sa machine, comme expliqué plus haut, l'utilisateur devra également télécharger Selenium et le lancer. Seule cette étape supplémentaire différencie cette installation de n'importe quelle autre application.

La procédure d'installation a été écrite pour et vérifiée sur la majorité des plateformes disponibles, y compris Windows. Pour m'assurer de sa clarté et de son fonctionnement, j'ai mené des tests utilisateurs* sur la procédure d'installation avec différentes personnes. Ces essais m'ont permis de corriger des problèmes sur certaines plateformes, en particulier l'installation sur un réseau d'entreprise protégé par un proxy.

Vérification

Pour que l'utilisateur puisse facilement vérifier que l'installation s'est bien passée, une commande est disponible : `watai --installed`. Cela permet d'éviter la frustration qui émerge de l'incapacité à déterminer si un éventuel dysfonctionnement a pour origine une mauvaise installation ou un mauvais usage.

1 https://github.com/MattiSG/Watai

Documentation

J'ai rédigé différents types de documentation, afin de répondre aux besoins de chacun des acteurs amenés à interagir avec Watai ou son code source.

Décideur

Le choix d'un logiciel étant souvent complexe vu le nombre de possibilités, j'ai souhaité aider à ce processus en fournissant les documents suivants :

- une description des contextes dans lesquels Watai est indiqué, ses points forts et ses limitations ;
- une définition précise de ce qu'est Watai et de ce qui le différencie de ses alternatives ;
- la version intermédiaire de ce rapport, qui explore en profondeur la problématique des tests.

Utilisateur

La documentation utilisateur est fournie sous la forme de :

- un fichier de type "Lisez-moi" présentant succinctement le projet et la procédure d'installation ;
- un document de type "problèmes habituels" (*troubleshooting*) listant les problèmes les plus couramment rencontrés, leurs symptômes et les solutions envisageables ;
- une introduction exposant de manière concise les enjeux des tests d'intégration et les limitations des solutions actuelles ;
- un tutoriel fournissant une base à l'utilisation de Watai, basé sur le même exemple que celui qui a été déroulé dans les parties précédentes.

Ces documents ont tous été validés par des tests utilisateurs. Ils ont été jugés par les usagers pilotes comme "agréables à lire" et "bien conçus". Ensemble, ils permettent à un nouvel utilisateur d'être opérationnel en une vingtaine de minutes, ce qui répond à la contrainte de simplicité.

Développeur

Le code source de Watai est libre et hébergé sur un dépôt public, comme annoncé plus haut. Néanmoins, pour que cette ouverture au public ait réellement un sens, il faut s'assurer qu'en plus d'être disponible, le code soit compréhensible et facilement modifiable par tous. Pour répondre à cette exigence, mon code est largement documenté techniquement (63 lignes de documentation pour 100 de code), avec le formalisme spécifique des conventions Google Closure[5] + JSdoc-Toolkit, qui permettent de générer une référence lisible par un navigateur web.

En plus de cette documentation technique classique, les éléments suivants sont fournis pour faciliter la collaboration et l'intégration :

- liste des bibliothèques utilisées et raisons de leur choix lorsque plusieurs étaient envisageables ;
- liste et documentation des outils de développement utilisés ;
- conventions de style[6].

Gestion de projet

Après avoir décrit précisément l'application réalisée pour répondre aux besoins exprimés précédemment, examinons les moyens mis en œuvre pour s'assurer de sa complétion et de sa qualité.

Méthodologie

Suivi et planification

D'une volonté commune, nous avons souhaité que le projet soit géré selon les méthodes agiles. Nous avons donc travaillé avec Scrum, une méthodologie qui décompose le temps de travail en cycles de durée fixe ("itérations") et évalue les tâches à accomplir au début de chaque itération. Cela permet des ajustements précis des fonctionnalités présentes dans le programme développé, afin de répondre facilement et efficacement aux changements et aux imprévus.

Gestion du temps

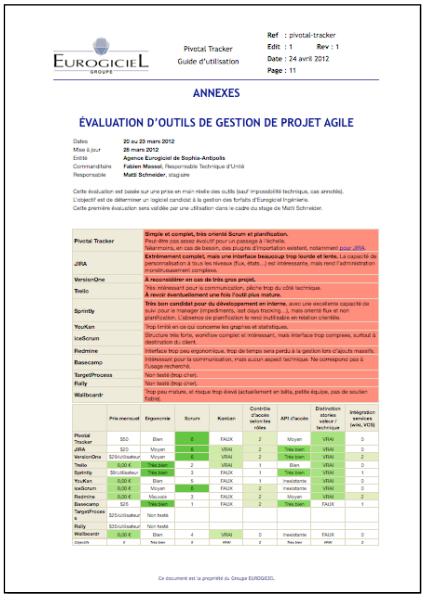
Pour garantir une efficacité maximale, j'ai travaillé en grande majorité selon la technique Pomodoro. Il s'agit d'une méthode d'organisation qui décompose le temps de travail en tranches de 25 minutes dédiées à une tâche spécifique et pendant lesquelles on refuse d'être interrompu, suivies de 5 minutes de repos. Cette stratégie permet de conserver un état de concentration pendant une durée supérieure à l'organisation classique consistant en une tentative de travail continu pendant plusieurs heures suivies d'une pause de type pause-café d'une quinzaine de minutes. Cette technique a par la suite été essayée et adoptée par mon encadrant.

Outils

Suivi et planification

Sélection

J'ai évalué et comparé, selon une méthode documentée et proche de celle employée lors de la comparaison des outils de test, de nombreux outils de gestion de projet en mode agile.


EUROGICIEL GROUPE

Pivotal Tracker
Guide d'utilisation

Ref : pivotal-tracker
Edit : 1 Rev : 1
Date : 24 avril 2012
Page : 11

ANNEXES

ÉVALUATION D'OUTILS DE GESTION DE PROJET AGILE

Dates 20 au 23 mars 2012
Mise à jour 26 mars 2012
Entité Agence Eurogiciel de Sophia-Antipolis
Commanditaire Fabien Massol, Responsable Technique d'Unité
Responsable Matti Schneider, stagiaire

Cette évaluation est basée sur une prise en main réelle des outils (sauf impossibilité technique, cas annotés).
L'objectif est de déterminer un logiciel candidat à la gestion des forfaits d'Eurogiciel Ingénierie.
Cette première évaluation sera validée par une utilisation dans le cadre du stage de Matti Schneider.

Pivotal Tracker	Simple et complet, très orienté Scrum et planification. Peut-être pas assez évolutif pour un passage à l'échelle. Néanmoins, en cas de besoin, des plugins d'importation existent, notamment pour JIRA.
JIRA	Extrêmement complet, mais une interface beaucoup trop lourde et lente. La capacité de personnalisation à tous les niveaux (flux, états…) est intéressante, mais rend l'administration monstrueusement complexe.
VersionOne	A reconsidérer en cas de très gros projet.
Trello	Très intéressant pour la communication, pêche trop du côté technique. A revoir éventuellement une fois l'outil plus mature.
Sprintly	Très bon candidat pour du développement en interne, avec une excellente capacité de suivi pour le manager (impediments, last days tracking…), mais orienté flux et non planification. L'absence de planification le rend inutilisable en relation clientèle.
YouKan	Trop limité en ce qui concerne les graphes et statistiques.
iceScrum	Structure très forte, workflow complet et intéressant, mais interface trop compress, surtout à destination du client.
Redmine	Interface trop peu ergonomique, trop de temps sera perdu à la gestion lors d'ajouts massifs.
Basecamp	Intéressant pour la communication, mais aucun aspect technique. Ne correspond pas à l'usage recherché.
TargetProcess	Non testé (trop cher).
Rally	Non testé (trop cher).
Wallboardr	Trop peu mature, et risque trop élevé (actuellement en bêta, petite équipe, pas de soutien Fabrice).

	Prix mensuel	Ergonomie	Scrum	Kanban	Contrôle d'accès selon les rôles	API d'accès	Distribution stories valeur / technique	Intégration services (wiki, VCS)
Pivotal Tracker	$50	Bien	6	FAUX	2	Moyen	VRAI	0
JIRA	$21	Moyen	6	VRAI	2	Très bien	VRAI	3
VersionOne	$25/utilisateur	Moyen	5	VRAI	2	Très bien	VRAI	3
Trello	0,00 €	Très bien	2	VRAI	1	Bien	VRAI	0
Sprintly	$8/utilisateur	Très bien	3	FAUX	1	Très bien	VRAI	1
YouKan	0,00 €	Bien	5	FAUX	1	Inexistante	VRAI	0
iceScrum	0,00 €	Moyen	6	FAUX	2	Inexistante	VRAI	0
Redmine	0,00 €	Mauvais	5	FAUX	2	Moyen	VRAI	2
Basecamp	$25	Très bien	1	FAUX	2	Très bien	FAUX	1
TargetProcess	$25/utilisateur	Non testé						
Rally	$25/utilisateur	Non testé						
Wallboardr	0,00 €	Bien	4	VRAI	0	Inexistante	FAUX	0
Gauche	0	Très bien	6	VRAI	2	Très bien	VRAI	2

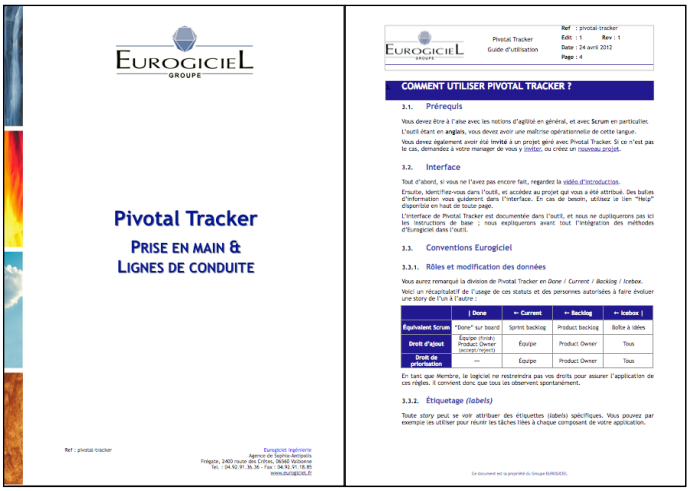
Ce document est la propriété du Groupe EUROGICIEL

Au final, j'ai émis une recommandation pour l'usage de Pivotal Tracker. Après deux mois d'utilisation pilote sur trois équipes distinctes et des retours positifs, tant des collaborateurs que des clients, Eurogiciel Sophia a validé le choix de cet outil et engagé les investissements correspondants.

Ce comparatif a par ailleurs été publié, à titre personnel, sur mon site. Il a rencontré un certain succès dans ce cadre, avec plusieurs recommandations sur le réseau Twitter par exemple.

Formation

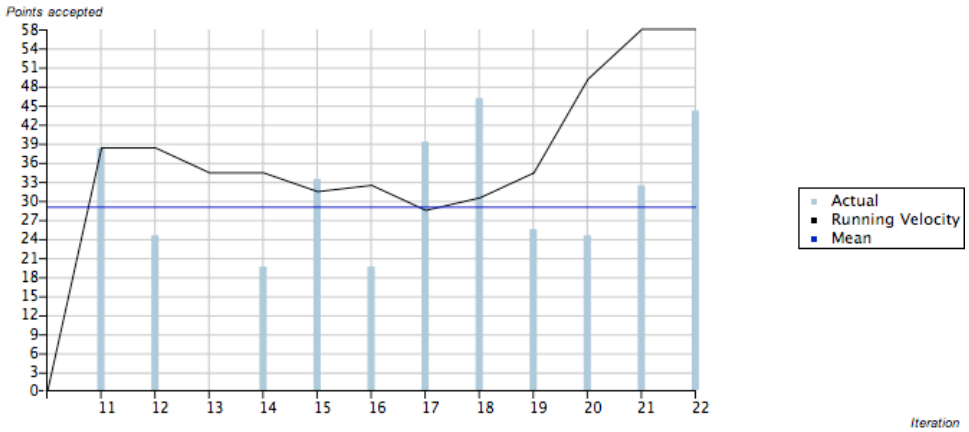
Au terme de la sélection de l'outil, pour faciliter son déploiement interne, j'ai été chargé de rédiger un guide d'introduction à l'usage de Pivotal Tracker, et une charte d'utilisation prescrivant des conventions spécifiques à appliquer au sein d'Eurogiciel. Ce document de 11 pages a été et sera distribué à toutes les équipes amenées à travailler avec cet outil.



Usage

Nous avons donc utilisé cet outil pour suivre et planifier les évolutions du projet avec la méthode Scrum. Cela signifie avant tout lister les tâches à accomplir, puis les organiser par ordre de priorité et évaluer l'effort nécessaire à leur réalisation. Cette évaluation du coût en temps et du gain en valeur métier permet de trouver les meilleurs compromis pour chaque itération, et donc, cumulés sur la durée du projet, pour l'intégralité du produit.

Une fois quelques itérations passées, on peut calculer la vélocité de l'équipe, c'est-à-dire la quantité de tâches qu'elle peut accomplir, et s'en servir pour planifier plus efficacement.



Évolution de vélocité →

D'autres métriques permettent de s'assurer de la bonne santé du projet et de sa capacité à respecter les jalons, comme par exemples les graphes de type *burndown*, *burnup* (voir annexe)… Utilisées tout au long du projet, elles nous ont permis d'ajuster l'étendue du projet selon les difficultés naissantes sans en compromettre l'objectif final.

Suivi technique

Tout le code source de l'application, ses jalons et ses anomalies sont visibles publiquement sur la page GitHub du projet. Cette plateforme est complète et efficace pour la gestion technique, avec l'avantage évident de centraliser tout ce qui a trait à l'aspect technique du projet.

Les anomalies détectées, tout comme les idées de fonctionnalités futures, sont donc ajoutées aux "*issues*" de GitHub. Cela permet tout d'abord d'être totalement transparent avec les utilisateurs, et de leur permettre de signaler facilement les éventuelles anomalies qu'ils détecteraient ; ensuite de référencer ces dernières directement depuis les messages de modification de code, afin de notifier immédiatement d'une résolution, par exemple. De même, la possibilité de recenser

les fonctionnalités au même niveau que les anomalies, associée à la description des jalons prévus, permet d'offrir une feuille de route claire.

Communication

Étant donné la taille de l'équipe technique, à savoir une seule personne, nous n'avons pas eu à mettre en place de méthodes de collaboration spécifiques. En revanche, nous avons décidé d'optimiser la communication entre mon encadrant et moi-même en partageant un espace de travail commun. Cela nous a permis de construire rapidement, puis de conserver, une vision commune sur le produit à réaliser.

Pour obtenir cette capacité à communiquer sans qu'elle ne diminue nos productivités respectives, nous avons défini un certain nombre d'artefacts et techniques minimisant les interruptions.

Nous avons par exemple mis en place un *happiness board*[7] sur un des murs. Cet artefact se comporte comme un *émetteur d'informations*. Un émetteur d'informations est un objet qui fournit en permanence des informations sur un aspect spécifique de l'équipe et/ou du projet, sans pour autant demander une attention immédiate comme une communication verbale directe. Il devient donc possible pour chacune des parties prenantes d'utiliser les informations émises comme base à une discussion ou une prise d'action, mais lors de sa disponibilité et non obligatoirement au moment où l'information arrive. Ce travail en *slow-time*[8] permet de respecter le temps de tous sans compromettre la qualité des échanges. Bien au contraire, en plaçant les discussions à un moment de disponibilité conjointe mais uniquement lorsque le besoin s'en fait sentir, par opposition à des réunions systématiques et sans objet prédéfini, on en vient à l'augmenter.

Dans cette lignée, nous avons également mis en place un moyen de signaler une question à l'autre pour sa prochaine plage de disponibilité, sans pour autant passer par la rédaction de mails détaillés : de simples indicateurs de besoin de type post-its, associés à un mot-clé deviennent autant de "tickets de conversation" à traiter.

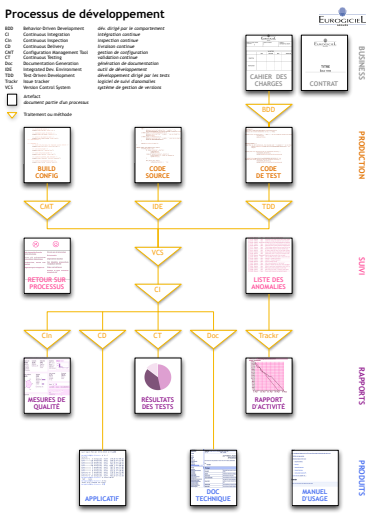
Ingénierie

Complétons la présentation de ces méthodes sociales par les méthodes techniques utilisées.

Infrastructure de développement

Au vu de l'absence d'infrastructure interne, la tâche de définir les outils à utiliser pour disposer d'un environnement de développement et de gestion de la qualité m'a été confiée.

J'ai donc commencé par analyser l'architecture des processus de développement à mettre en place, avant de la faire valider et d'en prioriser les outils de manière collaborative. Ce document (en annexe) est désormais distribué aux collaborateurs intégrant les projets internes, et sert de feuille de route pour l'infrastructure qui sera mise en place dans les prochains mois.



Automatisation

J'ai mis en place un système d'automatisation de la plupart des tâches assurant une bonne qualité de code, afin de pousser à leur usage.

Des solutions d'automatisation génériques existaient bien sûr déjà, mais un examen rapide ne m'a pas convaincu quant au retour sur investissement de leur apprentissage et de l'ajout de complexité. En conséquence, l'outil d'automatisation est un simple script (shell) dont les fonctionnalités ont été ajoutées au fur et à mesure de l'avancement du projet, et complété petit à petit, lorsque certaines tâches étaient répétées trop souvent.

Aujourd'hui, cet outil est doté des capacités suivantes, toutes documentées :

- exécution des tests ;
- calcul de couverture de code (proportion de code source validé par des tests) ;
- calcul de ratio code / documentation ;
- génération de documentation technique (au choix publique ou privée) ;
- création d'une archive contenant un paquet déployable de la dernière version stable ;
- publication de la dernière version des exemples de code dans la documentation ;
- exécution du programme principal.

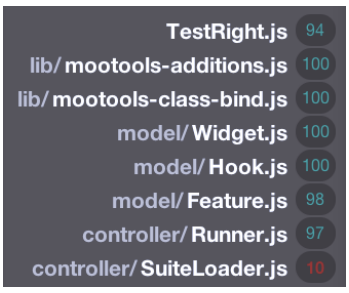
Cet outil permet un gain de productivité non négligeable, et assure surtout que d'éventuels autres développeurs puissent facilement valider et incorporer leurs modifications.

Tests

Les tests sont exécutés par un outil de référence dans l'écosystème de Node : Mocha. Il a en particulier l'intérêt de s'intégrer facilement avec JScoverage, outil utilisé pour calculer la couverture de code.

Ces tests couvrent une importante partie du code (78 % au total), assurant ainsi un filet de sécurité important en cas de modification.

Analyse détaillée de la couverture de code →



La difficulté principale consiste en l'écriture de tests dont l'exécution soit suffisamment rapide pour être la plus systématique possible, tout en conservant une couverture importante.

Vu le système testé, il est malheureusement inévitable d'avoir une forte latence, puisque toute tentative de contrôle d'un navigateur implique son démarrage, et la gestion de cas limite tels que la fin de session nécessite ensuite de le quitter, ce qui est également lent.

La solution employée consiste donc d'abord à organiser les tests pour réutiliser le plus possible les instances de navigateurs. J'ai ensuite ajouté la possibilité d'exécuter de manière aisée un

sous-ensemble des tests, permettant ainsi une validation rapide des composants modifiés sans systématiquement vérifier le système entier.

Gestion de versions

De par mes analyses personnelles, j'avais depuis longtemps déterminé les avantages de Git sur les autres logiciels de gestion de versions*. J'en ai donc proposé l'usage pour ce projet, et en ai démontré les forces. La présentation de la plus-value ayant été convaincante, j'ai été chargé de concevoir une formation à destination des autres ingénieurs d'Eurogiciel.

J'ai donc imaginé une formation à Git, et créé les supports nécessaires à deux sessions d'environ une demi-heure, la moitié de ce que serait une formation complète. Ces éléments permettent d'aboutir à une connaissance suffisante pour devenir opérationnel et capable de collaborer efficacement. Des sessions pilotes ont été effectuées avec des ingénieurs, qui ont permis de valider le contenu et la qualité des présentations.

Évolutions possibles

Bien que fonctionnel et utile en l'état, l'outil développé pourrait bien entendu être amélioré sur plusieurs points, que nous allons examiner ici.

Packaging

Malgré tout le soin apporté à fournir une procédure d'installation aussi simple que possible, rien ne peut être plus simple qu'une installation automatique. Il serait donc intéressant d'intégrer le serveur Selenium WebDriver au paquet installé par l'utilisateur, et de le lancer et quitter automatiquement avec les tests. Cela permettrait de masquer totalement la complexité de l'exécution. Il faudrait néanmoins conserver la possibilité d'exécuter les tests sur un serveur distant, dans le cas où un serveur d'intégration est disponible.

Syntaxe

La syntaxe à respecter pour rédiger des tests est raisonnable et tente d'être aussi simple que possible. On pourrait néanmoins encore l'améliorer, en abandonnant la syntaxe JavaScript et en la remplaçant par un nouveau format spécifique. En contrepartie, il faudrait écrire un analyseur syntaxique dédié, ce qui est assez coûteux.

Navigateurs

La variété de navigateurs supportés est largement satisfaisante en l'état. En revanche, il serait intéressant de paralléliser la validation sur plusieurs navigateurs en un seul test, plutôt que de forcer l'utilisateur à choisir un navigateur par exécution.

Par ailleurs, comme suggéré lors de l'analyse des besoins, l'usage d'un navigateur en mode *headless*, c'est-à-dire sans interface graphique, permettrait d'accélérer considérablement la validation. Le projet GhostDriver, dont c'est l'objectif, a largement avancé pendant la période de mon stage, mais pas suffisamment pour pouvoir être intégré. Je n'ai malheureusement pas pu collaborer à sa complétion suite à des difficultés de communication avec l'équipe responsable du projet, mais l'intégration devrait être aisée une fois qu'il sera finalisé.

Parallélisation des tests

Il serait très rentable de paralléliser la validation de différentes fonctionnalités, plutôt que de les exécuter séquentiellement. Néanmoins, les dépendances étant malheureusement monnaie courante (identifier un utilisateur étant par exemple un préalable nécessaire à des actions sécurisées), il faudrait offrir un moyen simple d'expliciter ces dépendances. Cela permettrait ensuite de définir un graphe des fonctionnalités, et de définir la stratégie de parallélisation la plus efficace.

Conclusion

Je suis heureux d'avoir eu l'occasion de travailler sur un projet utilisant des technologies que je souhaitais maîtriser depuis longtemps, et ce dans un cadre méthodologique agréable. Je suis convaincu que cette expérience m'a été extrêmement bénéfique, tout à la fois pour valider mes compétences existantes avec un encadrement professionnel, pour découvrir le monde du service et pour avoir enfin eu l'opportunité d'effectuer plusieurs études que j'envisageais de faire depuis longtemps à titre personnel.

J'ai beaucoup apprécié les relations avec mes encadrants, qui ont été à mon écoute et m'ont toujours témoigné beaucoup de respect et de confiance.

J'ai enfin été satisfait de mener ce projet à bien, et ce malgré les difficultés organisationnelles qui sont apparues, avec notamment le départ imprévu de mon maître de stage à mi-parcours et la solitude qui en a découlé. Bien que plusieurs fonctionnalités supplémentaires seraient appréciables, tel que souligné plus haut, le produit résultant me semble correspondre aux besoins analysés en premier lieu et apporter une réelle plus-value sur le marché actuel de la validation d'applications web.

J'espère avoir l'opportunité de faire connaître ce produit et de le voir utilisé par la communauté.

Je remercie Eurogiciel et les personnes qui ont œuvré en son sein pour l'accomplissement de cette possibilité ; également l'ensemble des enseignants, personnel et acteurs de l'école Polytech'Nice pour m'avoir permis, à travers l'ensemble de ma formation, d'aboutir à ce projet.

Matti Schneider, le 1er septembre 2012

Glossaire

AJAX

Asynchronous JavaScript And XML — Technologie aujourd'hui largement utilisée dans les sites et applications web, qui permet le chargement d'informations depuis le serveur distant sans recharger la page.

API

Application Programming Interface — Moyen d'appeler du code depuis un autre programme.

Agile (méthode)

Méthode de travail, actuellement principalement utilisée dans le cadre de projets informatiques bien que non spécifique à ce contexte, qui se veut plus pragmatique que les méthodes traditionnelles. L'accent est mis sur une implication du client dans le projet, sur la rapidité du développement et sur l'atomicité des tâches accomplies, en s'affranchissant d'une planification trop lourde et souvent hasardeuse.

Application web

On désigne généralement par "site web" un ensemble de pages permettant au public d'accéder à des informations, sans pour autant pouvoir agir dessus. Une "application web", ou webapp, par opposition, est un logiciel au même titre qu'un programme installé sur un ordinateur personnel, mais dont l'interface est accessible via un navigateur.

Asynchrone (chargement)

Dans le contexte d'une page web, opération chargeant des données depuis le serveur distant sans recharger la page entière. *Voir AJAX.*

Feature freeze

Point temporel dans un cycle de développement logiciel où l'on décide d'interdire tout ajout de nouvelles fonctionnalités. Ce choix délibéré a généralement pour but de permettre de finaliser un état stable d'un programme pour aboutir à une livraison.

Flex

Flex est une solution permettant de créer et de déployer des applications Internet riches (RIA) multiplate-formes grâce à la technologie Flash.

Intégration continue

L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.

Libre (logiciel)

Logiciel dont le code source (texte écrit par le programmeur, permettant le fonctionnement du logiciel) peut être lu, copié et modifié par tous.

Logiciel de gestion de versions

Logiciel permettant de stocker un ensemble de fichiers et l'historique de toutes leurs modifications.

Patron de conception *(Design pattern)*

En informatique, et plus particulièrement en développement logiciel, un patron de conception est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.

QA

Quality Assurance — Département de test et de validation, dédié uniquement à la qualité du code et des livrables.

SSII

Société de Services en Ingénierie Informatique — Entreprise vendant le temps de travail des ingénieurs qu'elle emploie.

SUT

System Under Test — Ensemble des éléments, logiciels et potentiellement matériels, évalués par un test.

Test utilisateur

Méthode permettant d'évaluer un produit en le faisant tester par des utilisateurs dans un environnement contrôlé, généralement avec un observateur notant les activités de l'utilisateur pilote sans influencer son usage, afin d'obtenir un rapport quant aux failles d'usabilité détectées.

Wiki

Site web dont les pages peuvent être modifiées par tout visiteur.

Bibliographie

[1] Code Complete

2004, McConnell Steve

[2] Selenium Design Patterns and Development Strategies

Selenium contributors

https://code.google.com/p/selenium/wiki/DesignPatterns

[3] Test Automation Snake Oil

1999, Bach, James — 14th International Conference and Exposition on Testing Computer Software

http://www.satisfice.com/articles/test_automation_snake_oil.pdf

[4] How to Lose Races and Win at Selenium

2011, Sauce Labs

http://sauceio.com/index.php/2011/04/how-to-lose-races-and-win-at-selenium

[5] Annotating JavaScript for the Closure Compiler

2010, Google

https://developers.google.com/closure/compiler/docs/js-for-compiler

[6] Google JavaScript Style Guide

Whyte A., Jervis B., Pupius D., Arvidsson E., Schneider F., Walker R.

http://google-styleguide.googlecode.com/svn/trunk/JavaScriptguide.xml

[7] Happiness Metric - The Wave of the Future

2010, Sutherland Jeff

http://scrum.jeffsutherland.com/2010/11/happiness-metric-wave-of-future.html

[8] Real time vs. slow time — and a defense of sane work hours

2011, 37signals

http://37signals.com/svn/posts/2888-real-time-vs-slow-time-and-a-defense-of-sane-work-hours

[9] Web Site Test Tools — More than 560 tools listed in 14 categories

Rick Hower

http://www.softwareqatest.com/qatweb1.html

[10] Outil de gestion de Plan de Déplacements Campus — Rapport de stage SI4

2011, Matti Schneider

http://mattischneider.fr/portfolio/Content/pdc/content/pdc_rapport_si4.pdf

[11] Browser testing done right

2011, Steven Hazel, Sauce Labs

http://sauceio.com/index.php/2011/05/testright-browser-testing-done-right/

[12] Embracing change in a less-than-ideal world

2012, Mel Llaguno, Coverity — SelConf '12

http://www.youtube.com/watch?v=Wzj8UXdcElo

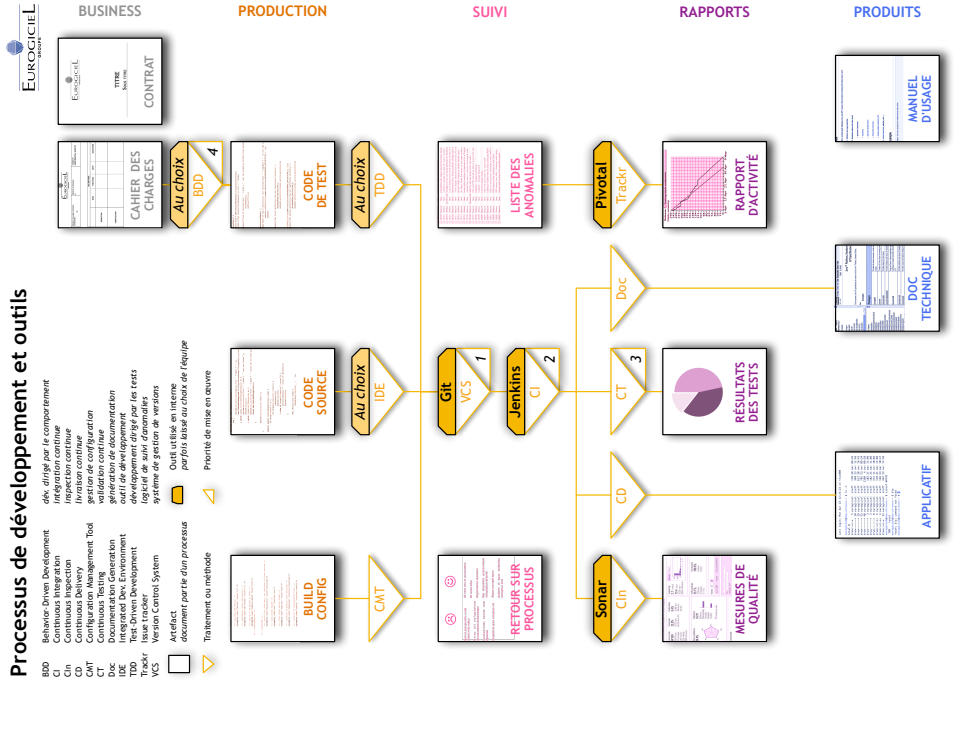
[13] Writing Well-Abstracted Self-Testing Automation On Top Of Jello-O

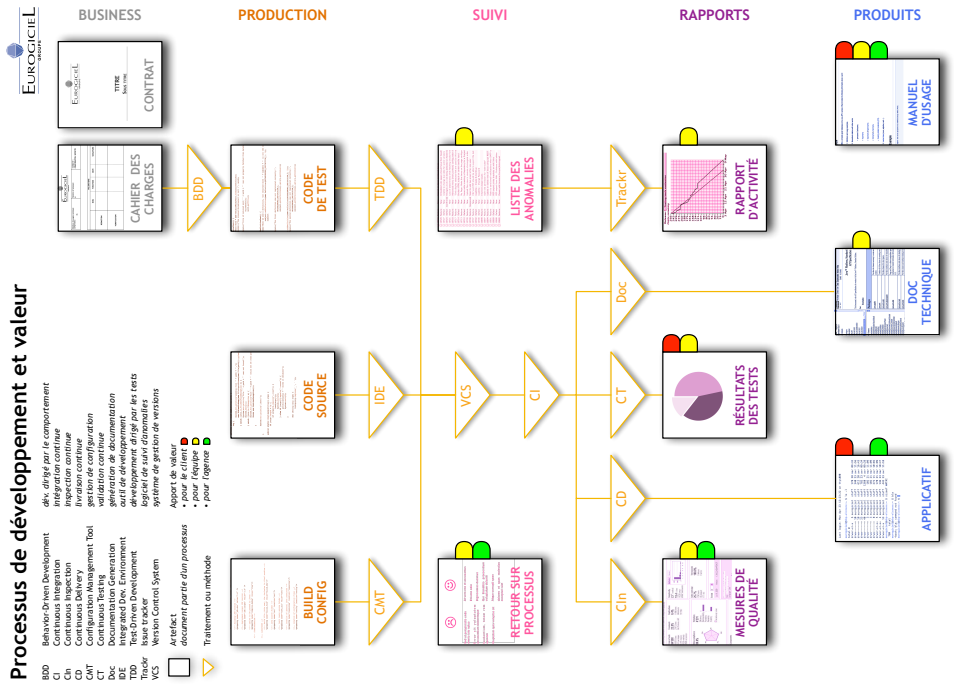
2012, Dan Cuellar, Zoosk — SelConf' 12

http://www.youtube.com/watch?v=_lJBHx4EE00

Annexe

Architecture des processus





Plan de haut niveau prévisionnel

High level plan

Matti Schneider

April 2nd, 2012

WATAI
Web Application Testing
& Automation Internship

Ambition

Visualize a cartography of all parts of a website and rank them on several dimensions.
Examples: functional testing, performance, standards compliance, accessibility, semantic field…

Objectives

- **Reduce QA costs.**
- **Assess quality.**
- **Assess regression risks.**
- **Visualize site navigation model.**

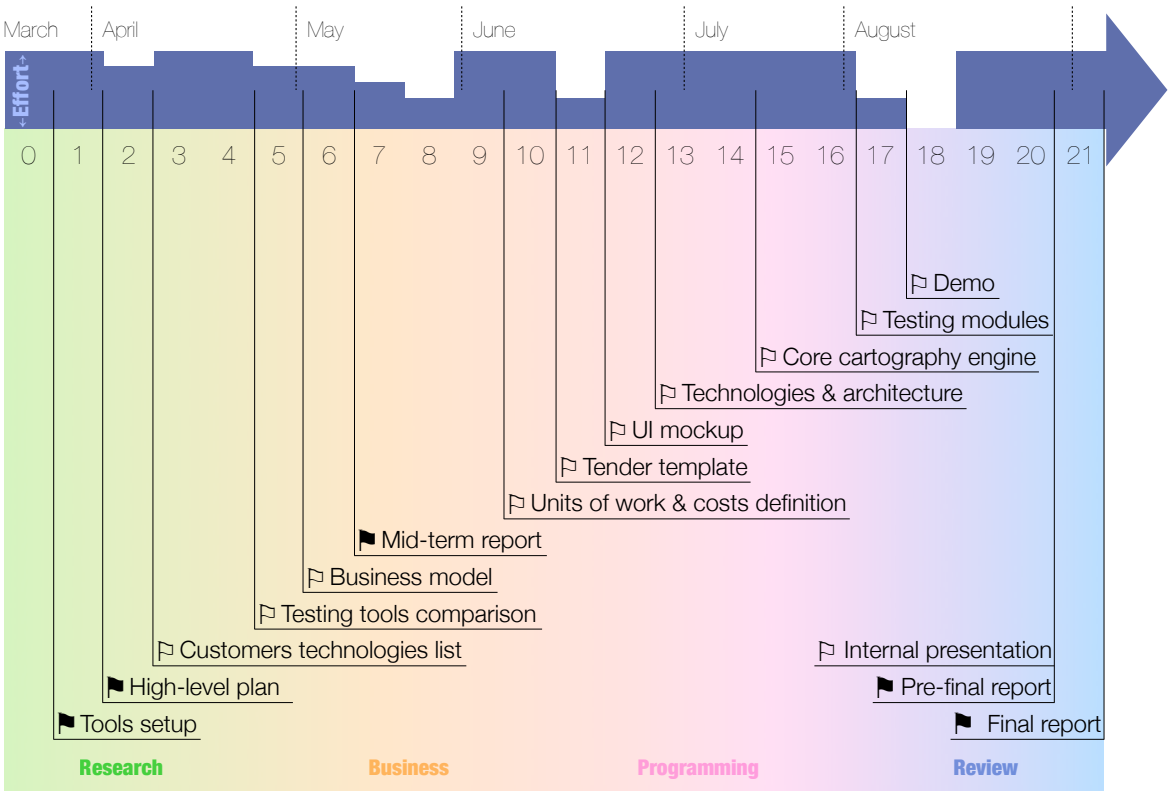
Metrics

- ➡ QA costs evolution.
- ➡ Amount of test-covered parts & tests results.
- ➡ Regressions frequency.
- ➡ Subjective understanding of site hierarchy.

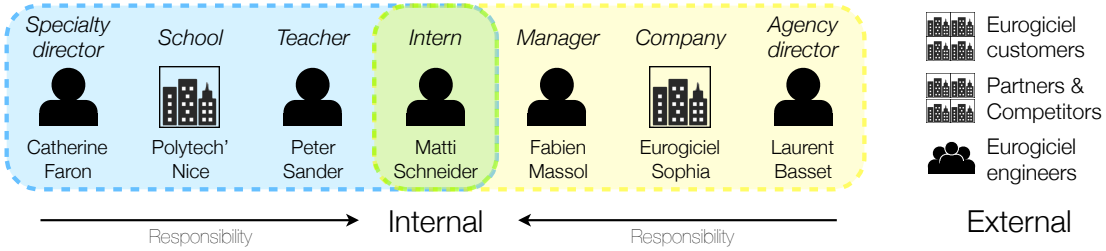
Roadmap

⚒ Methodology: Scrum
↻ Sprint length: 1 week

⚐ Indicative milestone
⚑ Committed milestone



Stakeholders



Plan de haut niveau final

High level plan

Matti Schneider

August 14th, 2012

WATAI
Web Application Testing
& Automation Internship

Ambition

Write maintainable functional tests for web applications.

Objectives

- **Reduce QA costs.**
- **Assess quality.**
- **Assess regression risks.**

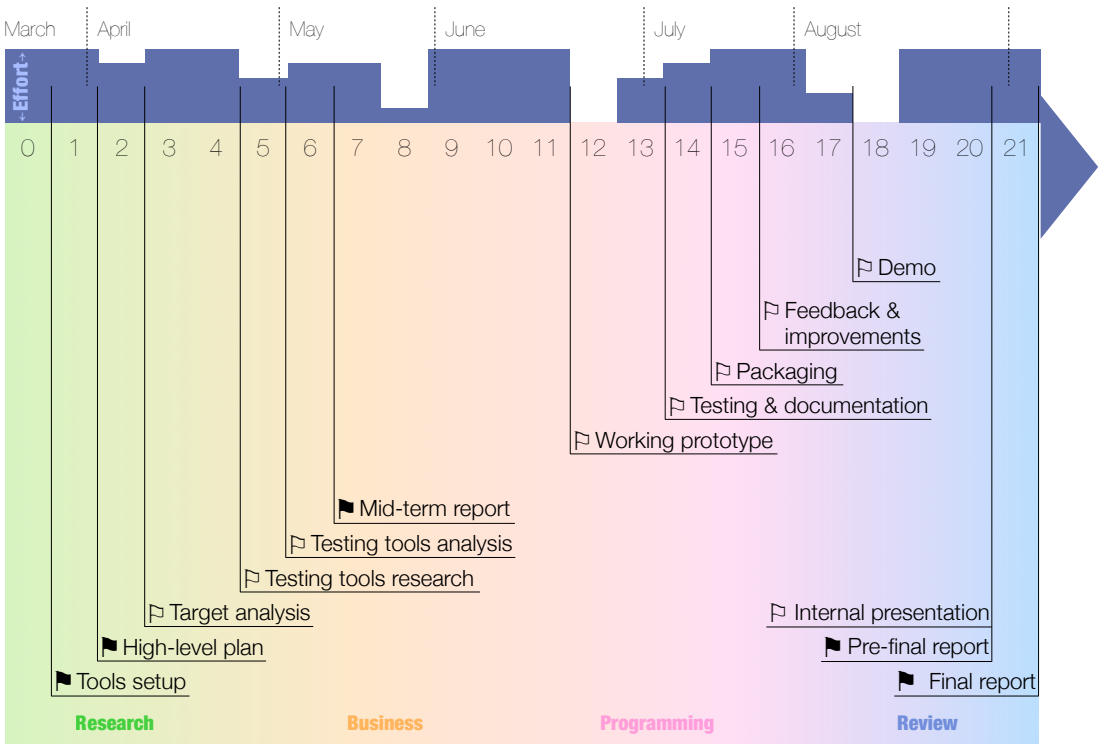
Metrics

- ➡ QA costs evolution.
- ➡ Amount of test-covered parts & tests results.
- ➡ Regressions frequency.

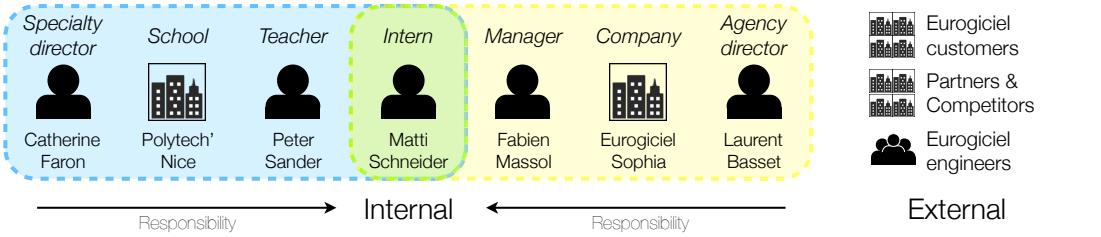
Roadmap

⚒ Methodology: Scrum
↻ Sprint length: 1 week

⚐ Indicative milestone
⚑ Committed milestone



Stakeholders



Scénarios de tests pour comparaison des outils de validation

checkmytrip.com

Precondition

1. Click on “OUTILS DE VOYAGE”.

Phone

1. Enter “France” in the “INDICATIFS TÉLÉPHONIQUES” field.
2. Check that the shown value is “00 33”

Clock

1. Enter “London” in the “Horloge” field.
2. Check that the updated value is one hour less than the previous one (assuming a French setup).

Account request

1. Enter “Toto” in “Prénom”.
2. Enter “Tutu” in “Nom”.
3. Click on “Inscription”.
4. Check that the Prénom and Nom fields are pre-filled with the corresponding previous input.

pdc.refedd.org

Login

1. Click on “Connexion”.
2. Login info: toto@toto.com / tototo .

Change name

1. Click on “Éditer” next to “Totok”.
2. Set name to “Totoka”.
3. Check that name is now “Totoka”.
4. Set name to “Totok”.
5. Check that name is now “Totok”.

Logout

1. Click on “Se déconnecter”.
2. Check that page is homepage.
3. Check that page exhibits login link.

Request a new password

1. Click on “Connexion”.
2. Click on “Mot de passe oublié ?”.
3. Set email address to “qsdfqsdf@qdsf.qsdf”.
4. Check that page says “Aucun compte avec l'adresse “qsdfqsdf@qdsf.qsdf””.

gsf-fr.net

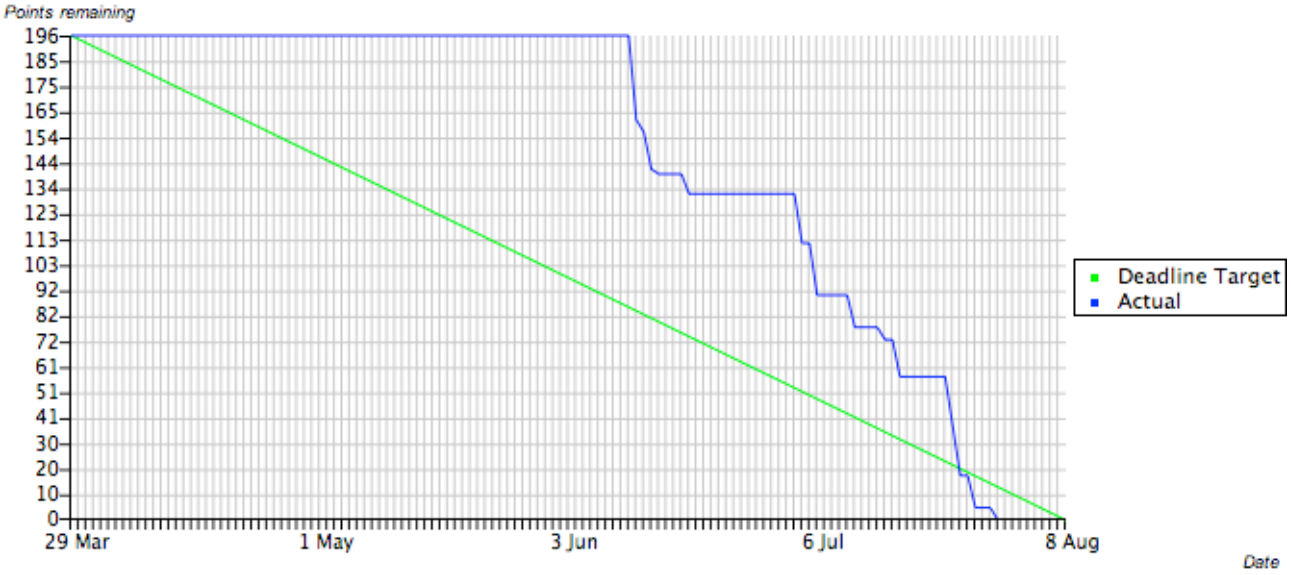
Basic navigation

1. Click on “Nous rencontrer”.
2. Check that final content contains “SOPHIA ANTIPOLIS”.

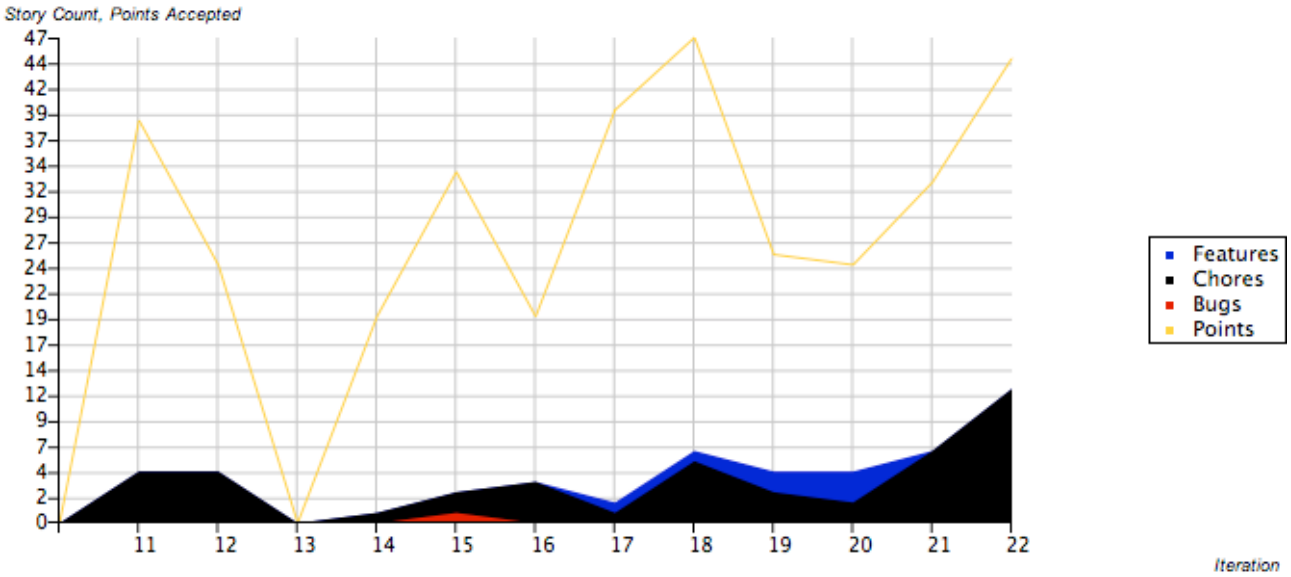
RSE

1. Click on “La démarche RSE de GSF”.
2. Click on “L'engagement social”.
3. Check that “Les enjeux sociaux identifiés :” is visible.

Graphes de suivi de projet



Burndown chart



Rapports entre anomalies, fonctionnalités et corvées (tâches n'apportant pas de valeur directe)

Abstract / Résumé

This document presents the author's final term internship in the product division of a software service company, Eurogiciel.

The goal of the internship was to devise a strategy for efficient automated testing of web applications and websites.

It led to research on the current state-of-the-art solutions and an analysis of their shortcomings. Then, specifications for an innovative solution were laid down. Those specifications were implemented in a software named Watai *(Web Application Testing Automation Infrastructure)*, which workings are detailed, along with current limitations and possible improvements. Both project management and technical methods are also thoroughly presented.

The end result is fully functional, and has a high level of software quality. The described software solution is available at this URL: https://github.com/MattiSG/Watai.

Ce document présente le stage de fin d'études de l'auteur au sein d'une division produit interne d'une société de services, Eurogiciel.

L'objectif de ce stage était le développement de compétences et d'outils dans le domaine de l'automatisation de validation d'applications et sites web.

Ce document présente un état de l'art concernant les solutions de validation automatique d'applications et sites web ; une analyse des limitations existantes et de ce qu'un nouvel outil devrait offrir ; une description de l'outil réalisé pour répondre à ces besoins, nommé Watai *(Web Application Testing Automation Infrastructure)* ; ses limitations ; et les méthodes, tant techniques que sociales, employées pour le mener à bien.

Le résultat final est pleinement fonctionnel, et est estimé à un haut niveau de qualité logicielle. Il est accessible à l'adresse : https://github.com/MattiSG/Watai.