R4.02 - Qualité logicielle - CM n°3

Pierre Nerzic

février-mars 2024

Dans ce cours :

- Tests fonctionnels
  - Robot Framework et Selenium
- Intégration continue
  - Pipelines GitLab

- Bibliographie :
  - partie tests fonctionnels : horustest.io C'est une entreprise qui propose une solution clé en mains, HorusTest, pour définir des tests fonctionnels sur des logiciels Web. Leur blog, écrit par Stéphanie Binet, est remarquable.

Pierre Nerzic

Tests fonctionnels

Tests fonctionnels ?

Ce sont des tests du logiciel dans sa globalité, tel que le verra l'utilisateur.

Le but est de vérifier les fonctionnalités du logiciel par rapport aux spécifications, cahier des charges, cas d'utilisation...

Cela consiste en l'exécution de **scénarios d'utilisation** : par exemple pour un logiciel Web, ouvrir le logiciel par son URL, cliquer sur des boutons et des menus, remplir un formulaire, etc. Les tests simulent un utilisateur dans son usage du logiciel.

Les scénarios sont composés d'actions, chacune étant accompagnée de vérifications validant les exigences du client.

Buts des tests fonctionnels

Garantir la qualité du logiciel :

- vérifier que les exigences client sont réalisées,
- détecter des défauts au niveau de ce que verra utilisateur,
- éviter des régressions sur des fonctionnalités existantes,
- évaluer les performances,
- en retour, améliorer les phases du développement en tenant compte des priorités sur les besoins utilisateur.

Positionnement des tests fonctionnels parmi les tests

Les tests fonctionnels sont, par exemple, effectués par une sorte d'automate qui effectue des actions (clics, saisies, vérifications) sur un navigateur internet. Toutes ces actions ne sont pas instantanées : délais réseau et temps de calculs du navigateur pour mettre à jour l'interface.

Les tests fonctionnels sont donc très lents, comparés aux tests unitaires et d'intégration. Ils sont donc effectués après ces derniers, uniquement s'ils sont réussis.

Les tests fonctionnels sont difficiles à mettre en place dans un projet Agile, et encore plus dans l'approche *Test Driven Dev*, car les spécifications définitives ne sont pas disponibles.

Avantages des tests fonctionnels automatisés

Par rapport à des tests réalisés par des humains, les tests fonctionnels automatiques sont :

- plus rapides, en particulier parce qu'ils sont parallélisables,
- reproductibles,
- adaptables à différents jeux de données,
- planifiables au moment souhaité, par exemple la nuit.

Cercle vertueux : en augmentant le nombre de tests fonctionnels automatiques, cela augmente la couverture du code et des fonctionnalités, donc ça améliore la qualité du logiciel, et en conséquence, ça augmente aussi la confiance de l'équipe et du client.

Remarques sur les tests fonctionnels automatisés

- Il est souhaitable que le test fonctionnel automatique se comporte comme l'utilisateur final humain. Ça peut être compliqué, étant donné l'étendue des possibilités sur les interfaces actuelles : clics souris, raccourcis clavier, etc.
- Les tests automatisés font des contrôles à chaque étape : présence/visibilité de tel ou tel élément sur l'écran, mais il est difficile d'avoir la même intelligence qu'un humain pour analyser ce qui est affiché, par exemple les positions et tailles relatives des éléments, la vitesse d'affichage, etc.
- Il est quasiment impossible de tout prévoir, c'est à dire de programmer toutes les actions et vérifications possibles/nécessaires pour valider le cahier des charges. On doit se limiter à ce qui est le plus pertinent pour obtenir la qualité voulue.

Limitations des tests fonctionnels automatisés

- Un système automatique de tests ne peut pas avoir l'esprit inventif des humains pour essayer des choses imprévues, et trouver des bugs ($\Rightarrow$ les béta testeurs sont des humains).
- Les tests sont souvent complexes et difficiles à maintenir. Quand le logiciel évolue, il faut du temps pour adapter les tests.
- Les tests fonctionnels peuvent demander des ressources coûteuses et/ou complexes, ex : un serveur Web réel avec un SGBD. Certains fournisseurs d'accès internet offrent des services dématérialisés pour cela.
- Contrairement aux tests unitaires, les tests fonctionnels vérifient simultanément un grand nombre d'assertions. Il peut être très difficile de caractériser un bug lors d'un échec de test.

Construction des tests fonctionnels

Il existe des outils pour construire un scénario de test sans programmation. Cela consiste à enregistrer des actions manuellement, puis à les faire rejouer automatiquement, par exemple des *macros*.

Le problème est que ces scénarios sont généralement trop rigides, et n'incluent pas des vérifications à chaque étape.

La suite du cours présente un outil appelé **Robot Framework** qui ne demande que quelques notions de programmation. C'est parce que les tests fonctionnels sont généralement spécifiés par les experts fonctionnels et métier, comme le Chef de Produit, qui sont moins dans la technique que les développeurs/euses.

Robot Framework

Présentation de Robot Framework

À l'origine, en 2005, c'était la proposition d'un chercheur, Pekka Klärck (LinkedIn), d'exprimer des tests fonctionnels à l'aide de mots-clés. C'est à dire qu'au lieu d'écrire un programme, comme on le fait avec JUnit, on écrit des spécifications avec un langage basé sur des phrases simplifiées.

Voici un exemple :

```
*** Test Cases ***
Test Chercher Robot Framework sur Wikipedia
    Go To               https://www.wikipedia.fr/
    Input Text          id=search     Robot Framework
    Click Element       //img[@id="search-icon"]
    Wait Until Location Is  https://www.wikipedia.fr/wiki/Robot_
    Capture Page Screenshot
```


Pierre Nerzic

Principes généraux

Robot Framework se présente sous la forme d'une commande shell (programmée en Python), accompagnée de librairies pour gérer les navigateurs, les bases de données, etc.

La commande shell s'appelle `robot` et elle demande un nom de script de test en paramètre. Un script robot contient des directives indiquant des actions ou des vérifications à faire :

- ouvrir tel URL sur un navigateur
- vérifier que tel élément est présent
- cliquer sur tel élément
- vérifier que la base de données contient tel enregistrement,
- vérifier qu'on est maintenant sur tel URL
- etc.

Syntaxe générale des scripts robot

Un script de test doit avoir une mise en page très particulière. À première vue, ça ressemble à du texte mélangé à un fichier CSV, c'est à dire des chaînes séparées par des tabulations, et indentées comme en Python. Ces chaînes sont appelées « mots-clés » et peuvent avoir des paramètres.

```
NOM DU TEST
    MOT CLÉ 1    PARAM 1    PARAM 2
    MOT CLÉ 3    PARAM 4
    ...
```

Les majuscules/minuscules ne comptent pas mais il faut **impérativement** qu'il y ait au moins **deux espaces** entre deux mots-clés. Raison : les mots-clés peuvent être écrits avec plusieurs mots séparés par un seul espace.

Exemple

Voici un exemple de script, pour observer cette syntaxe :

```
*** Settings ***
Documentation   Test du tuto CodeIgniter
Library         SeleniumLibrary
Library         DatabaseLibrary

*** Test Cases ***
Endpoint test6 doit afficher la liste des produits
    Go To                       http://localhost:8000/test6
    Wait Until Page Contains    Voici la liste des produits :
    Page Should Contain Link    /test6/nouveau
    Capture Page Screenshot
```

`Go To`, `Wait Until Page Contains`... sont des « mots-clés ».

Pierre Nerzic

Sections d'un script

Un script robot est composé de plusieurs sections :

- initialisation générale : importation de librairies, commentaires

  ```
  *** Settings ***
  ```

- suite de tests

  ```
  *** Test Cases ***
  ```

- définition de mots-clés, de variables...

  ```
  *** Keywords ***
  *** Variables ***
  ```


Pierre Nerzic

Section Settings

Cette section sert à importer des librairies, c'est à dire des collections de mots-clés et variables permettant d'effectuer certaines tâches, et configurer les tests.

```
*** Settings ***
Library  SeleniumLibrary
Test Setup     Open Browser   http://localhost:8000/  firefox
Test Teardown  Close Browser
```

- `Library` fait charger une librairie de mots-clés et fonctions internes, voir transparent 26.
- `Test Setup` définit l'initialisation de chaque test quand elle est identique d'un test à l'autre (comme `@BeforeEach` de JUnit)
- `Test Setup` définit la clôture commune de chaque test (comme `@AfterEach`)

Section Test Cases

C'est là qu'on place les scénarios de test. Tous les tests doivent être présentés ainsi, indentés comme en Python :

```
*** Test Cases ***
Nom du test
    Mot clé     premier param     2e param      ...
    ...
```

Les paramètres dépendent des mots-clés. Voir transparent 26.

Il peut également y avoir des directives juste devant le mot-clé :

- `[Setup]`, `[Teardown]` voir transparent 24,
- `[Timeout]` *durée* spécifie un temps maximal, voir la doc,
- `[Documentation]` `texte` pour des informations.

Variante de syntaxe pour les tests

Robot Framework offre une autre syntaxe, permettant de mettre en valeur le patron AAA. Ce sont des mots-clés `Given`, `When`, `Then` et `And`. `Given` remplace *Arrange*, `When` remplace *Act*, et `Then` remplace *Assert*.

```
Test Wikipedia's Robot Framework page
    Given Open Browser  https://www.wikipedia.fr/  firefox
    When Input Text     id=search     Robot Framework
    And Click Element   //img[@id="search-icon"]
    Then Page Should Contain    est un framework de test
```

Le mot-clé `And` permet de continuer comme le précédent mot-clé.

Section Variables

Cette section permet de définir des variables qui seront accessibles dans tout le script. Il y a trois syntaxes :

- `${NOM}` pour une variable contenant une simple chaîne.
- `@{NOM}` pour une variable contenant une liste. Les éléments doivent être séparés d'au moins deux espaces.
- `&{NOM}` pour un dictionnaire de paires clé-valeur.

Voici un exemple :

```
*** Variables ***
${URL} =      http://localhost:8000/
@{WINSIZE} =  1280   1024
&{TEST1} =    user=uti1   pass=uti1pw   vrainom=Pierre
```

Le signe = après le nom de la variable est facultatif.

Utilisation des variables

- Pour les variables simples, il suffit d'écrire `${NOM}`
- Pour une liste, `@{NOM}` vaut toute la liste, et la notation `@{NOM}[ind]` fonctionne exactement comme en Python.
- Pour un dictionnaire, `&{NOM.clé}` et `&{NOM}[clé]` retournent la valeur associée.

```
*** Test Cases ***
Check login
    Open Browser        ${URL}/main.html    headlessfirefox
    Set Window Size     @{WINSIZE}
    Input Text          //input[@name="user"]   &{TEST1.user}
    Input Text          //input[@name="pass"]   &{TEST1.pass}
    Click Button        id=Login
    Wait Until Page Contains    Bonjour &{TEST1}[vrainom]
```


Pierre Nerzic

Section Keywords

Pour définir de nouveaux mots-clés utilisables dans les tests :

```
*** Keywords ***
Start Services
    ${PHP} =  Start Process   /usr/bin/php  -S  ${HOST}:${PORT}
    Set Suite Variable  ${PHP}

Stop Services
    Close Browser
    Terminate Process   ${PHP}
```

Le mot-clé `Set Suite Variable` rend la variable globale dans la suite de tests. On peut alors l'employer dans l'autre mot-clé.

Les mots-clés `Start Services` et `Stop Services` pourront être employé dans les tests, ou bien au niveau de la suite de tests.

Définition d'une suite de tests

Une suite de test est un ensemble de tests qui partagent la même initialisation/terminaison. Il est préférable de :

- Définir l'initialisation et la terminaison en tant que mots-clés, comme dans le transparent précédent
- Indiquer que le démarrage et l'arrêt de l'ensemble des tests se font avec ces mots-clés :

```
*** Settings ***
Suite Setup     Start Services
Suite Teardown  Stop Services
```

- `Suite Setup` est comme `@BeforeAll` de JUnit, une initialisation commune exécutée une seule fois, au tout début,
- `Suite Teardown` est comme `@AfterAll` faite une fois à la fin.

Démarrage ou terminaison spécifique

S'il y a un mot-clé spécifique pour le démarrage ou l'arrêt d'un test, on le fait précéder par :

- `[Setup]` pour un mot-clé qui initialise le test,
- `[Teardown]` pour ce qui termine le test même s'il a échoué, et s'il y a plusieurs `[Teardown]`, ils sont tous faits, même si certains échouent.

Exemple :

```
*** Test Cases ***
Test de la page d'accueil de Google
    [Setup]     Log   test de base sur la page google.fr
    Open Browser      https://www.google.fr
    Title Should Be   Google
    [Teardown]  Close Browser
```

Librairies

Il y a un grand nombre de mots clés pour toutes sortes d'interactions avec le navigateur : actions sur la page Web et vérifications sur le contenu.

Ces mots-clés sont définis dans des bibliothèques (*library* en anglais).

On inclut celles dont on a besoin dans la section `Settings`.

Exemple :

```
*** Settings ***
Library   Collections
Library   SeleniumLibrary
Library   JSONLibrary
```

Librairie `SeleniumLibrary`

Cette librairie sert à interagir avec un navigateur. Sa documentation est sur cette page. Elle définit de très nombreux mots-clés, 173 au total, pour :

- contrôler le navigateur : Open Browser, Close Browser. En général, on utilise un navigateur sans interface graphique visible, par exemple `headlessfirefox`.
- naviguer sur des URLs : Go To, Go Back, Wait Until...
- vérifier le contenu affiché : Page Should Contain..., Element Should..., Get... avec de très nombreuses variantes
- cliquer sur un élément : Click..., Double Click Element, Submit Form
- saisir des valeurs dans des *input* : Input...
- cocher des checkbox et radio : Select...

Désignation des éléments

Un point important est la désignation de l'élément du DOM HTML qu'on souhaite manipuler, par exemple un `<button>` pour cliquer dessus. Selenium appelle ça un *locator* (documentation).

```
Click Element    id=btn-submit
```

Il y a deux syntaxes :

- le mode implicite (par défaut) qui dépend du mot-clé, par exemple pour le mot-clé `Click Button`, Selenium recherche l'attribut `id`, `name` ou `value` qui correspond au paramètre.
- le mode explicite signalé par un préfixe :
  - `id:truc`, `name:truc`, `class:truc`
  - `css:selecteur` désigne l'élément qui possède ce sélecteur
  - `xpath:chemin` désigne l'élément qui correspond au chemin. Ce mode est aussi implicite, car le préfixe `xpath:` est optionnel.

Librairie `DatabaseLibrary`

Elle sert à interagir avec un SGBD. Sa documentation est sur cette page. Elle définit 15 mots clés pour :

- gérer une connexion `Connect To Database` *paramètres*, `Disconnect From Database`
- exécuter un script SQL `Execute Sql Script` *nomscript* ce qui permet de remplir une base avec des valeurs prédéfinies
- vérifier la présence d'une table `Table Must Exist` *nomtable*
- compter les n-uplets sélectionnés par une requête `Row Count Is Equal To X` *select nb*
- supprimer des n-uplets `Delete All Rows From Table` *nomtable*

Librairie `Process`

Elle sert à lancer des processus de type « démon », par exemple un serveur Web. Sa documentation est sur cette page.

Elle définit 15 mots-clés, dont ceux-ci :

- lancer un processus Run Process qui attend la fin du processus, et Start Process qui le lance en arrière-plan. Ce mot-clé retourne le PID du processus, à mettre dans une variable,
- arrêter un processus, Terminate Process à qui on fournit le PID,
- vérifier l'état d'un processus Is Process Running, Process Should Be Running.

Exécution des tests

Les scripts doivent porter l'extension `.robot`

- exécuter un seul script

  ```
  robot script.robot
  ```

- exécuter tous les scripts d'un dossier

  ```
  robot dossier
  ```

La commande crée plusieurs fichiers dans un dossier « `reports` » :

- un bilan dans un fichier `report.html`
- des compte-rendus `output.xml` et des `*.log`
- des copies d'écran par le mot-clé `Capture Page/Element Screenshot`

Pierre Nerzic

Intégration continue et livraison continue

Intégration continue ?

Cela consiste à systématiquement appliquer les suites de tests et de couverture lors de chaque enregistrement (*commit*) du code source.

Dans une équipe comprenant plusieurs développeurs/euses, chacun/e travaille sur des parties du projet. L'intégration consiste à regrouper les différents travaux et faire en sorte qu'ils soient compatibles et qu'ils passent les tests.

Le logiciel Git permet de travailler relativement séparément puis de regrouper les contributions. Les logiciels Web GitHub et GitLab possèdent un dispositif, « CI/CD » (*Continuous Integration/Continuous Deployment*), pour déclencher automatiquement les suites de tests à chaque *commit+push*.

Déroulement de l'intégration continue (CI)

- Il faut avoir un système de dépôt de code source avec versions et qui gère l'intégration continue.
- On y définit ce qui s'appelle un **pipeline de test**, une succession d'étapes devant réussir :
  1. compilation du code source,
  2. application de tous les tests unitaires, d'intégration et fonctionnels,
  3. mesure de la qualité
  4. génération des livrables pour les clients (installateur Windows, paquet APK ou Debian, archive, etc.).

Chaque étape peut échouer. Une sorte de tableau de bord permet d'informer les utilisateurs.

Livraison et déploiement continus ?

C'est la suite de l'intégration continue. La livraison continue concerne l'exécution du logiciel dans un environnement réaliste, par exemple sur des machines virtuelles, similaires aux machines des clients, afin de faire des tests fonctionnels en situation et des mesures de performance.

Par exemple, on vérifie que le logiciel s'installe bien et tourne bien sur toutes les versions de Windows, avec des PC plus ou moins performants, etc.

La livraison continue débouche sur le déploiement continu : la mise à disposition du logiciel pour les clients.

Ce cours ne pourra pas traiter ces aspects, faute de temps.

Intégration continue dans GitLab

En TP, nous utiliserons GitLab qui intègre tous les outils utiles : git et un ordonnanceur d'intégration continue.

NB: nous allons utiliser seulement un tout petit sous-ensemble des possibilités. C'est parce qu'il n'est pas possible de vous rendre administrateur des projets. Vous serez obligés de rester dans un tout petit cadre, mais qui permet quand même de se faire une idée générale.

Pour l'intégration continue (CI/CD), le principe est de définir un *pipeline* à l'aide d'un script dédié.

Définition d'un pipeline

Créer un fichier appelé `.gitlab-ci.yml` à la racine du projet et contenant par exemple ceci :

```
RobotFramework tests:
  tags:
    - robotframework
  stage: test
  variables:
    OUTDIR: /usr/local/reports
  script:
    - mkdir -p $OUTDIR
    - robot --outputdir $OUTDIR Tests
```

Ce script fait passer les tests Robot Framework à votre projet. Ces tests sont présents dans un dossier appelé `Tests` à la racine du projet. Les résultats vont dans `/usr/local/reports`.

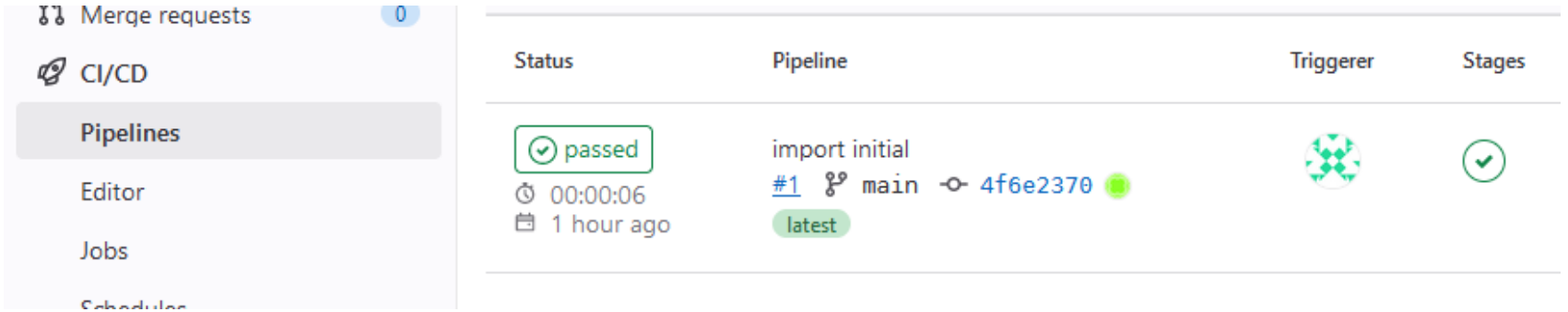
Exécution d'un pipeline

Voici ce qu'on voit quand un job s'exécute, le même résultat qu'avec Robot Framework :

```
35 ==============================================================================
36 Tests.TestMainHTML
37 ==============================================================================
38 Check main.html                                                      | PASS |
39 ------------------------------------------------------------------------------
40 Tests.TestMainHTML                                                  | PASS |
41 1 test, 1 passed, 0 failed
42 ==============================================================================
43 Tests                                                               | PASS |
44 3 tests, 3 passed, 0 failed
45 ==============================================================================
46 Output:  /usr/local/reports/root/projet1/4f6e2370/output.xml
47 Log:     /usr/local/reports/root/projet1/4f6e2370/log.html
48 Report:  /usr/local/reports/root/projet1/4f6e2370/report.html
50 Job succeeded
```

Tableau de bord des pipelines

Voici une partie de l'interface montrant un pipeline exécuté avec succès :



L'exécution du pipeline a été déclenché par un *commit* appelé « import initial ».

Nous verrons en TP ce qui se passe quand un pipeline échoue, et comment les développeurs sont invités à corriger le problème.

Fin

C'est tout pour aujourd'hui. Rendez-vous en TP pour la mise en pratique.

Pierre Nerzic