Noyau AMDA-NG - 2nde partie

PLAN DE VALIDATION DU LOGICIEL

Référence : CDPP-PE-32500-460-CS

Etat Préliminaire

Version 1.1 du 13/01/2014

afaq ISO 9001 Qualité AFNOR CERTIFICATION

CS SYSTEMES D'INFORMATION

22, avenue Galilée – 92350 le Plessis Robinson - Tél. : 01 41 28 40 00 – Fax : 01 41 28 40 40
w w w . c - s . f r

Tableau des signatures

	Nom	Fonction	Date	Signature
Préparé par :	N.Boursier		13/01/2014	
Validé par :	S.Frayssines		13/01/2014	
Approuvé par :	R.Patrier		13/01/2014	

Sommaire

Table des illustrations

Liste des tableaux

Liste des figures

Liste des éléments à définir (AD)

Glossaire

Abréviation	Définition
AMDA	Automated Multiple Dataset Analysis
AMDA-NG	Automated Multiiple Dataset Analysis - New Generation
CNES	Centre National d'Études Spatiales
CSSI	Communication et Systèmes – Systèmes d'Information.
IRAP	Institut de Recherche en Astrophysique et Planétologie

1. INTRODUCTION

L'objet du plan de validation est de définir l'organisation et la démarche de gestion associées à la réalisation des activités de validation des nouveaux développements du noyau AMDA-NG. Il présente ainsi l'ensemble des tests et moyens à mettre en œuvre pour réaliser la validation.

- ✓ Le Chapitre **4. Planification du processus de validation logiciel** présente la validation en termes de ressources, d'organisation et d'outils.
- ✓ Le Chapitre **5. Identification des tâches de validation du logiciel** présente ensuite les différents types de tests à réaliser pour assurer la validation de l'ensemble des nouvelles fonctionnalités.
- ✓ Enfin, le Chapitre **6. Démarche de validation** résume la demarche de validation identifiée dans la plan de développement

Ce document présente également, en Annexe 0 l'environnement utilisé pour l'automatisation des tests d'acceptation et détaille notamment la procédure à suivre pour ajouter un nouveau test.

2. DOCUMENTS APPLICABLES

DA01 Consultation AC-IS N°DAJ/AR/EO-2013.08449 "Développement du noyau AMDA-NG - Seconde Partie". CDPP-CO-32500-452-CNES, 06/05/2013, Ed. 01 Rev 00.

DA02 Spécification de besoins techniques pour la seconde prestation du nouveau noyau AMDA. CDPP-ST-32500-451-CES, 13/05/2013, Ed. 01 Rev 00

DA03 Cahier des Clauses Techniques Particulières AC-IS. DCT/PS-2011-003173.

DA04 Exigence de réponse aux clauses de sécurité des Systèmes d'information de l'Accord Cadre AC-IS. DCT/PS-2010-15734

DA05 Projet de cahier des prescriptions de Sécurité des Systèmes d'Information Accord Cadre Informatique Spatiale Clauses Générique. DCT/PS-2011-003191

DA06 Exigences Normatives associées aux prestations de développement et de maintenance dans le domaine de l'informatique spatiales. ACIS-ACIBS-SP-GEN-1-CNES

3. DOCUMENTS DE REFERENCE

DR01 Dossier d'architecture du noyau d'AMDA-NG. CDPP-AR-32500-382-SI, Ed. 02 Rev. 01, 29/11/2012.

DR02 Dossier de conception du noyau AMDA-NG. CDPP-CD-32500-436-SI, Ed. 01 Rev. 06, 11/02/2013.

DR03 Dossier de contrôle des interfaces du noyau AMDA-NG. CDPP-IF-32500-438-SI, Ed. 01 Rev. 04, 05/02/2013.

DR04 Étude sur les solutions alternatives à IDL. CDPP-NT-32500-383-SI, Ed. 01 Rev. 02, 10/01/2010.

DR05 Manuel d'installation de AMDA Kernel. CDPP-MI-32500-440-SI, Ed. 01 Rev 05, 11/02/2013.

4. PLANIFICATION DU PROCESSUS DE VALIDATION LOGICIEL

4.1Organisation

Dans le cadre d'une méthodologie AGILE, il n'existe pas une unique phase de validation telle qu'elle existe dans un cycle en V traditionnel mais un ensemble de tâches de validation réalisées tout au long et en parallèle des tâches de développement.
L'objectif de ces tâches de validation est de vérifier l'adéquation du fonctionnement effectif de l'applicatif aux besoins exprimés à travers les *User Stories* définies dans le *backlog*. Ceci afin de soulever les éventuels écarts et dysfonctionnement puis d'identifier leur correction.

4.2Planning

Des tâches de validations apparaissent tout au long d'un sprint : il existe un *test d'acceptance* pour chacune des *User Stories* d'un sprint. En conséquence, il n'y a pas de planning prédéfini pour ces tâches de validation.

Cela étant, la validation d'une release est formalisée, en fin de release, par la tenue d'un cycle de recette terminé par un point clé *Acceptance Review* (AR).

4.3Ressources

La validation s'effectue :
- ✓ manuellement, sur le serveur de développement
- ✓ automatiquement, sur le serveur d'intégration continue

Ces machines disposent toutes d'un os CentOs 6.3, 64bits.

La validation requiert un accès au serveur DDServer sur le réseau IRAP depuis les serveurs de développement (tests manuels) et d'intégration continue (tests automatisés).

4.4Responsabilités

Les tests unitaires de validation sont mis en œuvre par l'équipe de développement, exécutés manuellement par cette même équipe et exécutés de manière automatisée par l'intégration continue.

Les essais de validation fonctionnels ou *Acceptance tests* sont rédigés sous forme textuelle libre par le Product Owner (IRAP) et pour chacune des User Stories. Les entrées nécessaires pour le passage des tests sont fournies avec la story. Les résultats attendus sont a minima décrits et fournis si possible s'il s'agit de fichiers.
Les tests sont ensuite formalisés pour être automatisés par l'équipe de développement et exécutés :
- ✓ manuellement par cette même équipe,
- ✓ automatiquement via les mécanismes d'intégration continue
- ✓ et, s'il le souhaite, manuellement par le Product Owner

Les résultats obtenus par l'exécution des tests automatiques sont compilés par le Scrum Master et présentés au Product Owner. Ce dernier, assisté éventuellement des Stakeholders, valide ou non ces résultats qui deviennent, le cas échéant, les résultats de référence pour la non-régression.

Le tableau ci-dessous présente une synthèse des rôles et activités de validation affectées.

Légende :
✓ responsable de l'activité
(✓) participation éventuelle à l'activité

	Product Owner	Scrum Master	Equipe	Resp. Qualité
Rédaction des tests d'acceptation (*acceptance tests*)	✓			
Fourniture des entrées du test	✓			
Fourniture (si possible) des résultats attendus	✓			
Mise en place des tests			✓	
Passage et/ou suivi des tests	(✓)	✓	✓	
1ère validation des résultats			✓	
Approbation des résultats et acceptation des résultats en tant que références pour les tests de non régression	✓			
Contrôle des métriques		✓		(✓)

Tableau 1 : Synthèse des activités et des intervenants

4.5Méthode et outils utilisés

Les tests sont décrits pour chaque story dans l'outil Icescrum. L'association fonctionnalité/test est ainsi immédiate.
Lorsque les résultats du processus de validation (manuel ou automatique) ne sont pas conformes aux résultats attendus, le suivi des corrections à réaliser est également effectué avec Icescrum par l'intermédiaire de tâches additionnelles associées à la story en défaut (validation interne) ou par la création de stories techniques relatives (validation de fin de sprint, correction à traiter dans l'un des sprints suivants).
Le tableau ci-dessous liste les outils mis en œuvre pour la validation du logiciel :

Besoin	Outil
Tests unitaires	CppUnit
Description des tests	Icescrum (acceptance test)
Mise en œuvre des tests	FitNesse + FitCpp
Gestion des contextes de données	SVN
Gestion des anomalies internes	Icescrum (tâche)

Tableau 2 : Outils mis en œuvre pour la validation

4.5.1 CppUnit

CppUnit est un outil de test unitaire pour les logiciels développés avec le langage c++.

4.5.2 FitNesse + FitCpp

FitNesse est un outil d'automatisation de tests d'acceptation. Les tests *y* sont décrits à l'aide d'un formalisme simple et lisible (spécification des entrées et des sorties attendues) puis exécutés par un moteur de test. Le résultat des tests est remonté à l'utilisateur.
FitCpp est un module complémentaire à FitNesse permettant d'exécuter des tests sur du code c++.

5. IDENTIFICATION DES TACHES DE VALIDATION DU LOGICIEL

Comme précisé dans le plan de développement ([PDV]), pour assurer la validation de l'applicatif, trois types de tests sont déroulés :

- ✓ **Tests unitaires** qui valident l'ensemble des chemins possibles du code, cas d'erreurs inclus,
- ✓ **Tests fonctionnels** qui ont pour finalité de prouver l'adéquation de l'application aux besoins de l'IRAP et de l'utilisateur définis à travers les users stories, ces tests sont automatisés et exploités pour vérifier la non-régression,
- ✓ **Tests de performances** pour valider que, dans des conditions normales de fonctionnement, les fonctionnalités de tracés respectent des contraintes de temps de traitement AD 1.

L'organisation des différents types de test dans le cadre du niveau d'exigence 'ES' est la suivante :

Catégorie de test	Tests unitaires	Tests fonctionnels	Tests de performance
Expression du besoin	CSSI	IRAP	IRAP/CSSI
Responsable mise en place	CSSI	CSSI	CSSI
Responsable approbation	CSSI	IRAP	IRAP/CSSI
But	Tester au plus tôt les fonctions de bases du logiciel. Compléter la couverture fonctionnelle.	Couvrir le domaine fonctionnel de la story	Estimer les performances des requêtes.
Outillage	CppUnit	FitNesse	CppUnit
Critère de succès	Résultats attendus et obtenus par CppUnit	Comparaison avec références ou analyse manuelle.	Seuil défini au cas par cas
Suivi	Fichier log produit par CppUnit	Rapport FitNesse. Cahier de recette.	Fichier log produit par CppUnit. Cahier de recette, Dossier de justification.

Tableau 3 : Stratégie de validation

Parallèlement aux tâches de validation, le contrôle du code source est réalisé par deux moyens :

- ✓ Par les outils de l'atelier de développement (cppCheclipse)
- ✓ Par les outils d'intégration continue (sonar, cppCheck)

6. DEMARCHE DE VALIDATION

La démarche de validation est décrite dans le Plan de Développement [DV], § 4.3.4. Elle peut être résumée ainsi :

- ✓ Une définition du « prêt » qui inclut la description des tests d'acceptation et en particulier la disponibilité des entrées nécessaires à la fabrication des résultats de référence
- ✓ Une définition du « fini » permettant de clore une story avec codage et validation
- ✓ Une utilisation systématique des moyens automatiques mis à disposition de l'équipe de développement (intégration continue, production des métriques, contrôle de code, …)
- ✓ Une validation croisée mise en œuvre le plus systématiquement possible

La validation d'une story est représentée par une tâche dédiée dans le backlog du sprint.

La non-regression est vérifiée quotidiennement via l'exécution des outils d'intégration continue et la compilation des résultats par le Scrum Master. Les résultats des tests validés par le Product Owner font office de résultat de référence et sont ajoutés à la non-regression. La base de tests établie lors du premier développement de noyau AMDA-NG est exploitée dans le but d'assurer une non-régression amont.

6.1Définition du prêt

Les critères de définition du « prêt » sont présentés dans le Plan de Développement [DV] § 4.3.3.2.

6.2Définition du fini

Les critères de définition du « fini » sont présentés dans le Plan de Développement [DV] § 4.3.3.3.

A Annexe : FitNesse

Cette annexe présente le principe global de fonctionnement de l'outil FitNesse couplé à CSlim dans le cadre de la validation du noyau AMDA-NG. Elle détaille par la suite la procédure de création d'un nouveau test d'acceptation à dérouler via FitNesse.

A.1 Principe de fonctionnement

Dans FitNesse il existe deux types de système de test :

- ✓ Fit (Framework for Integrated Testing) qui actuellement n'est quasiment plus mis à jour du fait de sa lourdeur dans la création d'un test,
- ✓ Slim (Simple List InvocationMethod) qui est préféré par sa simplicité d'utilisation et de mise en oeuvre.

La raison pour laquelle Slim est préféré réside dans sa souplesse face aux différents langages qui existent sur le marché. Slim est un simple protocole dont la communication est basée sur une socket. Ainsi, quelque soit le langage, la communication entre FitNesse et le projet à tester sera toujours possible.

A partir du moment où l'implementation du protocole est présente dans un langage, son utilisation est triviale. Pour pouvoir lancer un test d'acceptance, seulement deux étapes sont à fournir par le développeur :

- ✓ la création d'un serveur qui permettra de récupérer et traiter la liste des instructions envoyées par FitNesse,
- ✓ la création d'un fichier « Fixtures » rescensant la liste des fichiers utilisés pour lancer les tests.

Chaque fichier définit un certain nombre de fonctions servant de point d'entrée pour lancer un test (il vise une partie spécifique du système à tester).

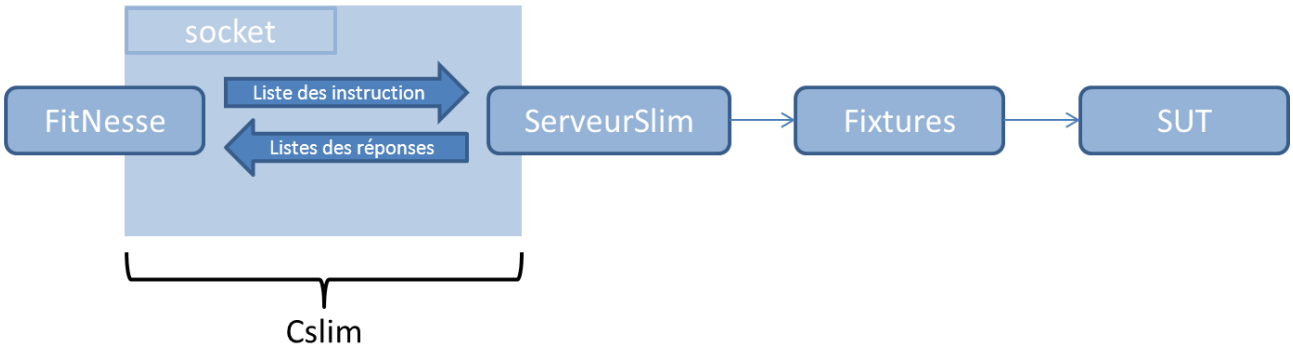

Figure 1 – Chaîne de fonctionnement d'un test

CSlim intervient dans la communication entre FitNesse et le « serveur slim » et particulièrement pour l'interfaçage avec un projet écrit en C/C++. C'est lui qui implémente le protocole Slim et donc la socket utile pour la communication.

Ainsi, FitNesse va exécuter le « serveur slim » et lui envoyer la liste des instruction à exécuter. Une fois le serveur executé, chaque instruction est parsé et mappé à une fonction (le fichier « Fixtures » est analysé afin d'établir la correspondance). Une fois le lien entre l'instruction et la fonction effectué, le resultat et renvoyé au serveur qui lui-même le renvoy à FitNesse.

A.2 Creation d'un test d'acceptance

A.2.1 Fichier « SUT »

La première étape pour créer un test d'acceptance est avant tout de définir la partie du système qui est à tester dans un fichier. Dans le cadre du C++, il s'agit d'implémenter une classe en utilisant différents artifices pour la lier au « serveur slim ».

Il est à noter que CSlim utilise le langage C et non C++. De ce fait, pour que les méthodes définies dans notre classe soient accessibles depuis un code c il faut rajouter ceci :

```
extern "C" {
        // Définition des fonctions de test
        static char* nomMethode_1(void *fixture, SlimList *args) {
        }

        static char* nomMethode_2(void *fixture, SlimList *args) {
        }
}
```

Le type de retour dans ces méthodes est très important car CSlim renvoie à chaque fois une chaine de caractère comme réponse à FitNesse.
Le premier paramètre des fonctions représente la classe elle-même (nécessite un cast), et le second paramètre contient une liste d'attribut (nous verrons plus tard le lien avec ces attributs).

Maintenant que nous avons définit la partie du système à tester, il ne reste plus qu'à déclarer ces fonctions auprès de CSlim en les repertoriant dans une « Fixture »
Pour créer une fixture, généralement en fin de classe il va être écrit ceci :

```
SLIM_CREATE_FIXTURE(nomDeLaClasse)
        SLIM_FUNCTION(nomMethode_1)
        SLIM_FUNCTION(nomMethode_2)
SLIM_END
```

La déclaration de la fixture est très explicite, une fixture est créé portant le nom de la classe qui contient les fonctions de test puis à l'intérieur de la fixture on déclare les fonctions à utiliser.

En bref voici à quoi peut ressembler le squelette d'un fichier test :

```
#include "CSlim/SlimUtil.h"
#include "CSlim/SlimList.h"
#include "CSlim/Fixtures.h"
#include "CSlimTools.hh"

NomClasse:: NomClasse () {
        // Liste d'initialisation
}

NomClasse::~ NomClasse () {
}
extern "C" {
        // Définition des fonctions de test
        static char* nomMethode_1(void *fixture, SlimList *args) {
        }

        static char* nomMethode_2(void *fixture, SlimList *args) {
        }

        // Déclaration de la fixture et des méthodes qu'elle contient
        SLIM_CREATE_FIXTURE(NomClasse)
                SLIM_FUNCTION(nomMethode_1)
                SLIM_FUNCTION(nomMethode_2)
        SLIM_END
}
```

A.2.2 Recensement

Une fois que la classe de test est créée, il ne reste plus qu'à la rendre accessible depuis le « serveur slim ». Pour ce faire, il suffit de rajouter le nom de la classe tests dans le fichier « Fixtures.c ». Ainsi on aura :

```
#include "CSlim/Fixtures.h"

SLIM_FIXTURES
        SLIM_FIXTURE(NomClasse)
SLIM_END
```

Chaque nouvelle classe de test sera donc rajoutée dans ce fichier pour qu'elle soit accessible depuis le « serveur slim ».

Le lien entre le serveur et ce fichier est implicite. On ne fait aucune déclaration de ce fichier, le lien sera fait lors de la génération de l'exécutable du serveur en spécifiant le répertoire à inclure qui contient le fichier.

A.2.3 Page de test

Maintenant que les tests ont été définis, il ne reste plus qu'à générer une page de test dans laquelle sera contenu l'ordre d'appel des fonctions ainsi que la valeur des paramètres (s'il y en a).

Voici un petit exemple de page de test :

```
!1 User Story 31: Connexion à DD_Server

!define path {../test/FitNesseRoot/ReleaseS/FirstPart/ReLease1/SprinT1/}
!2 Description du test
Le but du test est de vérifier que le code !-SimpleTest-! de DD_client fonctionne correctement.
Nous testons l'existence de la variable xxx avec !-SimpleTest-!
Et enfin nous vérifions le retour (stdout) de la commande avec un diff par rapport a un document de ref

!2 Test

!|script|TestAmdaCommandLine|
|set|${path}|path|


!|script|TestCommandLine|
|note|!-Exécution du programme SimpleTest-!|
|check|execute|SimpleTest|command|cmd1|alias|1|
|note|La sortie standard de la commande 1 doit contenir 25 lignes|
|check|execute|!-test `wc -l ${cmd1:OUTPUT} | awk '{print $1}'` -eq 25-!|command|count1|alias|0|
|note|Afficher le temps mis par cmd1|
|check|execute|!-grep "Time elapsed: msecs" ${cmd1:OUTPUT} | sed "s/Time elapsed: msecs\(.*\)/\1/"-!|command|GetTime1|alias|0|
|show|read|!-${GetTime1:OUTPUT}-!|file|
---
!contents -R2 -g -p -f -h
```

Les premières lignes de cette page décrivent le test qui va être déroulé avec une mise en page. Ce n'est seulement qu'à partir de la ligne « !|script|TestAmdaCommandLine| » que les tests commencent à être déroulés. Celle-ci exige que le script « TestAmdaCommandLine » soit exécuté. La ligne qui suit immédiatement demande à appeler la fonction « setPath » avec pour argument la valeur qui est définie dans la ligne « !define path {**../test/FitNesseRoot/ReleaseS/FirstPart/ReLease1/SprinT1/**} ».

Lorsque les lignes commencent par le mot-clé « note » cela veut dire que ce n'est pas du code à exécuter. Cela peut servir dans le cas où une erreur est retournée afin d'ajouter des informations sur ce qui est testé.

La ligne d'instruction « |check|execute|SimpleTest|command|cmd1|alias|1| » indique qu'il faut vérifier la valeur retournée par la fonction « executeCommandAlias » qui doit être 1.

Enfin quand une ligne d'instruction commence par le mot clé « show » par exemple « |show|read|!-${GetTime1:OUTPUT}-!|file| », cela signifie que l'on veut afficher le résultat retourné par la fonction « readFile ».

En général, la pratique est de mettre un argument entre chaque mot qui compose une fonction. Il existe une autre manière de faire décrite ici (http://fitnesse.org/FitNesse.UserGuide.SliM.ScriptTable).

Voici le rendu que produit cette page :

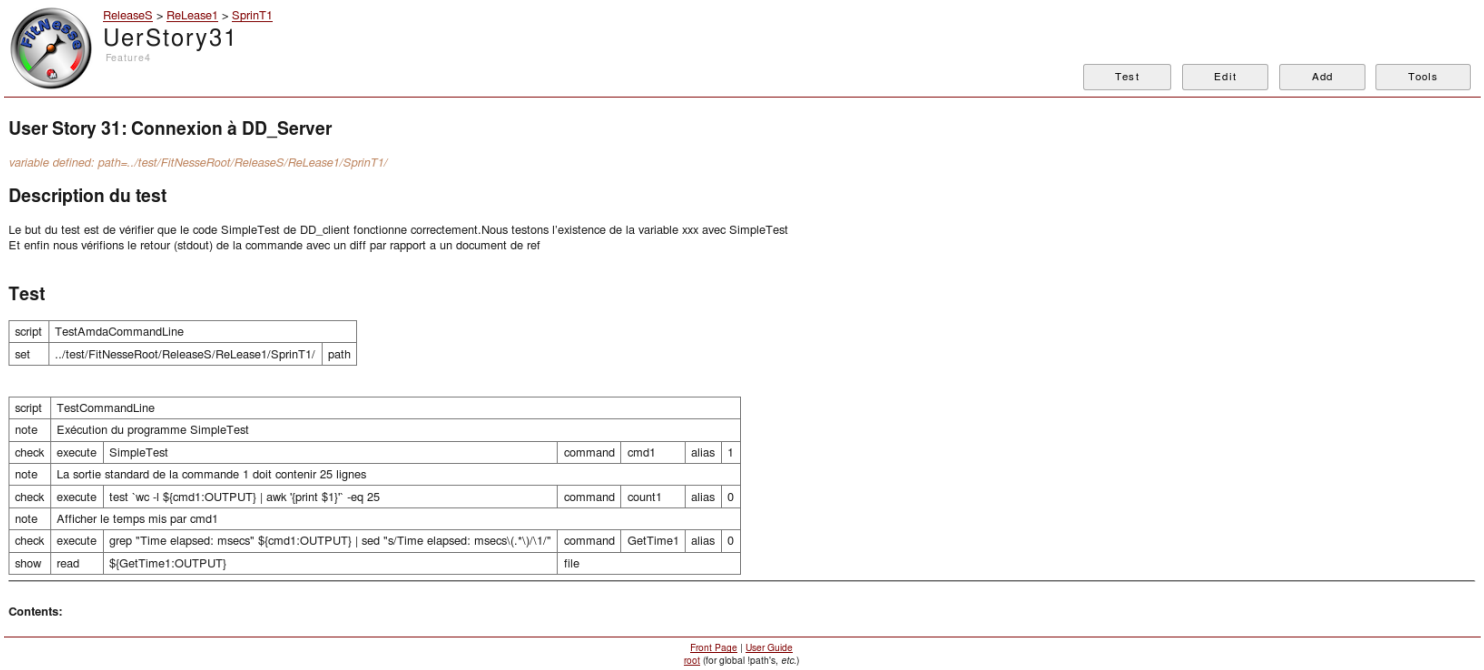

Figure 2 – Exemple de rendu d'une page de déruolement d'un test

Ce style de page ne peut lancer qu'un seul test à la fois. Pour remédier à cela, il suffit de changer la propriété de la page pour la passer de simple page de test à une page de série de test. Pour cela il suffit d'aller dans « Tools/Properties » et de sélectionner « Suite » dans la section « PageType ».

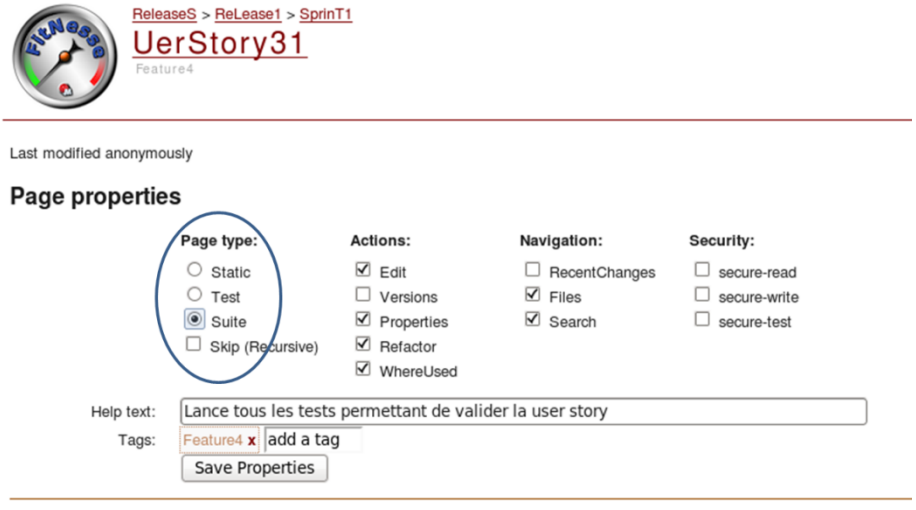

Figure 3 – Passage d'une page en série de test

Lorsque une page est configurée de manière à lancer une série de test, celle-ci s'appuie sur l'arborescence des dossiers et déroule le contenu de chaque page

A.3 Lien serveur tests

Pour que FitNesse sache quel serveur il doit appeler, il faut lui spécifier dans la page de test « root » l'emplacement du serveur : **!define TEST_RUNNER {build/bin/CSlimTestServer}**

Pour un chemin en relatif, celui-ci ne dépend pas non pas du dossier où se trouve la base de test mais du dossier où à été lancé FitNesse.

Voici un exemple d'une page permettant de lancer tous les tests :

```
* [[1° partie][>FirstPart]]
* [[2° partie][>SecondPart]]

------
!contents -R2 -g -p -f -h
------

!define TEST_SYSTEM {slim}
!define TEST_RUNNER {build/bin/CSlimTestServer}
!define COMMAND_PATTERN {%m}

!define SLIM_VERSION {0.0}
```

Actuellement, la version de CSlim accepte uniquement la version 0.0 de Slim. En effet, cette valeur est statique dans le code de cette implémentation et si une utilisation directe est faite avec FitNesse cela produira une exception. Pour palier à ce problème, il suffit dans la page « root » de donner le type de version que l'on veut utiliser avec FitNesse : **!define SLIM_VERSION {0.0}** (l'exemple ci-dessus illustre le cas dans son contexte).

Versions successives

Version	Date	Émetteur	Vérificateur	Approbateur	Motif
1.1	13/01/2014	N.Boursier	S.Frayssines	R.Patrier	Prise en compte de l'action 8 Ajout de l'annexe A Finalisation pour Release 1
1.0	26/07/2013	N. Boursier	S.Frayssines	R. Patrier	Création du document

Diffusion

Ce document est mis à disposition sous forme informatique sur serveur.
Il n'est donc pas formellement diffusé sous forme papier.