Cucumber : Kesako ?

Un soft part toujours d’une idée, une idée qui doit être expliquée par le maitre d’ouvrage et comprise par une équipe de développement. Le monde du logiciel, aujourd’hui, souffre d’une mauvaise communication entre ces deux parties, entrainant un écart entre le concept de base et l’application développée.   
Nous avons donc besoin de filtres pour protégé le concept de l’application : c’est le premier rôle de Cucumber.

Plutôt que d’exposer ses exigences à l’équipe de développement, méthode qui ne favorise pas le feedback au cours du projet, les parties prenantes et les développeurs collaborent pour écrire ensemble une batterie de tests qui répondent aux besoins précis du maitre d’ouvrage. Cela permet donc de se centrer sur ce qui est le plus important pour lui, et qui doit être fait. On appelle ces tests « acceptance tests » car ils expriment ce que le soft a besoin de faire, dans l’ordre, pour être qualifié d’ « acceptable ».

Il faut bien faire la différence entre ces « acceptance tests » et les tests unitaires : ces derniers sont là pour aider le développer à vérifier et valider leur application tandis que nos tests d’acceptance sont là pour diriger le projet. On dit que le tests unitaires servent à « build the thing right » (faire les choses bien) tandis que les « acceptances tests » servent à « build the right thing » (faire les bonnes choses).

Le Behaviour Driven Development (BDD) dérive du TDD (Test Driven Development) en gardant les bonnes habitudes du développer TDD. Le but étant de créer des tests qui reflètent les attentes du maitre d’ouvrage, lisibles par n’importe qui. C’est alors qu’entre en jeu l’ « Ubiquitous language » (langage omniprésent).

Ce langage omniprésent part d’une idée : un projet trop souffre souvent d’un manque de communication entre les experts et le développement. Ceci est principalement dut à la barrière du langage : les experts expriment vaguement ce qu’ils veulent, dans leur jargon, et les développeurs, essayant souvent de comprendre le domaine pour lequel ils doivent développer, comprennent vaguement ce qu’ils doivent faire.

Avec Cucumber, l’ « Ubiquitous language », peut être compris par n’importe qui impliqué dans le projet, quel que soit son domaine. Quand vous l’utilisez dans vos conversations, vos documentations et votre code, la barrière du langage n’est plus et les risques d’incompréhension sont considérablement réduits.

Les tests Cucumber interagissent directement avec le code produit, mais sont écrit dans un langage intermédiaire, que le maitre d’ouvrage peut comprendre. Cette collaboration permet de mettre le doigt sur les incompréhensions de l’équipe de développement bien avant qu’elles ne viennent polluer le code.

Examples : Voici un exemple du langage utilisé par cucumber



Vous remarquerez à quel point ce langage décrit facilement ce que nous attendons du système. Cela permet donc, pour toutes les entités du projet, de visualiser le système avant qu’il ne soit construit. Ces tests d’acceptance deviennent alors des spécifications exécutables.

Ces tests servent également de documentation. Plutôt que d’écrire une doc qui devra être mise à jour continuellement dans le futur, ces tests d’acceptances deviennent une documentation vivante qui reflète réellement l’état du projet.

Les tests façon Cucumber permettent alors d’éviter de faire évoluer en même temps les documents, les tests et le code d’un projet classique. Les tests représentent alors une idée commune du projet, renforçant un esprit de confiance au sein de l’équipe, car personne n’a sa propre compréhension du soft.

Cucumber est un outil en ligne de commandes. Quand vous le lancez, il va lire dans vos spécifications chaque features qui contiennent les scénarios à tester. Chaque scénario contient une liste d’étapes à suivre. Donc, pour que Cucumber puisse comprendre vos scénarios, il faut respecter un certain nombre de règles syntaxiques. L’ensemble de ces règles est appelées Gherkin.

Cucumber transforme le code « humain » en Ruby pour exécuter les tests dans votre système.  
Un test peux générer seulement deux ou trois lignes de code Ruby qui seront ensuite délégués à une librairie de support, selon le domaine de votre application.  
Si le test se termine sans erreur, Cucumber se rend à l’étape suivante du scénario. Si un scénario se termine avec une erreur, il l’affiche en précisant ce qui n’a pas marché et passe ensuite au scénario suivant.

Les tests Cucumbers peuvent être écrits en 40 langages parlés différents. Vous pouvez également grouper vos scénarios en utilisant les tags.

