ESIGELEC
Saint-Etienne du Rouvray (76)

Pierre TAQUET

Contrat de professionnalisation

Titre de la mission

Sous la direction de Le Tuteur.

Date Visa du tuteur industriel	Date Visa du tuteur pédagogique	Date Visa du service formation continue

Sommaire

Introduction

La 1ème partie de mon contrat s'est étalée du 25 juillet au 29 septembre 2015, période durant laquelle j'ai pu, d'une part, me familiariser avec les différents outils d'un employé SAP. D'autre part, étudier le document d'architecture de WebI et apprendre à utiliser ses caractéristiques de base : , , , , , .

Chapitre 1

Présentation de l'entreprise

1.1 SAP dans le monde

1.2 Context actuel

La structure actuelle est le fruit de la fusion entre SAP France (+/- 500 salariés) et Business Objects (BOBJ - +/- 1000 salariés) dont la date légale est le 1er janvier 2010. Précédemment, BOBJ avait absorbé une autre entité, CARTESIS. D'autres petites entreprises (60/100 salariés) sont en cours d'intégration avec salariés français en 2013 : Sybase (au 01/04/2013), Success Factors et Ariba. avec crystal object ils ont intégré les deux en même tempds => la 4.0

1.3 Présentation de l'équipe

Nom de l'équipe : P&I BIT BI Suite Paris ST Transverse team
Diagramme présentant l'équipe
Les tickets : IT ticket, hrTicket, Ticket CSS

Chapitre 2

Software tester

2.1 Généralités sur le test

Interview avec Fabien

2.2 Le test à SAP

!!!!!!!!!!!Présenter l'architecture de test et les différents process!!!!!!!!!!!!!

L'équipe Tranverse est particulièrement foculisée d'une part, sur le test coverage, les tests de régression, les tests de performance ; et d'autre part sur le principe « un bug un test ». Ce principe consiste, comme son nom l'indique, que chacun des bugs fixés par un développeur soit systématiquement testé par un ST

SAP utilise un grand nombre d'outils pour gérer le code de ses nombreux produits, possédant chacun plusieurs branches, ayant chacune une suite de tests exécutée quotidiennement. Globalement les codes, quels qu'ils soient, sont hébergés sur le gestionnaire de code source Perforce. La compilation journalière est exécutée par Jenkins ou ASTEC, dépendamment des équipes et des produits, dont les résultats sont automatiquement envoyés aux personnes concernées (cf. annexe **??** page **??** qui présente l'un de ces mails automatiques).

Les ST sont régulièrement tenus au fait des tests à implémenter aussi bien par Jira ou Java Correction WorkBench que par liste de distribution de mails (cf. figure 2.1 page 6 pour le process complet)

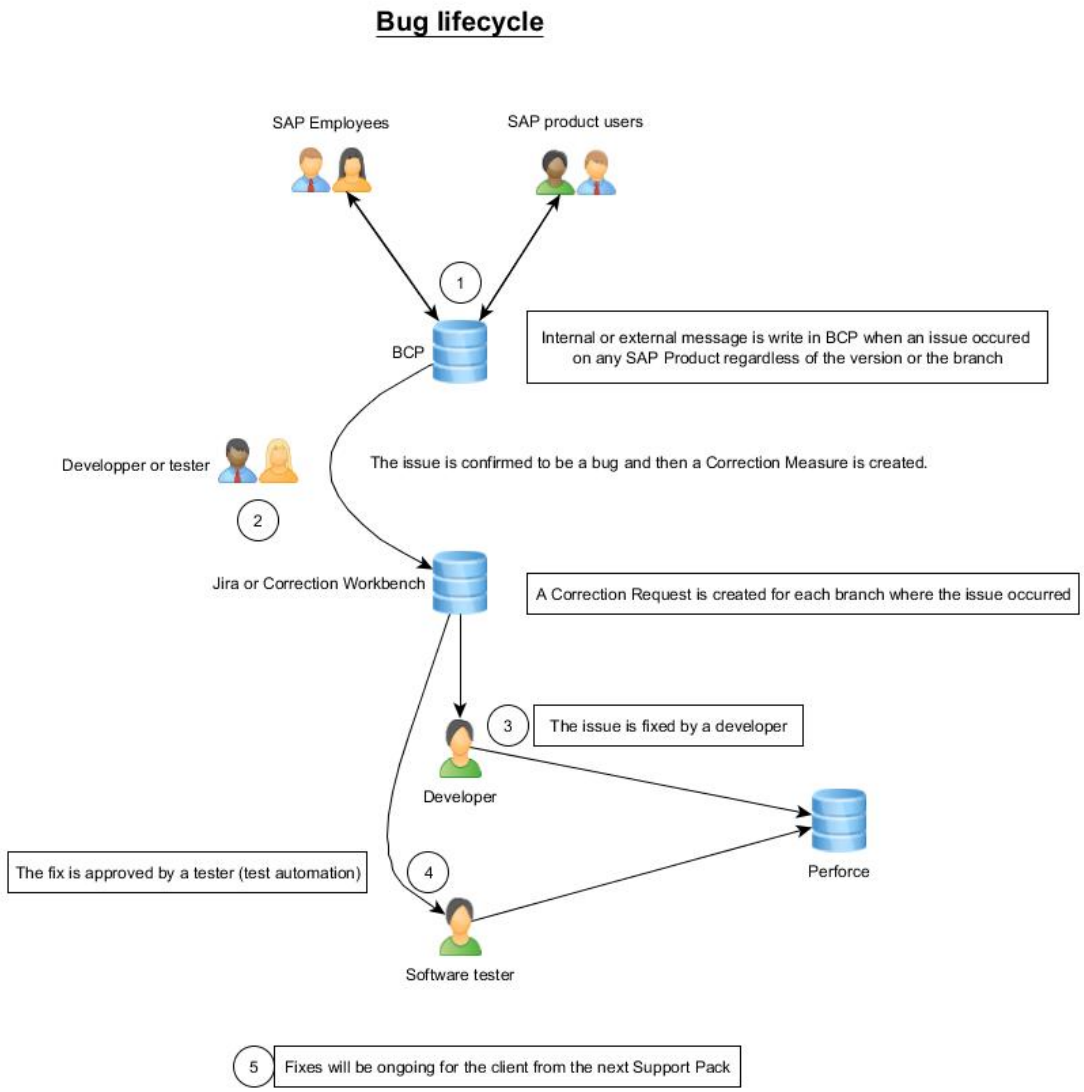


FIGURE 2.1 – Le process de test

2.3 Présentation du produit testé : WebI

WebI est un logiciel de BI permettant d'accéder à des données stockées en ligne dans un « univers » (anciennement des fichiers aux extensions .unv, mainteant .unx). L'accès aux données peut se faire via rich client via applet ou via le client dhtml.

2.4 Préparation aux tests

Mes 1er jours à SAP se sont déroulés de la manière suivante :

— Présentation à l'équipe et visite des locaux
— Réunion avec mon tuteur et mon manager pour une description de la mission
— Récupération des différents droits d'accès aux serveurs
— Familiarisation avec les outils internes (tickets HR, CSS, IT, ...)
— Installation des logiciels nécessaires au développement (IDE, SCM, éditeur de texte)
— Mise en place du framework de test (création workspace perforce) à l'aide de Christophe DOLIMONT
— Au terme de la 1ère semaine, j'ai pu commencer à utiliser le framework de test en me basant sur les tests déjà existants

Une fd

2.5 Déroulement de la 1ème mission

!!!!!cf journal de bord tests!!!!!!!!!!!
!!!!! caractéristique de base de WebI!!!!!!!!!!!
L'objectif principal de cette mission était d'être, à terme, capable d'implémenter seul un test automatique. La difficulté majeure rencontrée au début de cette période à été de comprendre la logique de test du framework de test. Les points suivants sont importants pour pouvoir utiliser le framework correctement :

— L'intégration du test à la suite de tests
— Quel type de test mettre en place, statique ou dynamique ?
— Comment initialiser un test, les constructeurs des tests dynamiques peuvent se baser sur un lcmbiar, un wid ou une queryspec
— Les tests ne se basant pas sur les mêmes versions du produits, les JARs ne sont jamais les mêmes
— Où trouver les fichiers nécessaires

2.5.1 Tests statiques ou dynamiques

Le test, qu'il soit statique ou dynamique, sera exécuté avec tous les autres dès lors que le nom du test plan est inscrit dans la test suite.
La différence fonctionnelle entre ces deux types de tests est que l'un ne fait que comparer deux doc alors que l'autre fait des modifications sans nécessairement le comparer à une référence.
La différence horaire est très importante, pour un test statique il n'est pas nécessaire d'implémenter de testcase ! Il suffit de générer la référence, de la mettre dans le dossier dédié, compléter le script.xml et la test suite, implémenter le test plan et commit.

Tests statiques

Le test statique

— ne fait que comparer un document par rapport à une référence, aucun testcase n'est donc à implémenter

— ne demande que très peu de connaissances technique, que ce soit en Java ou sur WebI

Lors de l'exécution d'un test statique, un fichier est généré à partir d'un wid, son format (txt_ [1], txt ou doc) est précisé dans le fichier script.xml.
Ensuite, à la fin de l'exécution du test, ce fichier généré est comparé avec un fichier de référence (ce fichier de référence étant généré par le ST lors de l'implémentation de son test en mettant l'option savingtxt à true).
Si les deux fichier sont identiques le test est correct, sinon il échoue.
Pour l'exécution d'un test statique il ne faut qu'implémenter un testplan qui respecte la structure suivante :

```
1  package rebean_wi.customers.Aerospatiale_Matra_Airbus.ID_187739;

3  import model.filters.Mode;
   import model.filters.Severity;
5  import model.filters.Type;
   import model.filters.testplan.Suite;
7  import model.filters.testplan.TestPlanType;
   import model.filters.testplan.features.Features_REBEAN;
9
   import org.junit.Test;
11
   import tests.exported.annotations.BOTest;
13 import tests.exported.annotations.BOTestPlan;
   import extensions.toolbox.WIStaticTestPlanDefinition;
15
   @BOTestPlan(Type = TestPlanType.FEATURE_VERIFICATION, Suite = Suite.AURORA,
       Feat = Features_REBEAN.WI)
17 public class CM_{cmNumber} extends WIStaticTestPlanDefinition{
     private static String _scriptID = "CM_{cmNumber}_{short
19 text}";
     public CM_{cmNumber} () {
21     super(_scriptID);
     }
23
     @Test
25   @BOTest(Objective = "Test CM_{cmNumber}_{short
   text}' functionality", Severity = Severity.CRITICAL, Author = "", Mode = Mode.
       PROD, Type = Type.FUNCTIONAL)
27   public void CM_{cmNumber}_{short text}" () {
       logNumericResults(launchTC(getTestcase("CM_{cmNumber}_{short text}")));
```

1. type personnalisé interne à SAP

```
  }
}
```

Tests dynamiques

À la différence du test statique, le test dynamique va modifier le document après ouverture. Ceci permettant de s'assurer que la mécanique interne de WebI produit l'effet escompté sur le document. La comparaison avec un document de référence est bien évidemment possible.
Un testplan respecte l'implémentation suivante :

```
package rebean_wi.customers.{customerName}.{customerID};
//imports
@BOTestPlan(Type = TestPlanType.FEATURE_VERIFICATION, Suite = Suite.AURORA41,
    Feat = Features_REBEAN.WI)
public class CM_{CMNumber} extends WIDynamicTestPlanDefinition {
  private static String _scriptID = " CM_{CMNumber}_{text}";
  public CM_{CMNumber}() {
    super(_scriptID);
  }
  @Test
  @BOTest(Objective = "Test 'CM_801959_2014_ValuesMissing' functionality",
    Severity = Severity.CRITICAL, Author = "ptaquet", Mode = Mode.PROD, Type =
    Type.FUNCTIONAL)
  public void CM_{CMNumber}_{text}() {
  logNumericResults(launchTC(getTestcase("aurora_customers. .{customerName}.CM_
    {CMNumber}_{text}");
  }
}
```

Et son testcase respecte l'implémentation suivante :

```
package aurora_customers.{ customerName};
public class CM_{CMNumber}_{text} extends MonoDocTestcase {
  MonoDocTestcaseConfigInfo _tccInfo;
  //private final static String _docPath = "";
  MSGStep _step = null;

  Public CM_{CMNumber}_{text} (MonoDocTestcaseConfigInfo tccInfo,String docName
    ) {
    super(tccInfo, docName, "corporate", _docPath, true);
    _tccInfo = tccInfo;
  }
  @Override
  protected void run() throws Exception {
    startTestcaseStep("");
  //
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    //
16    _step = _msgHelper.startStep("");
    //Here, the implementation of the code
18    _msgHelper.stopStep(_step);

20
    stopTestcaseStepWithoutActionWI();//Stop without compare
22  }
}
```

À la lecture du test case on peut observer le constructeur de la super classe, celui-ci prends un certain nombre d'arguments. Il existe 10 constructeurs différents pour la super classe MonoDocTestCase mais 2 sont particulièrement important. L'un permet de baser son test sur un document généré à partir d'une queryspec, l'autre sur un document .wid. Par exemple le constructeur se basant sur un wid :

```
1 MonoDocTestCase(MonoDocTestCaseConfigInfo tccInfo, String sDocumentName, String
     sCategoryType, String sCategory, Boolean useAuroraCtx);
```

où

tccInfo
: Objet contenant les informations générales relatives au test case

sDocumentName
: Nom du document .wid qui sera chargé lors de la construction de l'objet

sCategoryType
: Nom de la catégorie, en général : « corporate »

sCategory
: Path du dossier dans lequel est stocké le .wid. Doit respecter la nomenclature suivante : /auto/{scriptname}/wid

useAuroraCtx
: Le test en question porte t-il sur aurora ?

Dans tous les cas, lorsque l'objet testcase est instancié, nous pouvons implémenter un code manipulant le document au niveau SDK ; ce qui signifie que l'on ne simule pas un clic sur un bouton mais que l'on appel l'une des méthodes « derrière » ce bouton, reproduisant ainsi le comportement d'une manipulation au niveau GUI

Exécution d'un test

Lors de l'exécution d'un test plusieurs fichiers sont générés en divers endroits, il y a par exemple :

Des fichiers de log
: sd

La sortie console

Enregistrée dans
. . . \Workspace_Aurora\rebean\logs\{scriptname}

Les fichiers générés

Enregistrés dans
. . . \Workspace_Aurora\rebean\results\Your Build Number\res\{ scriptName}\CM_{cmNumber}_{sh
text}\
uniquement si l'enregistrement des ressources est spécifié dans le script.xml
Si savingDoc est à True le document de référence est enregistré sur le CMS dans le dossier Folders/Public Folders/auto/, voir l'illustration 2.2 page 11, et le document généré est enregistré dans My Documents/My Favorites/Personnals Documents, voir l'illustration 2.3 page 12

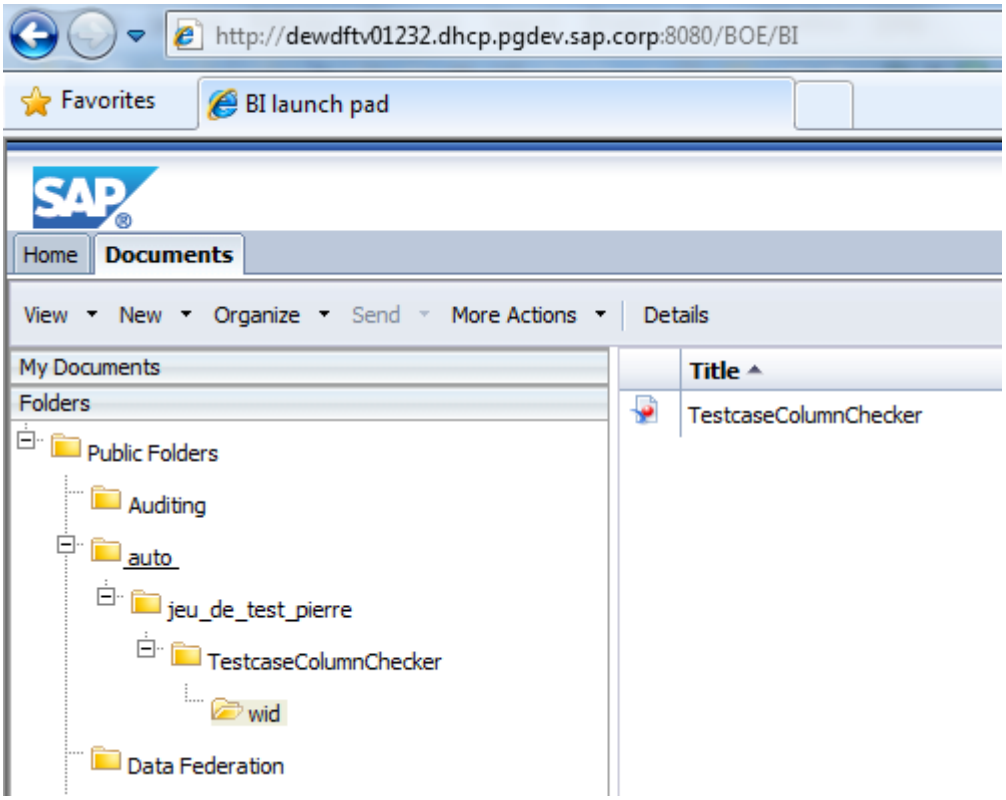


FIGURE 2.2 – Capture d'écran de WebI de l'arborescence du document de référence

Les fichiers de référence

Les .wid sont enregistrés dans
. . . \Resources_AURORA\storage\auto\{ scriptName}\wid
et les autres documents de références (txt, html, etc.) sont dans
. . . \Resources_AURORA\storage\auto\{scriptName}\CM_{cmNumber}_{short

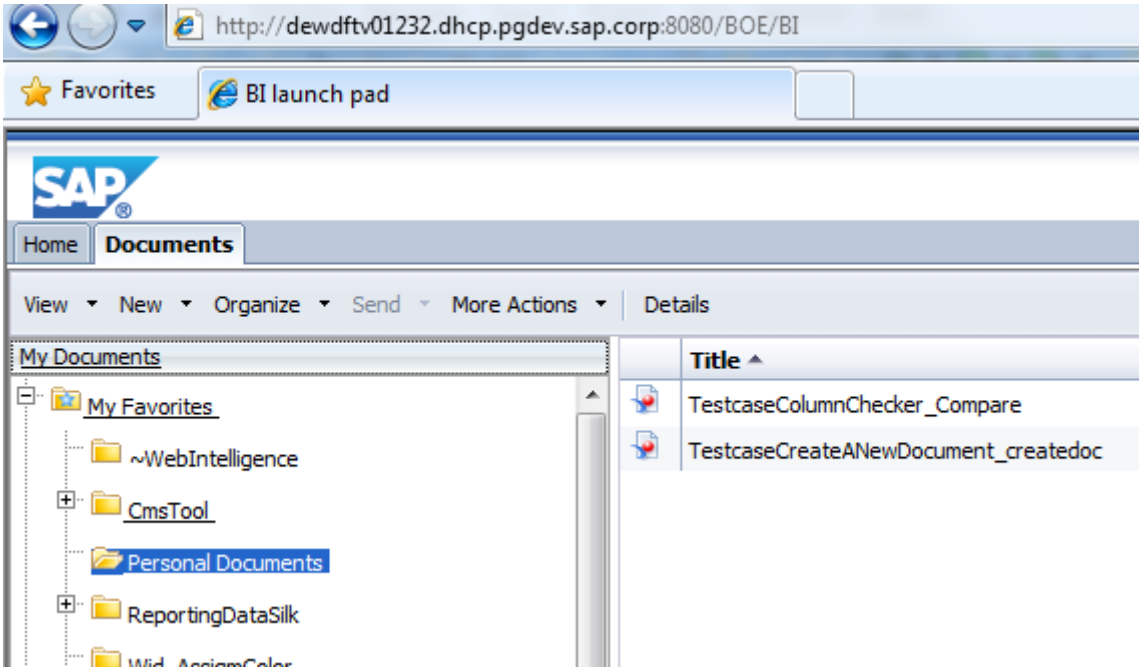


FIGURE 2.3 – Capture d'écran de WebI de l'arborescence du document généré

text}\
voir une illustration de cette arborescence figure 2.4 page 12

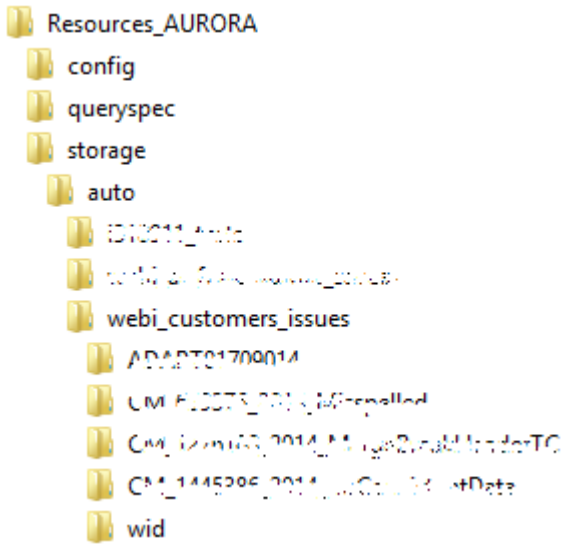


FIGURE 2.4 – Capture d'écran de WebI de l'arborescence des références locales

Les logs du serveur tomcat

Disponible dans le dossier d'installation du serveur ces logs sont trop verbeux pour pouvoir être utiliser, mais sont cependant utiles dans certains cas.

2.5.2 De l'étude à l'intégration

Tout d'abord, les requêtes de test auto ne sont pas toujours réalisables à notre niveau, il faut donc d'abord vérifier la faisabilité du test. Une fois que l'on a validé que la réalisation est possible et que l'on a choisie une stratégie de test, il faut préparer l'environnement de travail, c'est-à-dire construire la coquille vide du test désiré.

Analyse du test à implémenter

Étudier le bug Au préalable, toutes les informations que l'on a sur le bug a tester se trouvent sur JCWB, étudier le workflow provoquant le bug, la description du bug, la/les branche(s) sur laquelle/lesquelles il survient (voir figure 2.5 page 13)
Ceci fait, il est intéressant d'aller étudier le code qui a été modifié pour corriger

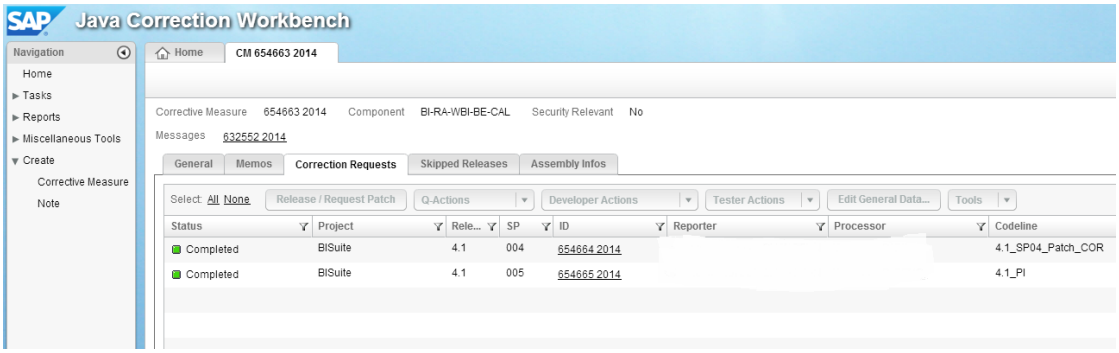


FIGURE 2.5 – Écran de JCWB propre à une CM

le bug. Pour cela il suffit d'utiliser la fonction « diff against previous revision » du SCM pour obtenir la liste exhaustive des fichiers modifiés ainsi qu'un vis-à-vis sur les versions pré-correctif et post-correctif (voir figure 2.6 page 14)

Reproduire le problème Une fois que le bug est bien compris il nous incombe de reproduire à la main le workflow et de valider l'existence du bug sur la version buggée ainsi que le bon fonctionnement de la version corrigée. À cette étape, si le bug survient sur le CMS (client léger), il est intéressant d'utiliser le debugger du navigateur pour observer les données transitant.

Définir la stratégie de test Maintenant que le problème est bien compris et localisé, nous pouvons savoir s'il est possible de le tester. Si non, soit le problème ne peut pas être testé soit nous redirigeons la correction vers l'équipe de testeurs concernée (plus ou moins haut ou bas niveau).
Si l'implémentation du test est possible, il faut choisir la meilleure manière de

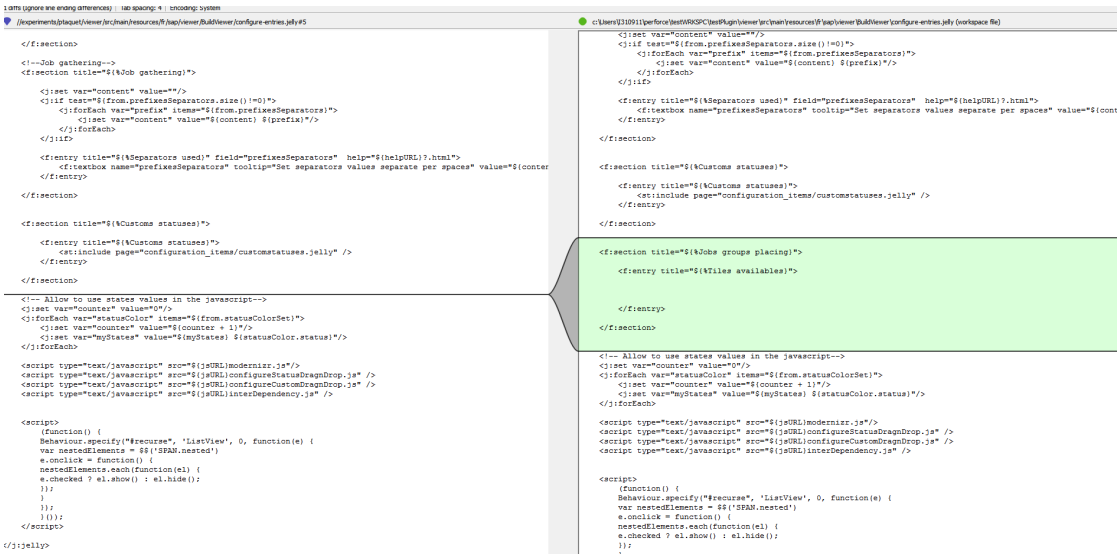


FIGURE 2.6 – Écran de comparaison des 2 versions d'un même fichier (avant et après correctif)

tester l'existence et la non-existence du bug ainsi que le moyen le plus rapide d'arriver à reproduire le problème. Par exemple faut-il un test statique ou dynamique ? Partir d'une queryspec ou d'un document .wid ?

Implémentation de la coquille vide

L'intérêt de coder d'abord la coquille vide est d'avoir un code qui compile mais qui ne fait encore rien. Ce qui garantit que tout se passe correctement au niveau de la nomenclature, la création du test case et, au besoin, la génération sur serveur des .wid. Comme schématisé figure 2.7 page 14, chaque test automatique est intégré à un test plan, lui-même étant intégrés à une test suite.

FIGURE 2.7 – Diagramme UML des suites de tests

Ci-dessous les fichiers à implémenter :

1. **Test plan** Créer le nouveau test plan dans le package correspondant au client qui a remonté le bug (dans testplan.srebean_wi.customers.{clientID}). Renseigner

toutes les informations relatives au test, si ce fichier est correctement implémenté il ne sera plus nécessaire de le modifier par la suite.

2. **Test case** Créer le test case dans le package correspondant au client (dans testcases.aurora_customers.{clientID}). Attention à respecter le pattern de nommage « CM_{CM_id}_shortText ».
3. **Test suite** Dans le package des tests plan, on peut trouver la test suite qu'il faut modifier. Il suffit de rajouter le nom du test plan à la liste de test plan que la test suite exécute.
4. **script.xml** Ajouter les différents paramètres correspondants au test.
5. **ressources** Générer les documents ressources nécessaires (queryspec, .wid, etc.) et les ajouter au dossier portant le nom de la CM
6. **parameters.xml** Renseigner l'url du CMS ciblé et mettre à jour les extracted JARs permettant au test vide de compiler
7. Vérifier l'intégrité de la change list de perforce et finir par push les modification.

D'une manière générale, la structure générale à connaitre pour implémenter correctement la coquille vide est illustrée figure 2.8 page 17

Les ressources

Les ressources sont très importantes dans le contexte du test, celles-ci se présentent sous plusieurs jours différents :

1. un document .wid créé en suivant à la lettre le workflow à tester mais dont la construction à été arrêtée juste avant que ne survienne le bug.
   Cette ressource sert à utiliser un document déjà construit et permet au ST de n'automatiser que la partie à tester.
2. un fichier (txt, txt_, pdf, doc, xls, xls, ...) considéré comme une référence
   Cette ressource est comparée au document obtenu à la fin du workflow pour garantir la similitude.
3. une queryspec, c'est un fichier xml représentatif du document .wid

Obtenir un fichier wid

Via le CMS Il suffit de parcourir l'arborescence du CMS pour arriver à l'emplacement du .wid. Dans les propiétés du fichier il y a son nom complet (différent du nom dans WebI) avec son arborescence à partir du dossier Input (figure 2.9 page 18). Ensuite, dans le système de fichiers du serveur (par exemple : « \\dewdftv01634.dhcp.pgdev.sap.corp\c$ ») aller dans « \Program Files (x86)\SAP BusinessObjects\SAP BusinessObjects Enterprise XI 4.0\FileStore\Input » et copier/coller le chemin d'accès au fichier. Le document .wid se trouve dans le répertoire en question.

Via le Rich Client Attention à la version du rich client qui doit être la même que celle du CMS. Ouvrir une connexion pointée sur le bon CMS, ouvrir le document désiré et enregistrer sous le document .wid.

2.6 Bilan de mes 1ères missions

Tout au long des mois de juillet, août et septembre j'ai implémenter beaucoup de tests [2] pour des bugs divers et à l'impact plus ou moins critique. J'ai beaucoup appris de mes erreurs, surtout lorsque plusieurs jours de travail s'avéraient inutiles parce qu'une meilleure solution, évidente pour qui sait, existait.

2.6.1 Les 1ères erreurs

Les principales pertes de temps : coder des choses inutiles ou déjà existantes.
réinventer la roue. Beaucoup de choses que j'ai codé se sont avérés être déjà existantes dans le framework

perdre du temps à implémenter des choses inutiles (exemple : création d'un doc niv sdk alors que l'on peut juste le récupérer)

Casser la build : ne jamais push le vendredi soir au risque de casser la build du week-end.

2.6.2 Les acquis

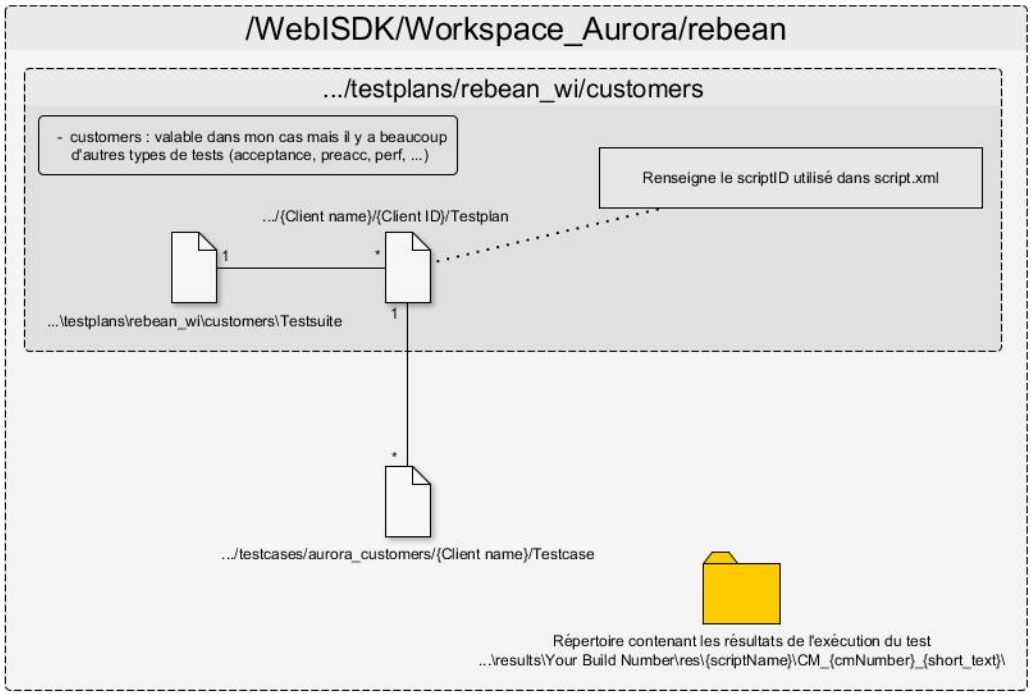
ce qu'il faut faire pour tester

comprendre le test (étudier le workflow et cibler exactement la partie du workflow à tester) mettre en place la coquille vide du test et générer et/ou récupérer les fichiers utiles au développement et/ou à l'exécution du test. Tester le test ! ie exécuter sur 2 versions, l'une fixée l'autre non. push le test

ne jamais push le vendredi soir

la composition d'un document WebI au niveau SDK

2. cf. annexe ?? page ?? pour la liste complète des tests implémentés



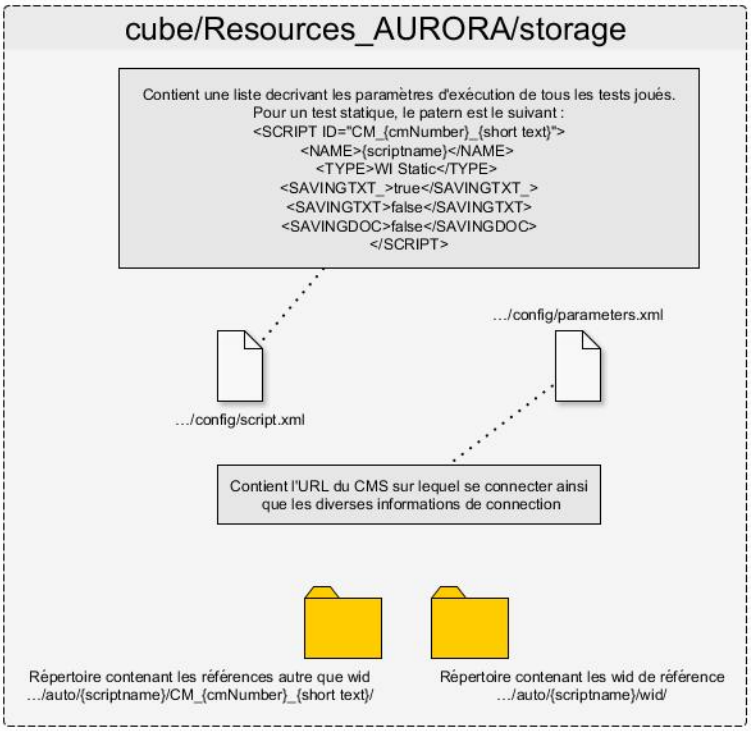


FIGURE 2.8 – Diagramme représentant les différents éléments qui compose la coquille vide d'un test dynamique

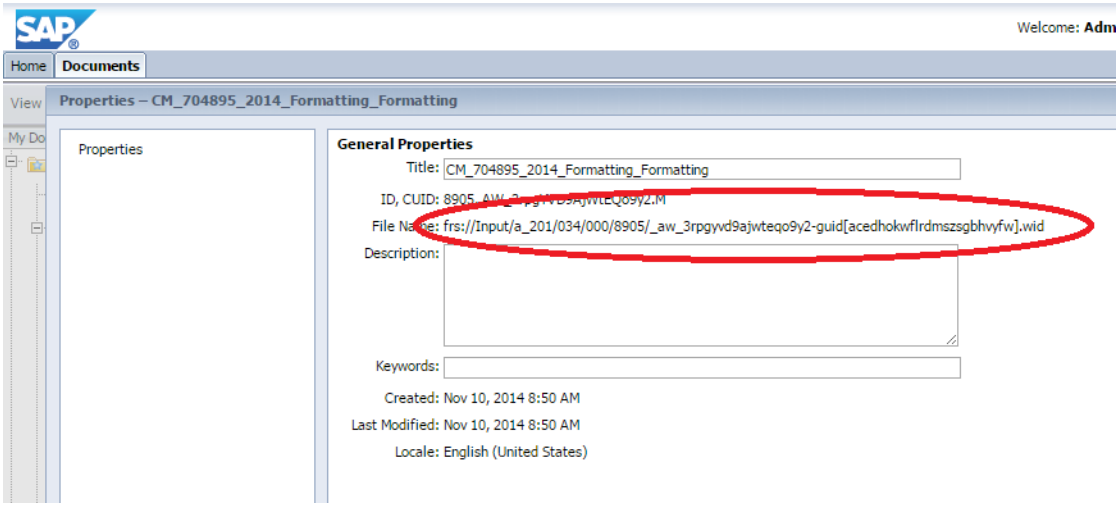


FIGURE 2.9 – Capture de l'écran de propriété d'un document WebI

Chapitre 3

Migration Jira/Java Correction WorkBench

3.1 Qu'est-ce que Jira et Java Correction WorkBench

Jira est un loigiciel de traçabilité des problèmes développer par Atlassian dont l'usage est gratuit pour les organisations à but non lucratif, de charité ou les projet open-source. SAP désire migrer de système de traçabilité pour utiliser maintenant Java Correction WorkBench (JCWB ou CWB).

JCWB est un logiciel..... voir Fabien

3.2 Présentation du contexte

Comment cela se passe actuellement à SAP à SAP tous les codes des différents projets sont hébergés sur le gestionnaire de version Perforce (SCM : Source Code Management) et ceux-ci sont compilés plusieurs fois chaque jour grâce à Jenkins ou ASTEC (CIS : Continuous Integration Software).
Chaque projet se compose d'une part son code source mais aussi d'une grande quantité de tests qui sont joués pour garantir [1] le bon fonctionnement du produit. Lorsqu'il y a un problème sur la build, que ce soit une erreur de compilation ou un test qui échoue, le statuts de la build change en fonction du problème. Dès lors que quelqu'un c'est aperçu du problème celui-ci inscrit un defect dans Jira [2].

1. La valeur de la garantie dépends directement du test coverage
2. L'utilité de Jira est bien plus large que la simple déclaration d'un test échoué, il sert à décrire n'importe quel problème quelle que soit la version, la branche ou le produit

3.2.1 Plugin de reporting

Pour leur permettre d'avoir un rapport visuel sur l'état des builds, ils utilisent le plugin Radiator. Ce plugin permet l'affichage, sur un seul écran divisé, des groupes de jobs[3]. Un groupe de jobs est un carré coloré dont la couleur représente l'état des jobs qui le compose.
Les statuts pris en compte sont les statuts Jenkins[4] ainsi qu'un statuts supplémentaire issue du plugin Claim. Ce plugin permet à un développeur de claim une erreur, c'est-à-dire que le projet comporte une imformation supplémentaire qui est un nom d'utilisateur et le message que celui-ci a laisser.

En résumé Radiator :

— permet de rassembler les jobs en un seul groupe de jobs
— où chacun des jobs a un statuts[5]
— le groupe de projet arborant la couleur jugée la plus importante
— de prendre en compte les informations supplémentaires du plugin claim

Par exemple, si nous avons deux groupes composés chacun de trois jobs, tous différents. Supposons que, sur les 6 jobs, l'un soit en échec les autres étant en succès. Nous aurons donc une vue colorée en verte (parce que tous les projets sont en succès) et la deuxième vue apparaissant en rouge (l'un des jobs ayant échoués) ou bien en orange si l'échec est investigué (échec claim par un développeur).

Problème

Cette solution permet de visualiser les statuts des jobs mais ne fait pas de distinction parmis les jobs échoués. Parmis ceux-ci on peut distinguer :

— les jobs échoués depuis très longtemps
— les jobs échoués avec un defect enregistré

L'inconvénient est que les tests échoués récemment et non reconnus apparaissent de la même manière que les tests échoués et reconnus. La solution provisoire mais permettant de ne plus polluer l'affichage avec du rouge, conciste à faire passer les jobs en échecs avec defect (ie. reconnus et enregistrés comme échecs dans Jira ou CWB) sont passés en vert. De cette manière, seul les nouveaux échecs apparaissent en rouge. En grande partie des régressions.
Le revers de la médaille est que l'affichage n'est plus représentatif de la réalité, une partie de l'information est perdue puisqu'un test apparaissant en succès peut en fait ne pas l'être.

Sauf qu'actuellement seuls les defects rentrés sur Jira sont pris en compte, la mécanique en place ne permet pas encore de récupérer ces mêmes informations de CWB.
Puisque de plus en plus de defects seront enregistrés sur CWB, et dans un soucis d'homogénéisation du process actuel, il faut rationnaliser le process actuel et permettre au statuts d'être ajustés quelque soit l'origine du defect (Jira ou CWB).

3. Typiquement, un job correspond à un projet logiciel
4. Error, Failure, Unstable, Aborted, Not built, Success
5. statuts Jenkins

3.3 Process actuel

1- Investigation du process actuel

la méthode apply de JUnit est surchargée
il y a une annotation qui permet de checker le status
l'annotation est prise en compte en fonction d'un paramètre -D

même résultat pour acceptance et GTP ? ? ? ? -» fabien

Dans l'état actuel des choses le process utilisé est illustré figure 3.1 page 21

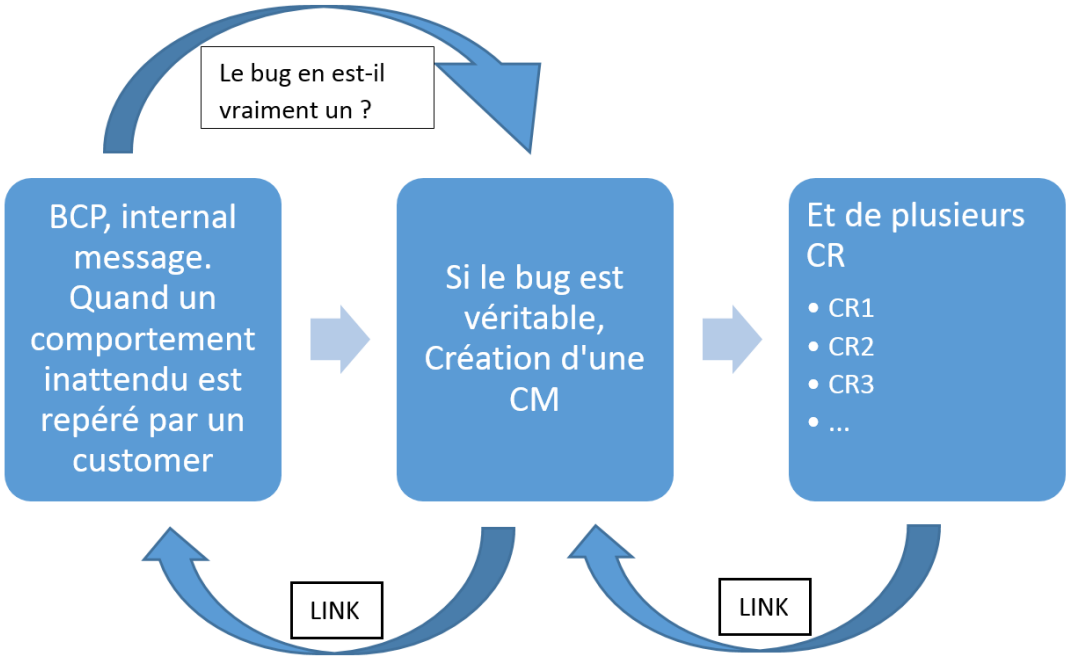


FIGURE 3.1 – Le process utilisé lors de gestion d'un bug

En résumé

— JUnit est surchargé par la classe JiraIssueWatcher.java
— Je vais devoir utiliser le système choisi par SAP pour fournir les credentials : prod-pass-access
— Il existe une API permettant d'obtenir les informations que je désire : CWB HTTP API

3.4 Travail effectué

La première partie du travail s'est cantonée à des recherches et des échange de mails internes à SAP.

!!!!! Ici je décris mes différents mails avec marco et Fabien !!!!!

3.4.1 Fonctionnement et utilisation de prod-pass-access

3.5 Résultats obtenus

Maintenant la mission précédente achévée, nous pouvons récupérer les infos de status quelque soit son emplacement (Jira ou CWB) depuis test-defects.xml (cf Migration)

-> connaissances Java J'ai approfondi mes connaissances sur les annotations et la manière de les implémenter. J'ai implémenté ma première classe abstraite

Chapitre 4

Plugin Jenkins

4.1 Contexte initial

Jenkins est le logiciel d'intégration continue, il utilisé à SAP dans le contexte des tests . Son rôle est d'intégrer les différents projets, tels que configuré par l'utilisateur, et met à disposition les résultats obtenus. Il permet d'avoir un retour régulier sur l'état des builds et du résultat de leurs tests.

Un plugin permettant une vision globale sur l'état des builds est déjà en place, Radiator. La figure 4.1 page 24 présente son environnement d'utilisation. Nous pouvons y voir les développeurs et les ST qui alimentent le code hébergé sur Perforce que Jenkins l'utilise dans ses diverses intégrations. Au centre, le plugin permet un retour visuel rapide des livraisons Perforce.
Ce plugin est aussi conçu pour fonctionner avec un autre plugin, le plugin Claim. Il permet à quelqu'un visitant la page d'un job en échec de le « claimer », inscrivant son identifiant ainsi qu'un message à destination de quiconque visiterai cette même page. Ceci permet de signaler que l'investigation est en cours.

Pour résumer ses caractéristiques principales, c'est un plugin qui permet :
— d'afficher un écran où chaque couleur correspond à un statut de build
— de rassembler les jobs par leurs préfixes communs
— de prendre en compte les claims
L'inconvénient est que le nombre de status existant ne permet pas la précision nécessaire pour définir l'état d'un jobs. Le plugin Radiator n'offre pas la malléabilité nécessaire. Voici la liste des différents états gérés par ce plugin :

SUCCESS Le projet compile correctement et tous les tests sont passés

ABORTED La compilation à été arrêtée en cours d'exécution

FAILURE Le code ne compile pas

ERROR

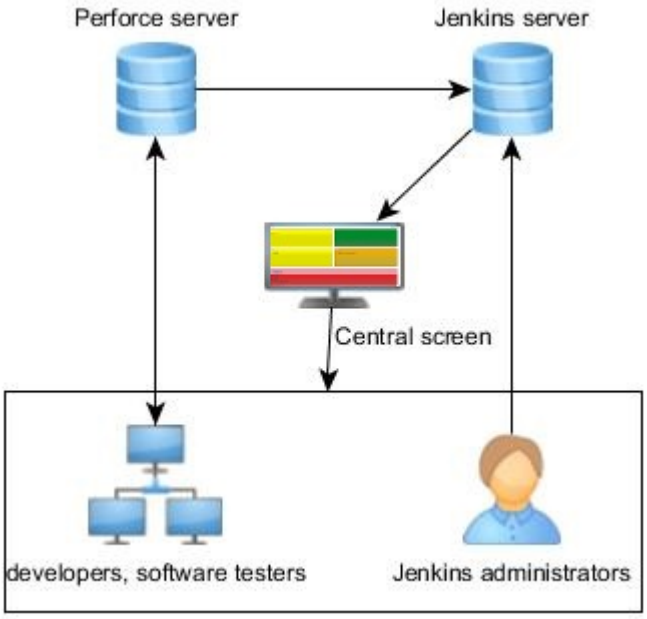


FIGURE 4.1 – L'environnement d'utilisation du plugin

NOT BUILT Le projet n'a pas été compilé

UNSTABLE Des tests JUnit ne sont pas passés mais le code compile

CLAIM Le job en échec à été claim par quelqu'un

Tel qu'il se présente le plugin ne permet pas de donner des informations autres que celles issues de Jenkins ou du plugin Claim. Il ne permet donc pas de donner de renseignements sur le defect associé à ce job. Tenir compte du defect permettrai de faire ressortir les régressions et autres erreurs, séparant ainsi les échecs pris en charge de ceux qui ne le sont pas.
Suite à la mission précédente nous avons maintenant accès aux informations relatives aux defects (celle-ci sont enregistrées dans les archives des builds sur le serveur Jenkins), dans un fichier XML que nous avons nommé test-defects (exemple page 25).

La solution proposée

Dans un 1er, et en tant qu'exercice, il faudra réaliser un plugin offrant les mêmes possibilités d'affichage des builds. C'est-à-dire rassembler les jobs par préfixe et afficher le résultat général de tous ces groupes ainsi créés sur un unique écran. Ajouté aux caractéristiques, ce plugin doit pouvoir prendre en compte les informations du plugin Claim.

Dans un second temps, il faudra pouvoir ajouter un ou plusieurs statut(s) supplémentaire(s) ainsi que de ré-ajuster leur ordre de priorité. L'ajout de ce nouveau statuts doit être générique de sorte qu'un status quelconque puisse être ajouté.
Le plugin est donc capable de receuillir des informations dans des fichiers sans savoir, au

préalable, comment seront structurées ses données. Rappelons que lors de l'exécution des tests, un fichier test-defect est généré contenant la liste [1] des jobs avec defect (exemple page 25).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<testcases>
  <testcase classname="test.sap.sl.sdk.authoring.IssueWatcherTest" name="
    failingTestNoDefect" status="failed"/>
  <testcase classname="test.sap.sl.sdk.authoring.IssueWatcherTest" name="
    successTestNoDefect" status="passed"/>
  <testcase actualStatus="failed" classname="test.sap.sl.sdk.authoring.
    IssueWatcherTest" message="Error" name="failingTest3_ClassDefect" status="
    passed">
    <defect expectedMessage="Error" link="https://sapjira.wdf.sap.corp/browse/
    BITBIWEBISL2-1542" status="Open" type="jira"/>
  </testcase>
  <testcase actualStatus="failed" classname="test.sap.sl.sdk.authoring.
    IssueWatcherTest" message="Error" name="failingTest2_ClassDefect" status="
    failed">
    <defect expectedMessage="Failing test" link="https://css.wdf.sap.corp/sap/
    bc/bsp/spn/jcwb?crPointer=0120061532000011591820 15" status="In Process"
    type="cwb"/>
  </testcase>
  <testcase actualStatus="failed" classname="test.sap.sl.sdk.authoring.
    IssueWatcherTest" message="Failing test" name="
    failingTest_CWB_MatchingMessage" status="passed">
    <defect expectedMessage="Failing test" link="https://css.wdf.sap.corp/sap/
    bc/bsp/spn/jcwb?crPointer=0120061532000011591820 15" status="In Process"
    type="cwb"/>
  </testcase>
  <testcase actualStatus="failed" classname="test.sap.sl.sdk.authoring.
    IssueWatcherTest" message="Failing test" name="failingTest1_ClassDefect"
    status="passed">
    <defect expectedMessage="Failing test" link="https://css.wdf.sap.corp/sap/
    bc/bsp/spn/jcwb?crPointer=0120061532000011591820 15" status="In Process"
    type="cwb"/>
  </testcase>
  <testcase actualStatus="failed" classname="test.sap.sl.sdk.authoring.
    IssueWatcherTest" message="Failing test" name="
    failingTest_Jira_MatchingMessage" status="passed">
    <defect expectedMessage="Failing test" link="https://sapjira.wdf.sap.corp/
    browse/BITBIWEBISL2-1542" status="Open" type="jira"/>
  </testcase>
  <testcase actualStatus="failed" classname="test.sap.sl.sdk.authoring.
    IssueWatcherTest" message="Error" name="failingTest_CWB_NoMatchingMessage"
    status="failed">
    <defect expectedMessage="Failing test" link="https://css.wdf.sap.corp/sap/
```

1. Au format xml

```
    bc/bsp/spn/jcwb?crPointer=0120061532000011591820l5" status="In Process"
    type="cwb"/>
  </testcase>
</testcases>
```

4.2 Étude préalable

Les 1er impératifs ont été de maitraiser le langage et les outils qui me permettrai de mener ce projet à terme. Je n'avais jamais entendu parler de Jenkins ou d'un quelconque logiciel d'intégration continue, et a forciori sur la manière de l'étendre.

Il existe un archétype maven (cf annexe **??** page **??**) permettant de générer le squelette d'un plugin Jenkins

J'ai donc commencé par installer Jenkins et ses plugins, tels qu'ils sont utilisés à SAP, de manière à me familiariser avec ceux-ci. Pour examiner le comportementde Jenkins j'ai créé de faux projets me permettant de simuler les différents états possibles. Ceux-ci m'ont permis d'observer les différences entre les statuts Jenkins et JUnit, certains statuts portent les mêmes noms mais ne signifient pas les mêmes choses provoquent quelques incompréhensions. La signification de ces statuts étant l'objet de ce plugin, il m'incombai de les comprendre. Voici le détail des configurations que j'ai testé pour déterminer l'équivalence statut JUnit/statut Jenkins :

Description du projet	**Status JUnit**	**Statut Jenkins**
projet vide et sans test	NA	Success
projet vide et avec test réussi	Passed	Success
projet vide avec test qui échoue	Failure	Unstable
projet avec erreur de compilation	NA	Error

Passée cette étape d'installations et de configurations je me suis penché sur la marche à suivre pour implémenter un plugin Jenkins.
La principale difficulté que j'ai eu à développer ce plugin a été de trouver l'information, il est reconnu sur internet que la documentation de Jenkins, lorsque l'on veut l'étendre, n'est pas clair. Mais la documentation officiel[2] offre un bel état des lieux de ce qu'il est possible de faire, décrit les différentes technologies constituant Jenkins et offre un tutoriel, trivial mais suffisant, pour comprendre les bases.

4.3 Le 1er plugin

Je me suis d'abord rendu sur la page du tutoriel officiel de Jenkins[3] afin d'obtenir un plugin fonctionnel avec lequel je pouvais faire mes expériences. Ensuite j'ai suivi un

2. https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins
3. https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial

autre tutoriel [4] non-officiel mais considérablement plus riche.

4.3.1 Le squelette du plugin

Configuration

D'après toutes les documentations disponible, le meilleur moyen de commencer le développement du plugin est d'utiliser maven [5] pour générer le squelette. Maven a besoin de configurations supplémentaires dans son settings.xml [6] pour pouvoir récupérer les sources relatives à Jenkins.
Dans « pluginGroups » il faut ajouter la ligne suivante :

```
<pluginGroup>org.jenkins-ci.tools</pluginGroup>
```

Et dans « profiles » il faut ajouter le profil suivant :

```
<profile>
<id>jenkins</id>
<activation>
<activeByDefault>false</activeByDefault>
</activation>
<repositories>
<repository>
<id>repo.jenkins-ci.org</id>
<url>http://repo.jenkins-ci.org/public/</url>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<id>repo.jenkins-ci.org</id>
<url>http://repo.jenkins-ci.org/public/</url>
</pluginRepository>
</pluginRepositories>
</profile>
```

L'attribut de la balise « activeByDefault » est à false, parcequ'on travaille avec d'autres profils sur d'autres projet. Mais il faut retenir que ce paramètre peut empêcher la génération du squelette du plugin via la commande :

```
mvn hpi:create
```

Pour exécuter cette commande ci-dessus il faut se placer dans le dossier qui recevra le projet.

4. https://cleantestcode.wordpress.com/2013/11/03/how-to-write-a-jenkins-plugin-part-1/
5. Maven 3 et JDK 6 ou plus récent
6. Situé dans le répertoire ~/.m2

Génération du squelette

À l'exécution de cette commande il est demandé de renseigner le groupId et l'artifactId, le groupId peut être org.jenkins-ci.plugins et l'artifactId est le nom du plugin. Cette commande crée l'arborescence du projet ainsi que les fichiers de base.

À la génération du squelette du plugin nous obtenons un plugin qui compile dont l'architecture est présentée figure 4.2 page 28. Certaines choses importantes sont à notées, le pom.xml [7] est généré à la base du projet dands lequel nous trouvons des informations telles que le parent de ce projet, les dépendances, l'auteur, la licence et la description.

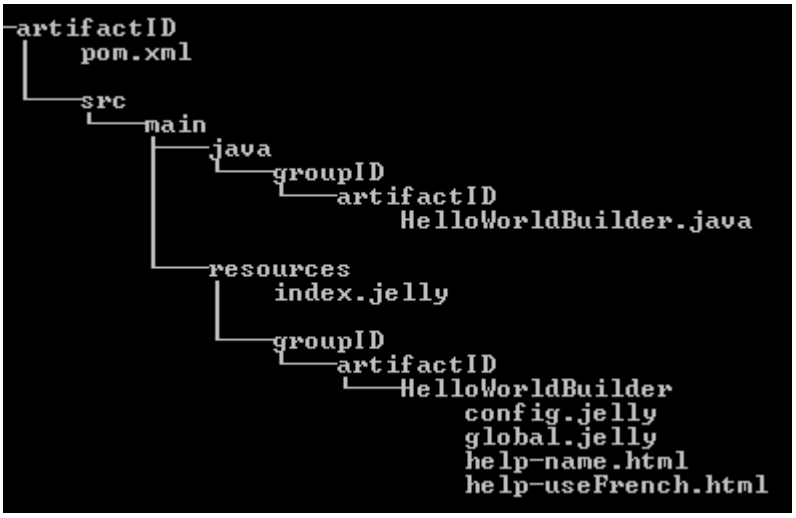


FIGURE 4.2 – Architecture d'un plugin Jenkins généré par maven

!!!!!! ok, maintenant on sais faire un plugin hello world
Plusieurs autre fichiers sont générés :

Nom du fichier	Description
index.jelly	Description utilisée à la création d'une nouvelle instance du plugin
config.jelly	Correspond à la page d'une instance du plugin
global.jelly	Correspond à la page de configuration générale du plugin
HelloWorldBuilder.java	Implémentation d'exemple du plugin
help-name.html	Fichier d'aide à la configuration lorsque clic sur le symbole d'aide
help-useFrench.html	Fichier d'aide à la configuration lorsque clic sur le symbole d'aide

!!! Tous les fichiers index.jelly config.jelly global.jelly
configure-entries.jelly main.jelly properties help
!!!Les points d'extensions

7. Project Object Model

Déploiement du squelette du plugin

Plusieurs solutions s'offre à nous pour déployer le plugin dans son environnement d'utilisation

Installation « à la main »

En ligne de commande et depuis la racine du projet [8], il faut exécuter la commande suivante :

```
1 mvn clean install -Pjenkins
```

Celle-ci va télécharger les sources nécessaires, les compiler, exécuter les tests et, si tout se passe bien, générer le fichier .hpi dans le dossier target.
Ceci fait, il faut se rendre dans Jenkins (par exemple http ://localhost :8080/ ou autre. En fonction de la configuration), s'identifier et aller dans les paramètres avancés de gestion de plugins. Ici il faut renseigner le chemin d'accès vers le .hpi et terminer.

Commande du plugin Jenkins de maven

En ligne de commande et depuis la racine du projet, il faut exécuter la commande suivante :

```
1 mvn hpi:run -Djetty.port=8090 -Pjenkins
```

Cette commande va déployer une instances de Jenkins propre à maven, ce qui nous permet d'exécuter le plugin ainsi que de le debugger. On peut très bien omettre de préciser le port utilisé si celui par défaut [9] est disponible, mais il vaut mieux éviter. Une fois que le déploiement est terminé maven signale que « Jenkins is fully up and running » nous pouvons aller voir Jenkins à l'url *http ://localhost :8080/jenkins* qui devrait ressembler à la figure 4.3 page 30.

Essaie du squelette du plugin

Jenkins se redémarre et voilà notre plugin intégré à celui-ci. Nous pouvons aller voir l'écran de configuration générale vérifier que le fichier global.jelly a bien été ajouté, on coche la case. Maintenant configurons un projet au hasard et ajoutons une étape pré build, nous aperçevons un item s'appelant « say hello world », essayons et remplissons le champ de texte. Compilons le projet et regardons la sortie console, au début nous voyons affiché notre texte.
Nous avons donc un plugin qui permet de dire quelque chose en début ou fin de build.

4.4 Implémentation de la base du nouveau plugin

Nous avons donc un plugin qui étend le comportement lors des builds que Jenkins exécute, ceci grâce au point d'extension Builder.

8. à l'emplacement du POM.xml
9. port 8080

FIGURE 4.3 – Jenkins à l'exécution de la commande maven qui l'encapsule

Jenkins possède beaucoup de points d'extensions et à cette étape il faut déterminer lequel, ainsi que quelle classe étendre ! Dans le cas où aucun point d'extension ne correspond, il est très facile d'en créer un en suivant la documention. Autrement, nous pouvons nous inspirer du plugin Radiator est regarder quelle classe il étend.
En parcourant la liste des points d'extensions disponibles nous pouvons trouver *View*, dont l'une de ses implémentation est *ListView* ; sa description étant « Displays Jobs in a flat list view ». Notre objectif étant d'afficher les Jobs, cette solution semble correspondre. De plus, en observant la javadoc on peut trouver la méthode nous donnant accès aux projet associés à la vue (figure 4.4 page 30).

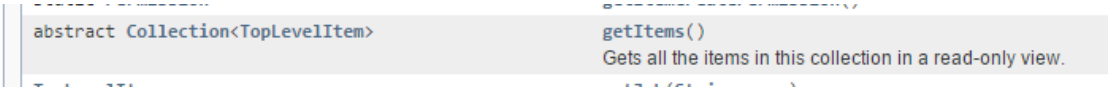


FIGURE 4.4 – Extrait de la javadoc de la classe ListView

4.4.1 Import du code dans l'IDE

J'ai expérimenté deux IDE pour développer le plugin : Eclipse puis NetBeans.
En utilisant Eclipse j'ai rencontré beaucoup de problèmes liés aux configurations maven et d'Eclipse, de telle sorte qu'à un certain moment je me trouvais obligé de compiler le code via Jenkins, de récupérer le .hpi en local pour l'intégrer ensuite à Jenkins ! Le cycle de développement était le suivant :

1. Implémentation du code sous Eclipse
2. Compilation du code grâce à maven (ou Jenkins !) et génération du fichier .hpi
3. Désinstallation du plugin de Jenkins + redémarrage
4. Installation du nouveau plugin sur Jenkins + redémarrage
5. Test du plugin

L'utilisation de Netbeans (et de son plugin Jenkins) évite de passer par toutes ses étapes car il suffit de cliquer sur run pour exécuter le plugin et sur debug pour le debugger. Cela facilite l'implémentation du code et permet surtout de se focaliser sur les problèmes liés au plugin plutôt qu'à ceux liés à ses outils de développement.

4.5 Investigation et résolution

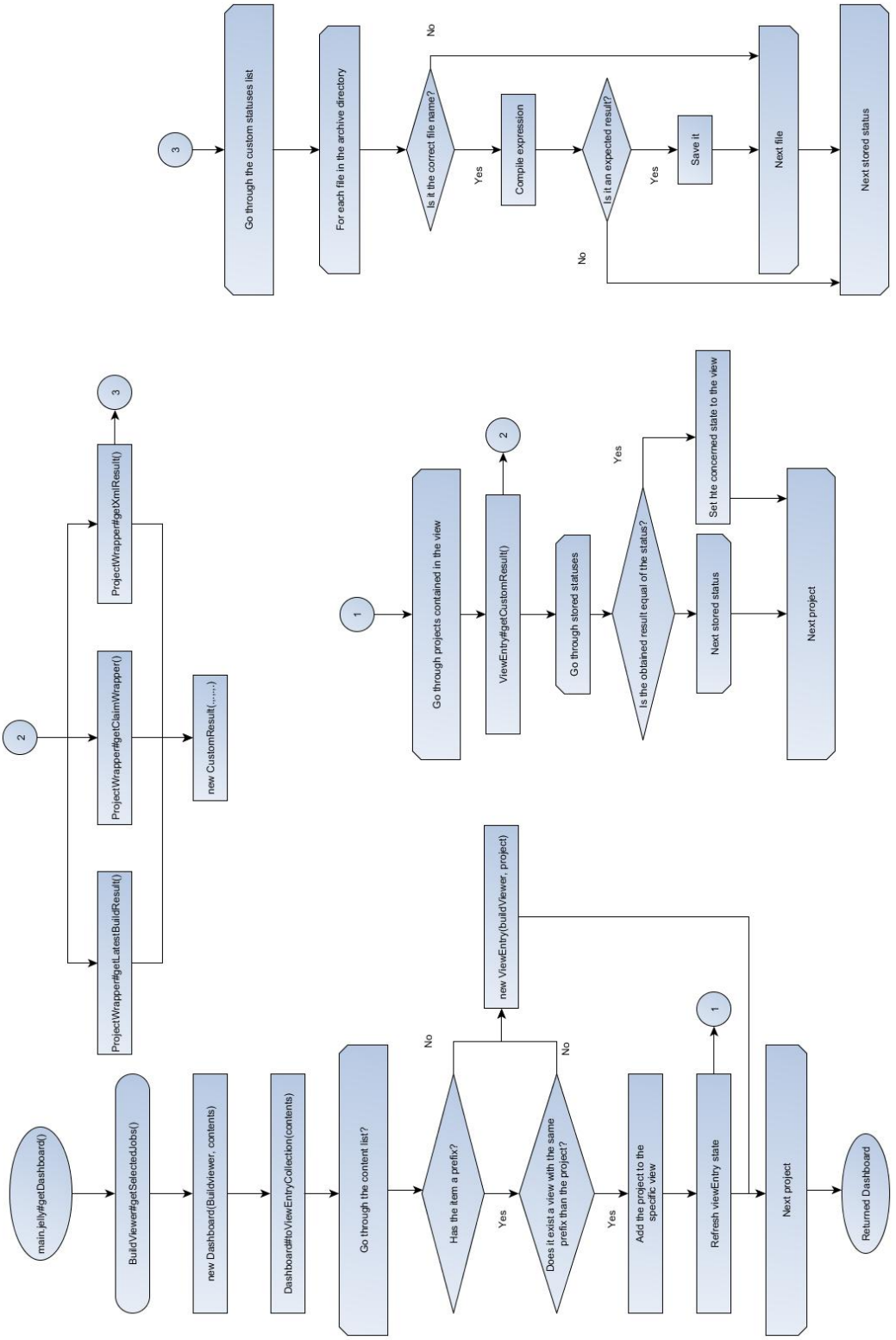


FIGURE 4.5 – Algorithme de création du Dashboard

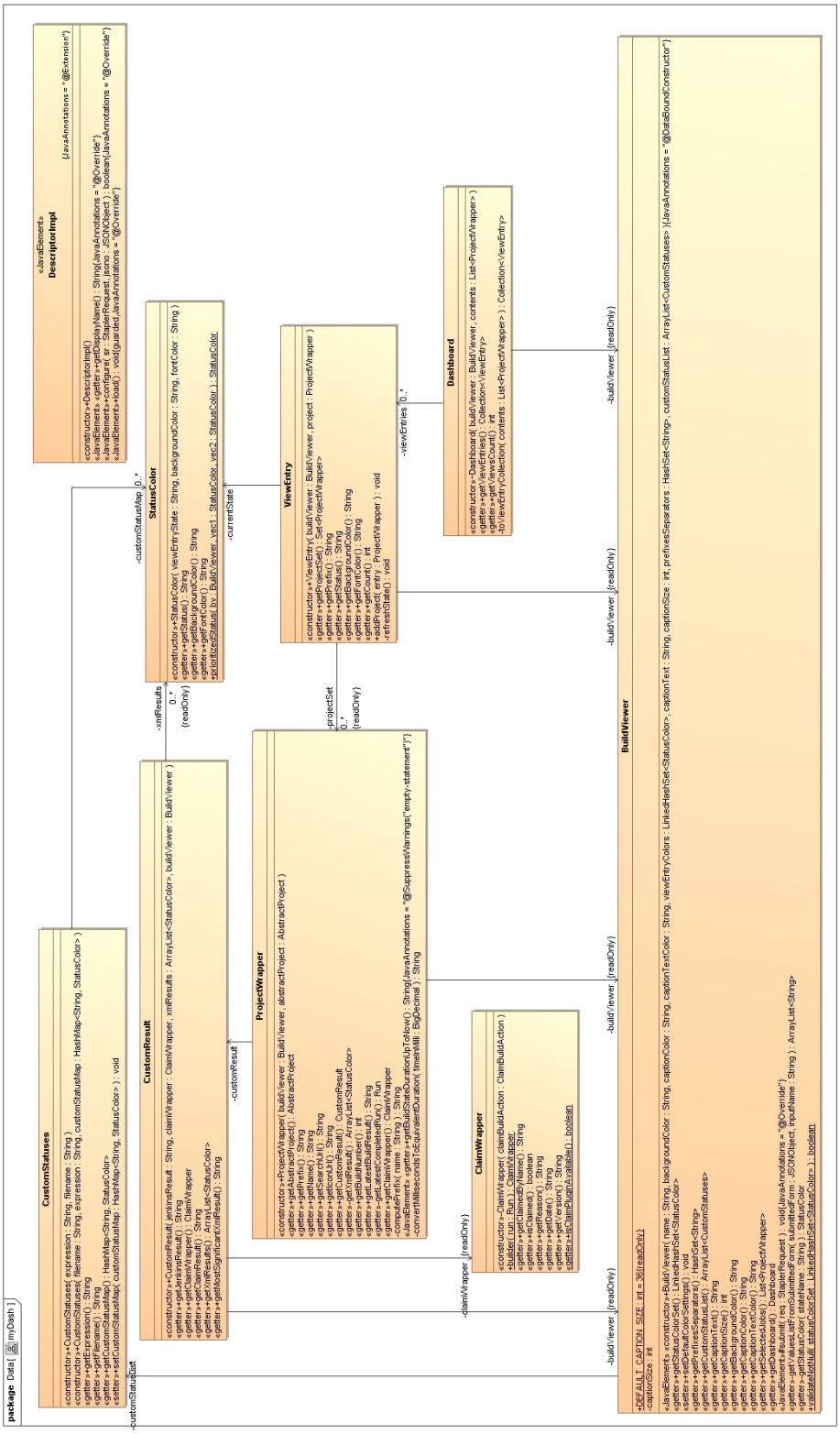


FIGURE 4.6 – Diagramme de classe du plugin réalisé

Conclusion

Mon contrat professionnel à SAP fût une expérience remarquable tant par la valeur de l'équipe que j'ai intégré que par les missions que je me suis vu effectuer.

M'a permi de découvrir l'environnement complexe d'une multinationnale et les outils mis à disposition des employés pour permettre une bonne communication au sein de l'entreprise. J'ai pu me rendre compte de l'ampleur d'un projet comme WebI ainsi que toutes les ressources impliquées dans la conduite de celui-ci.

Les apports de ma mission pour l'entreprise ? L'avenir du projet dans l'entreprise. Un prolongement éventuel ? Une proposition refusée

Index

Table des matières