

Intervention sur le développement d'un client riche Java pour l'assistance à la surveillance des centrales nucléaires



Etudiant : Fobis Samuel
Tuteur professionnel : Régis Ferrier-Tarin
Tuteur universitaire : Roland Ducournau
Université Montpellier II

Rapport de stage professionnel soumis pour l'obtention de
Maîtrise Informatique
Spécialité : AIGLE

22 juin 2016

A mon père pour m'avoir toujours fait chercher ce qu'il aurait pu se
contenter de me donner.

A ma mère pour m'avoir toujours donné le courage de continuer à
chercher.

Remerciements

Étant l'aboutissement de cinq années, un stage de fin d'études constitue une étape importante dans la vie professionnelle, permettant de faire mûrir ses efforts, et de faire germer de nouvelles idées.

Je remercie Sopra Steria Montpellier et sa directrice, Isabelle Reix Di Biase, ainsi que Sylvain Tisserand, responsable du pôle EMC, de m'avoir donné la chance d'intégrer une de leur équipe.

Mes remerciements s'adressent particulièrement à Régis Ferrier-Tarin pour m'avoir mis à l'aise, encadré et soutenu depuis le premier jour.

Je remercie également Roland Ducournau pour sa bienveillance tout au long de ces cinq mois.

Je remercie enfin la chaleureuse équipe de projet dans laquelle j'ai été affecté pour leur professionnalisme et leur pédagogie, particulièrement M. Torreilles et M. Djeghem, avec qui j'ai passé de nombreux repas à prendre note de leur expérience et conseils.

Synopsis


Ce présent rapport a pour vocation de présenter le déroulement de mon stage de fin d'études au sein de Sopra Steria Montpellier.

L'objectif principal a été de participer aux différentes étapes d'un projet, ayant pour objectif de fournir une application permettant aux inspecteurs réalisant les surveillances des réacteurs et autres composants sensibles dans les centrales nucléaires, de les assister dans cette démarche.

Mon intervention eut pour but le développement sur la partie client riche du projet, principalement sur l'aspect ergonomique.

Ce document a été remis à M. Ducournau et M. Ferrier-Tarin, dans le cadre du stage *Mission de développement Java JEE H/F* le 01 juillet 2016. Ce document pouvant contenir des informations confidentielles, il ne saurait être communiqué à des tiers sans l'accord explicite et écrit de Sopra Steria.

Table des matières

1	Introduction	1
2	Entreprise	2
2.1	Sopra Steria	2
2.1.1	Présentation du groupe	2
2.1.2	Marchés et répartition mondiale	3
2.1.3	La division Sud-Est	5
2.1.4	L'agence de Montpellier	6
3	Le projet - GIPSI	7
3.1	Les acteurs	7
3.2	Contexte	9
3.2.1	Le nucléaire	9
3.2.2	État des lieux	9
3.2.3	Les constats de GIPSI v6	10
3.3	Présentation de GIPSI v7	11
3.3.1	Déroulement d'une surveillance	11
3.3.2	Architecture de la solution	11
3.3.3	Objectifs	13
3.4	Équipe et gestion de projet	14
3.4.1	Ligne temporelle	14
3.4.2	Equipe	14
4	Intervention	15
4.1	Technologies utilisées sur le client riche	15
4.1.1	Apache Maven maven	15
4.1.2	JavaFX 	15
4.1.3	HyperSQL DataBase HyperSQL	16
4.1.4	Google Guice Guice	16



4.1.5	Google Guava  guava	17
4.1.6	Jersey  Jersey	17
4.2	Méthodologie de travail	18
4.2.1	Poste et outils de travail	18
4.2.2	Ticket : JIRA et HPQC	18
4.2.3	Phases de tests	20
4.2.4	Suivi d'activité	22
4.3	Ergonomie	22
4.3.1	Définition et intérêt	22
4.3.2	Analyse des architectures	23
4.3.3	Application du MVP sur GIPSI	28
4.4	Intervention	30
4.4.1	Exemple simple de correction d'anomalie	30
4.4.2	Exemple avancé de correction d'anomalie	32
4.4.3	Exemple d'évolution	34
4.4.4	Exemple de proposition de solution et chiffrage	38
4.4.5	Atelier ergonomique	41
5	Bilan & conclusion	42
5.1	Bilan	42
5.2	Conclusion	43
A	Lexique	44
A.1	trigramme/etc	44
A.2	Lexique	44
B	Annexe	45
B.1	Outils	45
B.2	Langages et format de données	45
	Bibliographie (dans l'ordre d'apparition)	46

Table des figures

2.1	<i>Logo du groupe Sopra Steria</i>	2
2.2	<i>Répartition des métiers en 2015</i>	3
2.3	<i>Verticaux stratégiques du groupe</i>	4
2.4	<i>Répartition mondiale</i>	4
2.5	<i>La division Sud-Est</i>	5
2.6	<i>Organigramme de l'agence</i>	6
3.1	<i>Relation entre les différents acteurs</i>	7
3.2	<i>Logo du groupe Électricité de France</i>	8
3.3	<i>Architecture du projet GIPSI v7</i>	12
4.1	<i>Processus de traitement d'une demande</i>	19
4.2	<i>Cycle sur la nature des tickets</i>	20
4.3	<i>Dataset XML</i>	21
4.4	<i>Requêtes SQL</i>	21
4.5	<i>Exemple de formulaire</i>	24
4.6	<i>Deux approches MVP : Passive view et Supervising controller</i>	28
4.7	<i>Architecture MVP associé à JavaFX</i>	29
4.8	<i>Architecture MVP associé à GIPSI</i>	30
4.9	<i>Méthode formatant le texte riche</i>	33
4.10	<i>Listener assurant le séquençage des actions</i>	34
4.11	<i>Diagramme d'état de l'évolution</i>	35
4.12	<i>Exemple de binding des composants dans la vue</i>	36
4.13	<i>Exemple de gestion d'évènement</i>	36
4.14	<i>Exemple de lancement d'une notification</i>	37
4.15	<i>Lancement d'une notification au démarrage de la synchronisation</i> . .	37
4.16	<i>Souscription à un évènement</i>	38
4.17	<i>Proposition initiale</i>	39
4.18	<i>Proposition avancée</i>	40

Chapitre 1

Introduction

Dans le cadre de ma formation Master Informatique spécialité *Architecture et Ingénierie du Logiciel et du Web* à l'université de Montpellier, j'ai effectué mon stage professionnel de fin d'études dans l'agence montpelliéraine de services du numérique Sopra Steria. Ayant choisi cette entreprise pour son haut niveau d'expertise et ses projets imposants, j'ai pu être affecté au sein d'un projet Java pour le compte d'EDF.

Durant ces mois de stage, j'ai pu mettre en pratique les connaissances accumulées tout au long de mon cursus universitaire sur un projet intéressant et formateur, dans une équipe chaleureuse et professionnelle en participant aux différentes étapes d'un projet informatique.

Dans sa première partie, ce document présentera l'environnement dans lequel j'ai évolué, allant de la place de Sopra Steria dans le monde à son agence montpelliéraine.

Dans un second temps, nous détaillerons le projet en lui-même, décrivant ses différents acteurs et son contexte, des besoins du client à la réalisation qui en a été faites par Sopra Steria pour finir sur la gestion du projet et de son équipe.

Enfin, ce rapport décrira dans sa troisième et dernière partie, l'intervention que j'ai réalisé durant ces cinq premiers mois, débutant par une présentation des technologies manipulées pour enchaîner sur les méthodologies de travail, de l'aspect ergonomique sur lequel j'ai principalement travaillé pour finir sur des exemples d'actions réalisées.

Chapitre 2

Entreprise

2.1 Sopra Steria



FIGURE 2.1 – *Logo du groupe Sopra Steria*

2.1.1 Présentation du groupe

Sopra Steria est une entreprise de services du numérique (ESN, anciennement SSII), leader européen proposant un portefeuille d’offres des plus complets : conseil, intégration de systèmes, édition de solutions métier, gestion d’infrastructures informatiques et business process services.

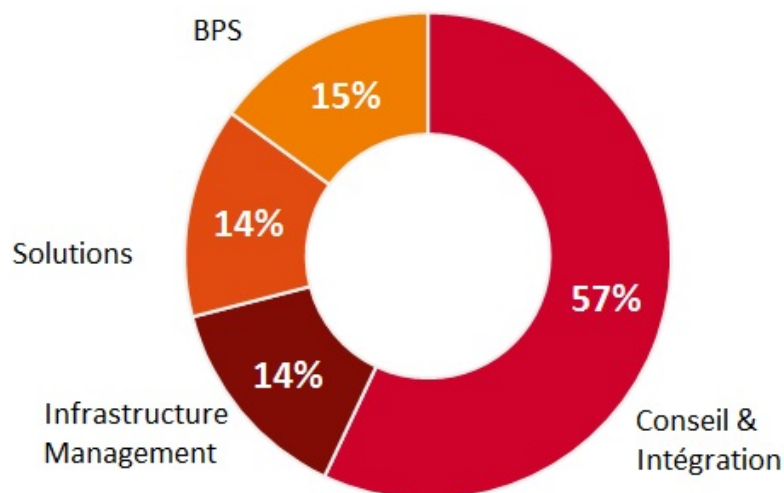
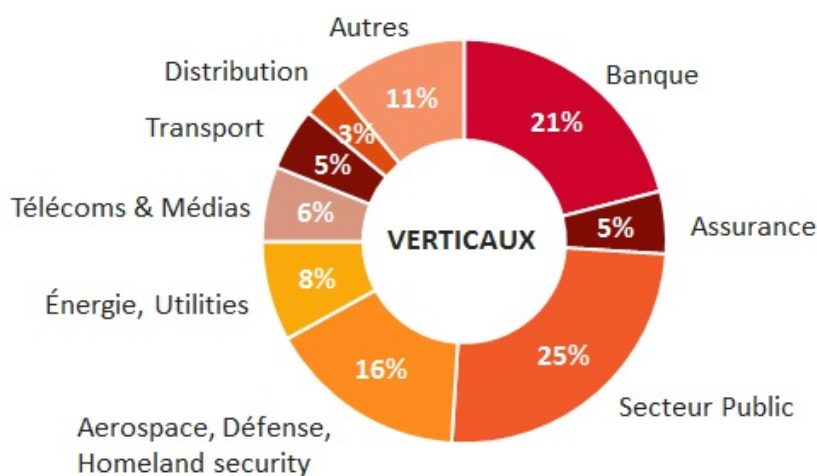


FIGURE 2.2 – Répartition des métiers en 2015

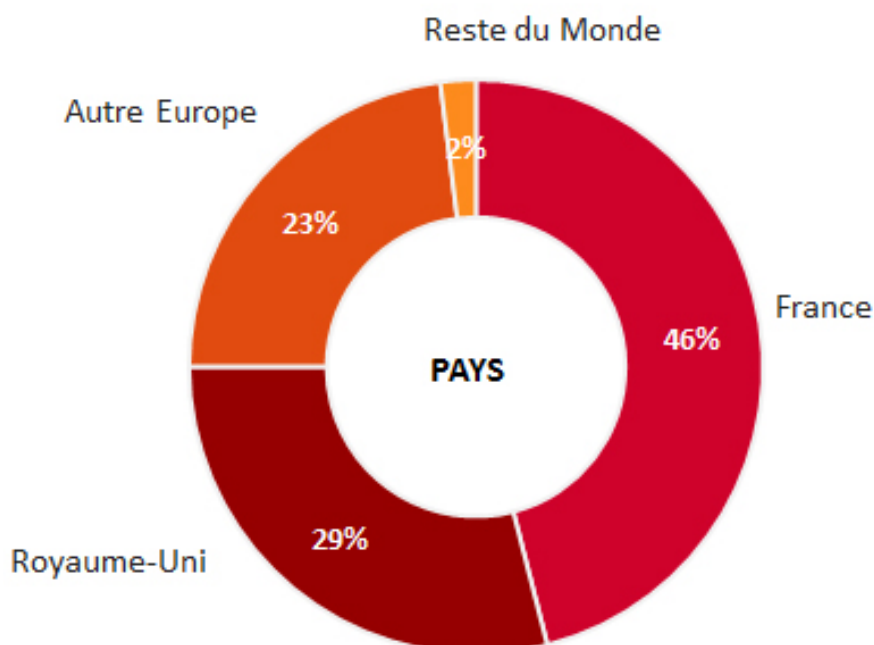
En 2015, Sopra Steria est le fruit de la fusion de deux entreprises françaises de services numériques Sopra et Steria, créées respectivement en 1968 et 1969. Le groupe compte désormais 38 000 collaborateurs répartis dans plus de 20 pays et réalise en 2014 un chiffre d'affaires de 3,4 milliards d'euros, de 3,6 milliards en 2015 et de 913,2 millions sur le premier trimestre 2016.

2.1.2 Marchés et répartition mondiale

Le marché des services financiers et du secteur public constitue plus de 50% du chiffre d'affaires du groupe, mais on le retrouve également dans les transports, la défense ou l'énergie. Les offres proposées visent essentiellement les grands comptes comme le Crédit Agricole, La Poste ou EDF.

FIGURE 2.3 – *Verticaux stratégiques du groupe*

Sopra Steria est implantée dans plus de vingt pays, principalement européens avec la France et le Royaume-Uni, mais aussi en Afrique (Maroc, Caméroun, etc.) et en Asie (Inde et Singapour).

FIGURE 2.4 – *Répartition mondiale*

2.1.3 La division Sud-Est

Sopra Steria est répartie en France en division. L'agence montpelliéraine fait ainsi partie de la division Sud-Est regroupant quatre sites avec plus d'un milliers de collaborateurs.

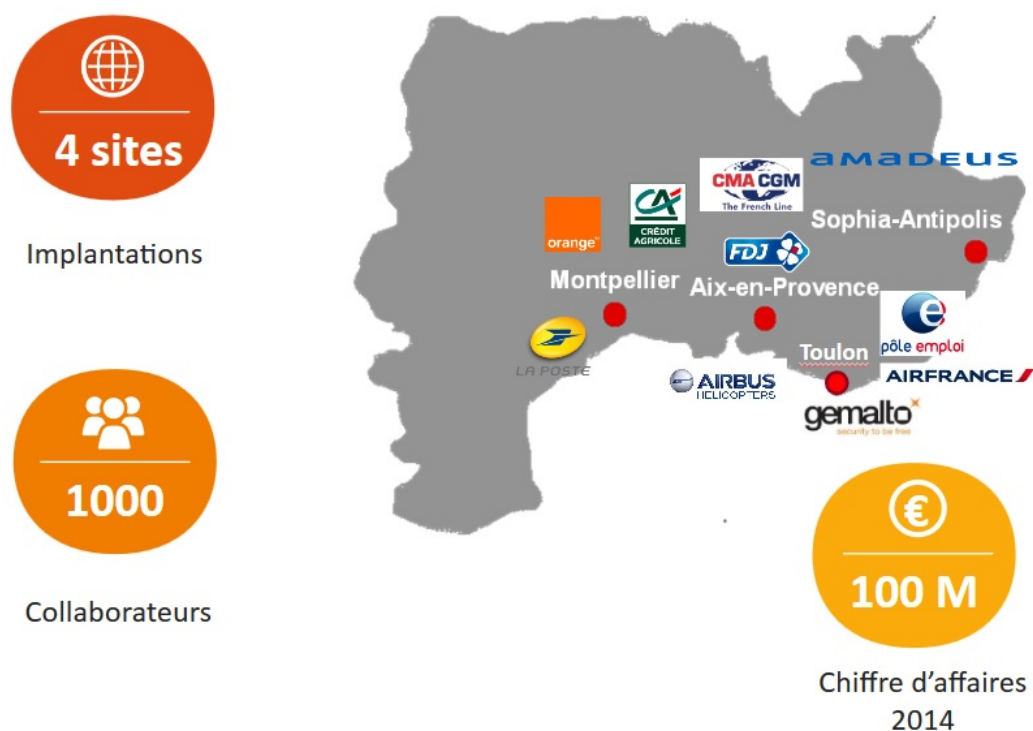


FIGURE 2.5 – *La division Sud-Est*

2.1.4 L'agence de Montpellier

L'agence montpelliéraine est physiquement séparée en deux entités regroupant 180 collaborateurs dans différentes équipes de projets pour de nombreux clients comme La Poste, EDF ou encore Royal-Canin. Elle détient notamment un pôle de Gestion de Contenu d'Entreprise (ECM) située dans l'agence du Parc Club du Millénaire dans lequel évolue le projet GIPSI v7.

Ci-dessous l'organigramme de l'agence de Montpellier :

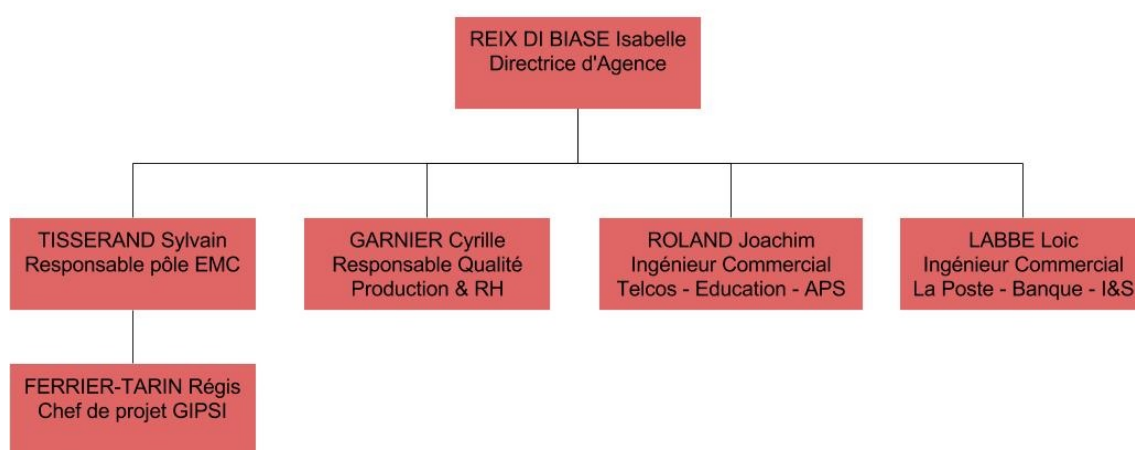


FIGURE 2.6 – *Organigramme de l'agence*

Chapitre 3

Le projet - GIPSI

3.1 Les acteurs

Trois principaux acteurs interagissent au sein du projet GIPSI¹ et forment une hiérarchie en trois couches illustrée ci-dessous :



FIGURE 3.1 – *Relation entre les différents acteurs*

Cette formation assez courante dans les projets informatiques apporte de nombreux avantages. En effet, la maîtrise d'ouvrage (MOA) exprime ses besoins, précise les objectifs et définit le budget et les délais. Elle ne détient pas les compétences pour répondre à ses besoins et va donc déléguer cette tâche à une maîtrise d'œuvre (MOE).

La MOE traduit les besoins de la MOA et assure la production du projet dans le respect des délais et du budget. Cependant, dans certain cas la MOE ne dispose pas non plus des compétences et de la main-d'œuvre, ni du matériel pour la production du projet et va déléguer elle-même cette partie à une autre entité, un intégrateur. Elle ne prendra en charge que la phase de spécification.

1. GIPSI : Gestion Informatisée des Prestations de Surveillance des Industriels

La maîtrise d’ouvrage : EDF

FIGURE 3.2 – *Logo du groupe Électricité de France*

Créé le 8 avril 1946 par Marcel Paul avec le soutien de Maurice Thorez, Électricité de France (EDF) est aujourd’hui le premier producteur et fournisseur d’électricité en France et dans le monde.

Son chiffre d’affaires en 2015 est de 72,9 milliard d’euros, elle emploie environ 160 000 personnes et dessert plus de 39 millions de clients de par le monde.

EDF représente le client du projet et celui qui a initialement exprimé ses besoins à la MOE.

La maitrise d’œuvre : l’ESIP

La MOE est représentée par l’ESIP². Ce département d’EDF est garant du maintien en condition opérationnelle (MCO) de l’application GIPSI. Bien que la MOE est une sous-entité d’EDF, elle considère dans les faits la MOA comme son client avec un budget attribué et des délais à respecter.

L’intégrateur : Sopra Steria

La troisième et dernière couche est représentée par l’intégrateur, étant un prestataire externe. Un intégrateur s’occupe d’insérer une solution au sein d’un système informatique pré-existant en s’assurant de fournir au client une procédure technique d’intégration (PTI), un document d’exploitation (DEX) et une description de l’architecture technique (DAT).

2. ESIP : Évolution du Système d’Information du Producteur

Après avoir lancé un appel d'offre, l'ESIP a choisi Sopra Steria pour leur déléguer la partie développement et intégration du projet GIPSI v7.

3.2 Contexte

3.2.1 Le nucléaire

La répartition du parc mondial installé d'EDF en 2014 est la suivante :

- Nucléaire : 54%
- Hydraulique : 16%
- Thermique fossile : 16%
- Autres : 14%

Le parc nucléaire français est composé de 58 réacteurs en fonctionnement répartis sur 19 centrales en exploitation construites essentiellement dans les années 1980. Aujourd'hui, alors que la plupart d'entre elles atteignent leur durée de vie initiale de trente ans, EDF compte doubler celle-ci en remplaçant des composants majeurs jugés obsolètes.

Compte tenu de la complexité et du nombre de réacteurs et de leurs composants, EDF a besoin d'outils pour permettre aux inspecteurs de réaliser la surveillance de chaque composant et d'en reporter leur état.

3.2.2 État des lieux

La Division Production Ingénierie (DPI), branche du groupe EDF, est confrontée à des enjeux de sûreté pour son activité de production d'énergie électrique : nucléaire, thermique à flamme et hydraulique (et autres énergies renouvelables), en particulier depuis l'accident nucléaire de Fukushima en 2011.

Définir des programmes de surveillance des industriels est décisif pour la sûreté des installations : ils définissent les procédures de contrôle des fournisseurs de matériel et de services, en particulier pour les Centres Nucléaires de Production d'Électricité (CNPE).

L'application existante, GIPSI v6, est une partie du système informatique d'EDF qui permet d'organiser et de réaliser cette surveillance des fournisseurs et sous-traitants conformément aux programmes de surveillance établis.

Le Centre d'Expertise et d'Inspection dans les Domaines de la Réalisation et de l'Exploitation (CEIDRE) assure en France comme à l'étranger, depuis la phase de fabrication en usine jusqu'à la phase d'exploitation sur site, le suivi et le contrôle des matériels mécaniques, électriques, électromécaniques, combustible et de génie civil. Elle représente la maîtrise d'ouvrage de cette application.

En plus du CEIDRE, l'application GIPSI v6 est utilisée par une autre entité d'EDF, le CNEN (Centre National d'Équipement Nucléaire) en charge de la surveillance des différentes étapes de construction d'un nouveau site nucléaire, en France comme à l'étranger, jusqu'à sa livraison pour exploitation.

La maîtrise d'œuvre de cette application est, au sein de la Division Appui Industriel au Producteur (DAIP), l'entité EDF ESIP située à Lyon. Elle assure le pilotage opérationnel de la réalisation, la qualification, l'intégration dans le système d'information et le Maintien en Conditions Opérationnelles (MCO) de l'application GIPSI v6.

L'évolution des enjeux relatifs à la surveillance des industriels fournisseurs des CNPE, en particulier la construction de réacteurs de troisième génération de type EPR³ en France comme à l'étranger, nécessite une évolution du système d'information vers une architecture fonctionnelle, logicielle et technique pérenne.

3.2.3 Les constats de GIPSI v6

Détenant le premier parc nucléaire mondial, EDF entreprend un nouveau programme de réalisation et d'exploitation de réacteurs nucléaires de nouvelle génération EPR.

L'application GIPSI v6, ayant atteint les limites de son architecture actuelle (sous Lotus Notes), elle ne permet plus de faire face pleinement à ces nouveaux enjeux compte tenu de plusieurs facteurs :

- Obsolescence technologique
- Interface contraignante
- Incohérence des données

3. EPR : Réacteur Pressurisé Européen conçu et développé par Areva Nuclear Power au cours des années 1990 et 2000

3.3 Présentation de GIPSI v7

3.3.1 Déroulement d'une surveillance

GIPSI v7 devra permettre, comme son prédécesseur, d'assister les inspecteurs lors de la préparation et la réalisation des surveillances des fournisseurs, soit lors de la fabrication des composants nucléaires, soit lors de l'exploitation d'une centrale sur site.

Lors d'une surveillance, un inspecteur peut être amené à être déconnecté du réseau EDF de quelques jours à plusieurs mois, les inspections se déroulant parfois à l'étranger, au sein de centrales nucléaires où aucun réseau n'est accessible. Une surveillance est composée de trois phases distinctes : la préparation en rapatriant les données nécessaires sur le poste local de l'inspecteur, l'inspection en elle-même en étant déconnecté et enfin la synchronisation à la reconnection.

Les données sont ensuite traitées et conservées de nombreuses années.

3.3.2 Architecture de la solution

En analysant le contexte et le déroulement d'une surveillance, on en distingue rapidement les contraintes :

- Migration de données d'anciennes versions
- Utilisation connectée
- Utilisation déconnectée
- Synchronisation de données
- Traitement des données

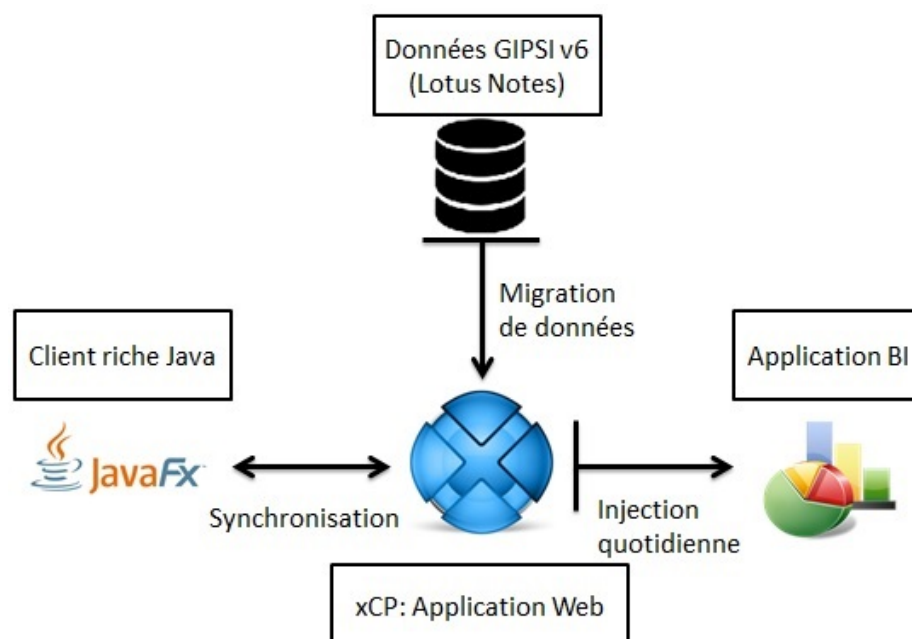


FIGURE 3.3 – Architecture du projet GIPSI v7

Le projet Gipsi v7 se compose ainsi de quatre phases expliquées ci-après.

Migration de données

La migration permet de transférer les données de l'ancienne version et ainsi conserver leur intégrité dans l'utilisation de la nouvelle. Cette opération va se dérouler en plusieurs phases distinctes permettant d'identifier les structures de données à migrer, de les traiter pour enfin les injecter dans xCP pour qu'elles puissent être utilisées. Ces phases seront guidées par des outils permettant de vérifier et corriger si nécessaire ces données.

Documentum et xCP : préparation de la surveillance

Documentum est un gestionnaire de contenu d'entreprise (ECM⁴) appartenant à *EMC Corporation* permettant de gérer l'ensemble des contenus d'une organisation. Il s'agit de prendre en compte sous forme électronique les informations qui ne sont pas structurées, comme les documents électroniques, par opposition à celles déjà structurées dans les bases de données. Elle comprend les phases de création/capture, stockage, indexation, gestion, nettoyage, diffusion, publication, recherche et archivage,

4. ECM : Enterprise Content Management

en faisant le lien du contenu avec les processus métier.

*xCP*⁵ est une plate-forme créée par *EMC Corporation* pour *Documentum*, fournissant de nombreux outils permettant l'élaboration de l'interface d'un client web, ainsi que des processus métiers. Ce client permettra aux utilisateurs de consulter un plus large éventail de données que sur le client riche mais aussi de permettre aux inspecteurs de préparer leur surveillance.

Client Riche : GIPSI Nomade : réalisation de la surveillance

Une fois que les inspecteurs ont préparés leur surveillance sur le client web, ils rapatrient les données sur le client riche. Développé en Java et lié à une base de données locale gérée par PostgreSQL, ce client permettra de réaliser la surveillance tout en étant déconnecté. Il devra néanmoins permettre de re-synchroniser les données avec les changements effectués en local comme sur le réseau lors de la reconnection.

Business Intelligence

Régulièrement, les données sont transmises à une application de Business Intelligence (BI) permettant de les traiter et d'en produire des statistiques. Cette application permettra de révéler l'état général du parc nucléaire et par exemple, de connaître quel composant a tendance à être défectueux pour ainsi effectuer une action corrective avec son fournisseur.

3.3.3 Objectifs

Les principaux gains apportés par cette version de GIPSI seront :

- Ergonomie améliorée
- Intégrité et cohérence des données
- Fiabilité et sécurité
- Adapté à un plus large éventail d'activités (nouveaux réacteurs EPR)

5. xCP : xCelerated Composition Platform

3.4 Équipe et gestion de projet

3.4.1 Ligne temporelle

GIPSI v7 a vu le jour en juillet 2013 suivi d'une phase de cadrage (spécification de l'application, de la migration et du BI) qui s'est étendu jusqu'en mars 2014 où s'est tenu le chiffrage du *build*. S'en est suivi le développement du client web pour une mise en production prévu en mars 2015, qui s'est ensuite vue reporté en septembre. Pour cause d'une remise en cause de la solution et de retards de planning des équipes EDF, un point d'arrêt a été effectué et la mise en production ré-estimée à décembre 2016.

3.4.2 Equipe

Au gré de l'activité, le projet s'est vu subir un roulement d'effectif. Au cours des cinq mois de stage, treize personnes ont participé à son avancement, dont sept de manière permanente.

L'équipe est composée d'un chef de projet, quatre membres exclusivement sur le développement du client riche, trois sur la partie xCP et un autre sur la migration des données. Ajouté à cela, trois responsables technique se sont succédés et un dernier membre jouant le rôle de tampon suivant l'activité entre le client riche et le client web.

Chapitre 4

Intervention

Ce dernier chapitre présentera concrètement le travail et les recherches accomplis durant ces cinq premiers mois de stage sur le client riche, partant des technologies utilisées à la méthodologie de travail en passant ensuite par mon analyse de l'ergonomie et de son importance, des différentes architectures existantes et de celle appliquée à GIPSI pour terminer sur des exemples d'actions effectuées.

4.1 Technologies utilisées sur le client riche

4.1.1 Apache Maven **maven**

Apache Maven est un outil pour la gestion et l'automatisation de production des projets logiciels Java créé en 2004, similaire à *Apache Ant*. L'objectif recherché est comparable au système *Make* sous Unix : produire un logiciel à partir de ses sources, en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication.

Maven utilise un concept connu sous le nom de *Project Object Model (POM)* afin de décrire un projet logiciel, ses dépendances avec des modules externes et composants, l'ordre à suivre pour sa production et les plugins nécessaires.

Il s'intègre parfaitement à *Eclipse* et a servi d'inspiration à d'autres outils comme *Gradle*.

4.1.2 JavaFX

JavaFX est une technologie créée en 2008 par Sun Microsystems appartenant désormais à Oracle, permettant de produire aussi bien des applications de bureau que des applications internet riche supportées par une grande variété de supports.

Un des principaux buts de son développement est le remplacement de l'API¹ *Swing* comme standard de bibliothèque graphique pour Java.

À ses débuts, cette librairie était utilisée via des scripts pour finalement être présente par défaut dans le JRE/JDK. Elle fut aussi nativement prévue pour être utilisée sur des systèmes embarqués, idée abandonnée par Oracle au passage de version 2.0, mais reprise récemment pour être intégrée dans *Java SE Embedded 8*.

JavaFX contient un éventail d'outils variés, notamment pour les médias audio et vidéo, le graphisme 2D et 3D ou la programmation Web. Des projets libres complètent cette librairie en fournissant des composants de haute qualité absents de *JavaFX* proprement dit, comme utilisé sur GIPSI : *ControlsFX*.

L'interface graphique peut être définie grâce au code Java mais aussi via un format de données basé sur le XML - le FXML - facilitant ainsi la construction des vues puisque la structure du XML est similaire à la structure d'une scène graphique de *JavaFX*. Enfin, une bonne pratique est d'utiliser du code CSS² pour modifier l'apparence des composants.

4.1.3 HyperSQL DataBase

HSQLDB est un système de gestion de base de données relationnelle écrit en Java, développé et maintenu par « *The HSQL Development Group* » et fondé sur le projet Hypersonic SQL de Thomas Mueller.

Il possède un pilote JDBC³ et supporte une grande partie des standards SQL-92 et SQL : 2011. Il offre un moteur de base de données rapide et léger (moins de 100ko pour certaines versions) qui propose à la fois une gestion en mémoire vive et sur fichiers. Des versions embarquées et client-serveur sont notamment disponibles et il inclut également plusieurs outils tels qu'un serveur Web minimal ou un gestionnaire en ligne de commande.

Il est très apprécié pour sa taille minuscule, sa capacité à être exécuté entièrement en mémoire et sa vitesse.

4.1.4 Google Guice

Google Guice (prononcé *Juice*) est un framework open source distribué par *Google* sous licence *Apache* permettant l'injection de dépendance sous Java.

Guice soulage le code ainsi que de remplacer l'utilisation du « *new* » dans certains

-
1. API : Application Programming Interface
 2. CSS : Cascading Style Sheets
 3. JDBC : Java DataBase Connectivity

cas. Ce module permet de définir des relations entre des interfaces et des classes les implémentant qui seront ensuite injectées dans des constructeurs, méthodes ou attributs via l'annotation « *@Inject* ». Quand plus d'une implémentation de la même interface est désirée, le développeur a la possibilité de créer ses propres annotations identifiant une implémentation en particulier.

Cette manière de procéder permet de ne traiter l'injection de dépendance qu'en Java, sans avoir à gérer de fichiers XML et améliore les performances en supprimant la nécessité du parsing XML.

Guice a été le premier framework générique pour l'injection utilisant les annotations en 2008.

4.1.5 Google Guava

Le framework open source *Guava* contient plusieurs librairies distribuées par *Google* et motivé par l'introduction des génériques dans le JDK 1.5¹. *Guava* offre de nombreux outils fournissant des fonctionnalités pratiques et avantageuses comme la programmation fonctionnelle, la gestion du cache, des immutables et du hachage. Une analyse effectuée en 2015 place *Guava* en quatrième position des bibliothèques les plus utilisées par les projets les plus populaire du site *GitHub*.

Ce framework est notamment utilisé dans GIPSI pour son objet « *EventBus* » permettant de lancer des notifications à travers toute l'application, où d'autres objets étant liés à ces événements via l'annotation « *@Subscribe* » peuvent réagir.

4.1.6 Jersey

Jersey est lui aussi un framework open source mais développé par Oracle. Il est construit, assemblé et installé à l'aide de l'outil d'automatisation *Maven*.

Cet outil, écrit en Java, permet de développer des services web selon l'architecture REST⁴ suivant les spécifications de JAX-RS⁵. Il est composé de plusieurs composants :

- *Core Server* : Production de services RESTful basés sur les annotations
- *Core Client* : Soulage la communication entre services REST
- Un support *JAXB*
- Un support *JSON*

4. REST : Representational state transfer

5. JAX-RS : Java API for RESTful Web Services

— Un module d'intégration, notamment pour *Guice*

Offrant sa propre API pour encore plus simplifier l'utilisation de service REST et le développement d'application client, *Jersey* propose de nombreuses extensions permettant de répondre aux besoins particuliers des développeurs.

4.2 Méthodologie de travail

4.2.1 Poste et outils de travail

Durant ces cinq mois de stage, il a fallu utiliser de nombreux outils, certains étant familier, et d'autres inconnus. Les développements sont effectués grâce à l'IDE ⁶ *Eclipse* où plusieurs Plugins permettent de faciliter cette tâche, que ce soit pour l'organisation du travail avec *Mylyn*, permettant de changer rapidement de contexte, particulièrement utile avec l'utilisation de JIRA, *Subclipse* pour la prise en charge du *versionning* ⁷ ou encore *SonarQube* pour surveiller la qualité du code.

4.2.2 Ticket : JIRA et HPQC

HPQC ⁸ est un logiciel de gestion de qualité créé par *Mercury Interactive* et racheté par la firme *HP* offrant la possibilité de gérer les processus de tests d'application numériques[2]. Elle est utilisée par la MOA, la MOE et la TMA pour communiquer entre elles des points évolutifs ou correctifs.

Introduit au dessus, *JIRA* est un système de suivi de bugs, de gestion des incidents, et de gestion de projets développé par *Atlassian*[3]. Il est utilisé au sein de Sopra Steria pour suivre les corrections ou évolutions en cours. La plupart du temps les tickets *JIRA* font suite à un ticket *HPQC*, mais peuvent être parfois initiés après la découverte d'une anomalie par l'équipe de projet elle-même. Un ticket *JIRA* permet de tracer l'avancement, de répertorier les actions déjà effectuées et celles qui restent ainsi à faire.

Que ce soit pour des corrections ou des évolutions, la démarche est la même. La MOA ou MOE transmet un ticket via *HPQC* qui trace ainsi le processus. Une fois

6. IDE : Environnement de Développement

7. Versionning : Démarche consistant à maintenir l'intégrité des versions d'un ou plusieurs fichiers

8. HPQC : HP Quality Center

validée par la TMA, ce ticket *HPQC* est converti en un ticket *JIRA* et attribué à un membre de l'équipe, suivant son occupation actuelle, son affectation sur le projet et son historique sur le traitement de ce type de ticket. Une fois traité, le développeur passe la main aux testeurs tout en ayant préliminairement noté en commentaire tout ce qui peut leur être utile, que ce soit le processus mis en place, les impacts possibles ou encore même des scénarios de tests

Un ticket est ainsi traité via le cycle en V suivant :

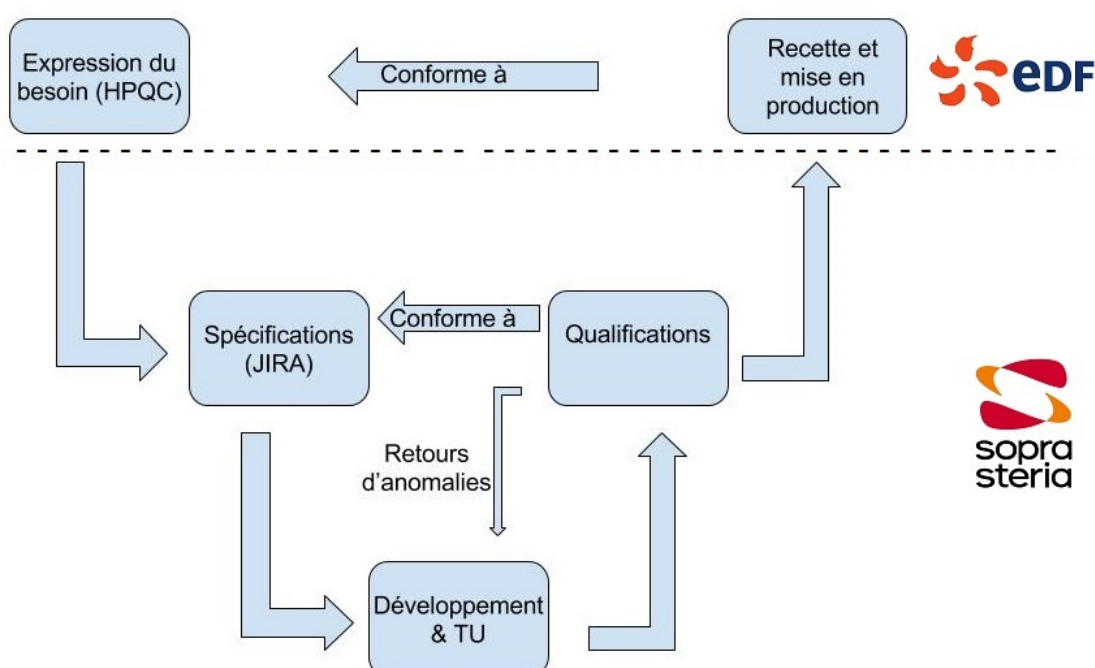
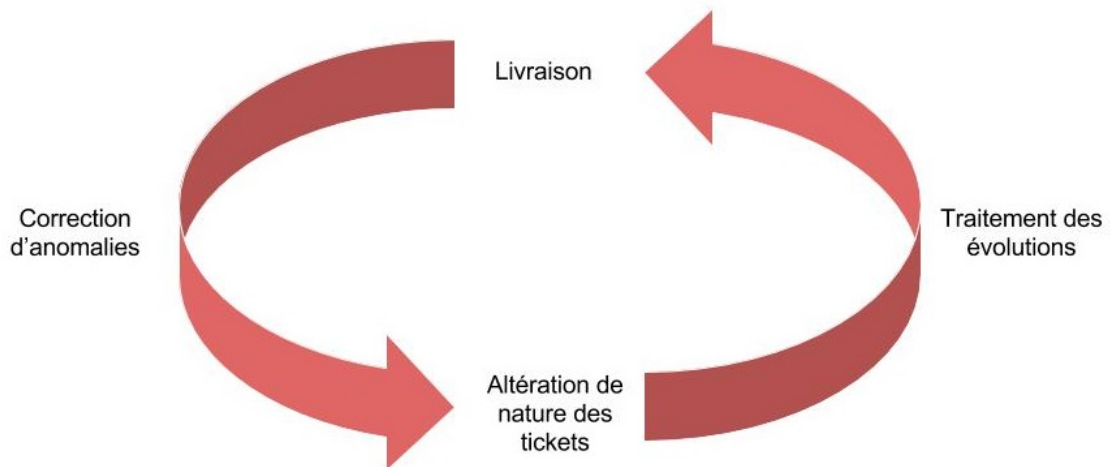


FIGURE 4.1 – *Processus de traitement d'une demande*

La nature des tickets varie et suit elle-aussi un cycle dépendant des livraisons, comme illustré ci-dessous :

FIGURE 4.2 – *Cycle sur la nature des tickets*

En effet, suite aux différentes livraisons, lorsque le client effectue des tests à plus grande échelle, on constate que le nombre de tickets concernant des anomalies augmentent significativement. Quand la majorité des anomalies a été corrigée, la nature des tickets changent progressivement en évolution, permettant de garder une activité constante sur le projet.

Enfin, lors des tests de qualification, des statistiques sont faites permettant d'analyser la progression des développeurs. Au cours de ces cinq mois, j'ai traité de nombreux tickets avec une constante diminution du taux de retours après tests, allant jusqu'à moins de 13% de retours négatifs sur les seize derniers traités, tout en sachant qu'ils étaient de plus en plus complexe.

4.2.3 Phases de tests

Le test unitaire (TU ou UT pour *Unit testing*) est une démarche permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion de programme et fait partie de la routine d'un développeur. Il s'agit pour celui-ci de tester un module, indépendamment du reste du programme, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances.

Un des avantages du test unitaire est qu'il permet de détecter rapidement les régressions. En effet après avoir modifié un bout de code, si les tests précédemment élaborés viennent à échouer, on sait que l'on vient d'en inclure une.

Lorsque l'on travaille sur l'ergonomie d'une application, il est important lors de la conception de séparer la logique de l'implémentation de l'interface graphique et c'est pour cette raison que l'on s'inspire des modèles d'architectures ayant déjà fait leur preuve comme le *design pattern* MVC⁹. Cette séparation réduit considérablement l'utilité de tests unitaires.

Néanmoins, lorsque cela fut nécessaire, il a fallu établir des procédures de tests. Pour ce faire, une extension du framework *JUnit* a été utilisée, *DBunit*, permettant entre autres d'initialiser des jeux de données grâce à des fichiers XML et d'extraire le contenu d'une base pour y faire des assertions.

La première chose à faire est de construire des fichiers XML appelés *Dataset* contenant nos objets. Cette démarche permet entre autres de pouvoir réutiliser les mêmes objets pour différents tests évitant d'en recréer à chaque fois. Ces jeux ont la forme ci-dessous :

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <AbstractLocalEntity id="1" modifie="false" modifiable="false"
    modifiableEnBase="false" dType="AEntity" />
  <AbstractLocalEntity id="2" modifie="false" modifiable="false"
    modifiableEnBase="false" dType="BEntity" />
  <AbstractLocalEntity id="3" modifie="false" modifiable="false"
    modifiableEnBase="false" dType="CEntity" />
</dataset>
```

FIGURE 4.3 – *Dataset XML*

Ce qui correspond à écrire les requêtes SQL suivantes :

```
INSERT INTO AbstractLocalEntity (id, modifie, modifiable, modifiableEnBase, dType)
VALUES ('1', 'false', 'false', 'false', 'AEntity');
INSERT INTO AbstractLocalEntity (id, modifie, modifiable, modifiableEnBase, dType)
VALUES ('1', 'false', 'false', 'false', 'BEntity');
INSERT INTO AbstractLocalEntity (id, modifie, modifiable, modifiableEnBase, dType)
VALUES ('1', 'false', 'false', 'false', 'CEntity');
```

FIGURE 4.4 – *Requêtes SQL*

Une fois défini et injecté, on peut commencer la réalisation du test sur ces objets.

9. MVC : Modèle-Vue-Contrôleur

4.2.4 Suivi d'activité

Le suivi d'activité permet de suivre l'avancement de chaque membre du projet, mais aussi au chef de projet d'avoir une visibilité sur les ressources disponibles. Chaque fin de semaine, tous les membres du groupe doivent remplir un document récapitulant les tâches accomplies au cours des 5 jours ouvrés précédents et le chiffrage de celles qu'il reste à faire.

Régulièrement, des réunions s'organisent pour faire l'état général des différentes sous-parties du projet, permettant de connaître l'état d'avancement de chacun et les possibles blocages. Chaque membre doit être en permanence conscient de sa charge de travail qui lui est attribué, et d'avertir le chef de projet si celle-ci devient trop importante.

Avoir des développeurs sachant reconnaître des situations difficiles et de les chiffrer en temps et en degré de risque est indispensable si l'on veut que la gestion du projet se déroule correctement.

4.3 Ergonomie

4.3.1 Définition et intérêt

L'ergonomie est l'étude de la relation entre l'homme et ses moyens, représentée en informatique par l'interaction homme-machine. Elle est souvent associée à deux critères :

- L'utilité : répondre au(x) besoin(s) et être pertinente
- L'utilisabilité : « degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec, efficience et satisfaction, dans un contexte d'utilisation spécifié » (norme ISO 9241)

Souvent délaissée par les développeurs au profit des aspects fonctionnels, on a souvent remarqué que malgré le fait qu'une application soit fonctionnellement riche, si l'utilisateur final rencontre sans cesse des difficultés en l'utilisant, il l'abandonnera au profit d'une autre.

À titre d'exemple, en 1994, la seconde version de la base de données Vax Rally corrige 20 des 75 problèmes d'utilisabilité identifiés sur la première version. Cette nouvelle mouture du produit enregistre un gain de 80% sur les bénéfices par rapport à la précédente. Ce résultat est supérieur

de plus de 66% aux prévisions de ventes.

En 1998, IBM a revu le design d'une partie de ses sites selon des principes ergonomiques simples tels que l'homogénéité de présentation et l'accès rapide aux pages les plus fréquemment utilisées. En mars 1999, dans le mois qui a suivi le redémarrage, le trafic a augmenté de 120% sur le site de commerce électronique ShopIBM et les ventes ont grimpé de 400%.[1]

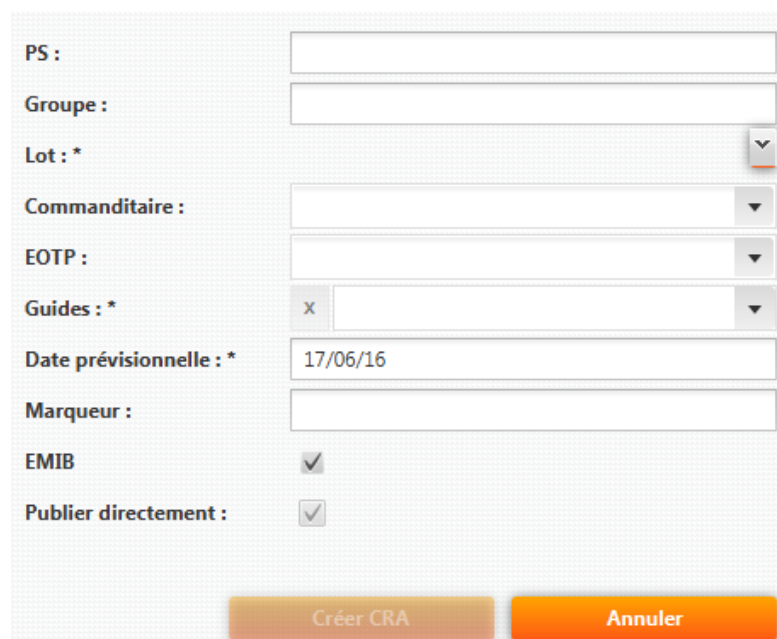
4.3.2 Analyse des architectures

Dans la partie qui va suivre, j'aimerais partager mon interprétation des architectures que j'ai étudié au cours de ce stage pour comprendre celle utilisée pour GIPSI Nomade, espérant apporter un point de repère au lecteur. Avant de commencer, je dois l'avertir qu'embrasser les multiples architectures n'est pas chose facile, notamment parce qu'elles sont en constante évolution et interprétées de bien des manières différentes.

Les patrons d'architecture en informatique sont des modèles de référence utilisés lors de la conception de logiciels complexes, ne prétendant pas être une solution pré-fabriquée mais plutôt une source d'inspiration devant des problèmes récurrents. Bien qu'ils peuvent être inadaptés pour de petits projets, depuis leur création, ils ont su prouver leur efficacité que ce soit en matière de réutilisation et d'adaptabilité du code, de sécurité ou de tests sur des projets ambitieux, offrant ainsi un cadre normalisé pour structurer une application et facilitant aussi le dialogue entre concepteurs.

Architecture basique

Je débiterai par décrire une architecture basique, notamment utilisée pour le développement des environnements client-serveur dans les années 90. Pour illustrer ce cas, prenons un formulaire de GIPSI Nomade considérant être implémenté avec cette architecture :



The image shows a web form with the following elements:

- PS :** A text input field.
- Groupe :** A text input field.
- Lot : *** A dropdown menu with a downward arrow.
- Commanditaire :** A dropdown menu with a downward arrow.
- EOTP :** A dropdown menu with a downward arrow.
- Guides : *** A text input field with a small 'x' icon on the left and a dropdown arrow on the right.
- Date prévisionnelle : *** A text input field containing the value "17/06/16".
- Marqueur :** A text input field.
- EMIB** A checkbox that is checked.
- Publier directement :** A checkbox that is checked.
- At the bottom, there are two orange buttons: "Créer CRA" and "Annuler".

FIGURE 4.5 – *Exemple de formulaire*

Si l'on regarde simplement ce formulaire sans en connaître l'implémentation, on remarque qu'il utilise des composants génériques disposés de manière spécifique à notre application. Les deux caractéristiques de cette scène sont :

- La forme qui définit l'arrangement des composants dans la scène
- Le fond qui implémente le comportement de ces composants

La plupart des bibliothèques graphiques amène le développeur à utiliser des éditeurs graphique pour s'occuper de la forme. Cela peut s'avérer utile mais ce n'est pas toujours la meilleure manière de procéder.

Les composants affichent des données provenant d'une autre source, comme par exemple une base de données. Les données extraites sont alors présentes en trois copies : une copie persistante dans la base, une temporaire en mémoire, et la dernière est celle affichée à l'écran. Il est alors primordial que la copie en mémoire soit identique à celle affichée.

Conserver cette synchronisation est une tâche importante qui peut être prise en charge par l'utilisation des *Data Bindings*. L'idée est que quel que soit le changement fait sur l'affichage, ces modifications seront reportées sur les autres copies. L'utilisation de ce fonctionnement résout de nombreux problèmes mais certains cas restent problématiques, comme par exemple sur notre formulaire, de modifier les valeurs d'une des

listes déroulantes en fonction de la sélection d'une autre. Ce traitement ne pouvant être effectué grâce aux *Data Bindings*, il faudra alors placer cette logique directement dans la scène.

Pour que cela fonctionne, il faut que celle-ci soit alertée du changement sur le composant en question. Plusieurs manières sont possibles pour résoudre ce problème mais la plus utilisée est la notion d'évènement. Chaque composant peut lancer des évènements et tout autres peut s'y abonner, simulant en quelque sorte le patron observateur mais cela est géré par les composants eux-même. Dans l'exemple précédent, la scène qui est elle aussi un composant, doit se lier aux évènements lancés par la liste déroulante. Une fois le processus implémenté, on peut réaliser ce que l'on souhaite tout en propageant les modifications sur les données en mémoire.

Cette architecture peut se résumer :

- Au développement d'application utilisant des composants génériques
- La scène observe les composants et réagit aux évènements dont elle s'est abonnée
- L'édition de données est prise en charge par les *Data Bindings*
- Les traitements logiques complexes sont réalisés au cœur de la scène

Modèle-Vue-Contrôleur

Étant probablement le patron le plus employé dans le développement de GUI¹⁰, il est aussi celui étant le plus détourné sur les analyses qui lui sont faites.

Issue des travaux de Trygve Reenskaug en 1978-1979, il est important de se rappeler qu'à cette époque les interfaces graphiques n'étaient pas courante, et ce fut l'un des premiers essais concrets sur l'étude de celles-ci. Même si l'architecture basique décrite plus haut est apparue des années après celui-ci, je l'ai décrite avant car elle est beaucoup plus simple, voire trop.

L'idée principale du MVC qui a influencée de nombreux travaux est la séparation entre les données, ou le domaine des objets modélisant le monde réel, et la présentation correspondante aux éléments graphiques. Le domaine se doit d'être entièrement autonome et fonctionne sans aucune aide de la présentation. Cette approche permet encore aujourd'hui à de nombreuses applications d'être manipulées soit via une interface graphique, soit en ligne de commande.

Comme dit précédemment, le domaine, appelé aussi modèle, est complètement ignorant de l'existence de l'interface graphique. Cette dernière est composée des deux

10. GUI : Interface graphique ou *Graphical User Interfaces* en anglais

autres entités : la vue et le contrôleur. Le rôle du contrôleur est de capter les actions effectuées par l'utilisateur sur l'interface graphique et d'en décider leur usage alors que la vue se chargera de son affichage. Dans l'architecture originale, chaque vue est associé à son contrôleur, mais ce schéma a été adapté et transformé maintes fois, comme par exemple en ajoutant un super-contrôleur permettant de redistribuer les événements aux sous-contrôleurs correspondants.

Le patron observateur est une notion introduite par l'architecture MVC : la vue et le contrôleur observe le modèle. Lorsque le modèle est modifié, la vue adapte l'affichage d'elle-même. Malgré qu'il soit à l'origine de la modification sur le modèle, le contrôleur n'interagit pas directement avec la vue et laisse le mécanisme d'observation faire son travail. Ce processus est différent de celui de l'architecture basique évoqué plus haut mais ils décrivent différentes manières de gérer la synchronisation entre les données en mémoire et celles affichées. La première l'effectue via l'application manipulant les flux de données et les composants alors que l'architecture MVC repose sur la relation d'observation existante entre la vue et le modèle.

Une des principales conséquences du patron observateur est que le contrôleur n'a pas besoin de savoir quels composants doit être mis à jour lorsqu'une modification à lieu dans un autre, alors que dans le modèle basique, la scène devra maintenir une cohérence entre les différentes informations affichées dans tout les composants. Ce mécanisme devient particulièrement efficace lorsque plusieurs fenêtres complexes sont ouvertes sur les mêmes objets.

Le patron MVC peut se récapituler comme suit :

- Forte séparation entre le domaine (modèle) et la présentation (vue & contrôleur)
- Contrôleur et vue ne communique pas directement
- La vue observe le modèle permettant à plusieurs composants de s'adapter automatiquement

Modèle-Vue-Présentation

Cette dernière section portera sur l'architecture Modèle-Vue-Présentation (MVP) étant celle utilisée dans GIPSI Nomade. Elle est apparut chez *IBM*¹¹ dans les années 1990.

11. IBM : International Business Machines

Pour comprendre le modèle MVP, nous allons nous aider des deux précédemment décrits au-dessus. Le premier fournit une structure facile à comprendre tout en marquant une distinction entre les composants et le code de l'application. Manquant au précédent, la séparation entre la présentation et le modèle fait la force du MVC. Selon moi, MVP est une sorte de mélange entre ces deux approches.

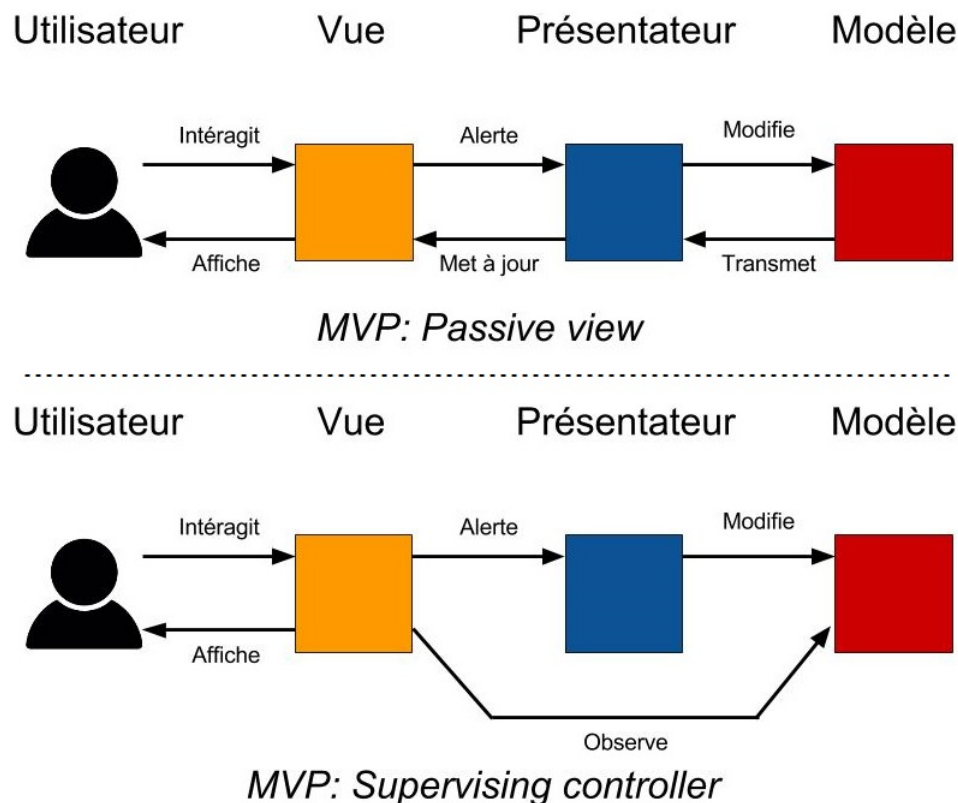
Deux écoles sont présentes lorsqu'il vient à expliquer le fonctionnement du MVP. La première traite la vue comme une simple structure de composants sans décrire leur comportement face aux actions de l'utilisateur, elle sera dite passive (*Passive View*). Certains comportements primitifs restent présents dans le composant lui-même, mais la plupart seront définis dans une seule méthode d'un présentateur qui décidera comment réagir lorsqu'il capture un évènement. Lorsque le présentateur modifie le modèle, il doit en avvertir la vue via des *Data Bindings* car elle se retrouve complètement isolée du modèle.

La seconde approche est d'importer de cette gestion d'évènements du présentateur dans la vue, lui laissant seulement les cas complexes (*Supervising Controller*). La vue peut être ici synchronisée comme le MVC grâce au patron observateur ou à travers des *Data Bindings*.

Il existe des similitudes évidentes entre les présentateurs MVP et les contrôleurs MVC, et certains diront que les présentateurs sont une forme lâche du contrôleur MVC, ce qui a pour conséquence que de nombreuses applications se prétendent suivre le modèle MVP, mais utilisent seulement le terme présentateur comme synonyme de contrôleur. Voici ce que l'on doit se rappeler du modèle MVP :

- Comme le contrôleur, le présentateur effectue les modifications sur le modèle observé par la vue
- Le présentateur peut communiquer avec la vue, du moins il sait qu'elle existe.
- La logique basique des composants a tendance à être placée dans la vue alors que la gestion des évènements utilisateur sont gérés par le présentateur.
- Ce dernier point peut varier, et plus la logique se trouve déplacée et monopolisée vers le présentateur, plus la séparation avec la vue sera importante.

Voici un schéma récapitulant les deux approches du modèle MVP décrites ci-dessus :

FIGURE 4.6 – Deux approches MVP : *Passive view* et *Supervising controller*

Ce type de graphes a tendance à représenter la vision de celui qui les interprètes, notamment dépendant de la technologie utilisée, et sont souvent soit trop complexes, soit trop simples. L'idée à retenir ici est la manière dont sont séparés les différents concepts.

4.3.3 Application du MVP sur GIPSI

Sous GIPSI, *JavaFX* décrit la vue dans deux types de fichiers : les fichiers FXML (ExempleDisplay.fxml) servent à décrire statiquement la structure générale de la scène, comme la position des composants. À ce fichier est associé une classe Java (ExempleDisplay.java) réalisant l'initialisation de l'interface, permettant entre autres de remplir les valeurs d'une *ComboBox* ou de lier la visibilité d'un composant en fonction de l'état d'un autre.

Le présentateur est lui aussi sous code Java (ExemplePresenter.java), implémentant obligatoirement une méthode « *Bind* » comme expliqué dans la section précédente, permettant de capturer les actions réalisées par l'utilisateur et de les traiter.

Enfin, le modèle est simplement un *Data Access Object (DAO)*, étant lui aussi un patron de conception complétant le modèle MVP, permettant d'accéder aux données persistantes dans des classes spécifiques, plutôt que de les disperser dans le code. Chaque *Presenter* est lié à son *Display* - mais non réciproque - , son *DaoEntity* ainsi qu'à un *EventBus* permettant de lancer et recevoir des notifications d'évènements grâce à *Guava* (voir 4.1.5).

Les fichiers se répartissent ainsi selon le modèle MVP :

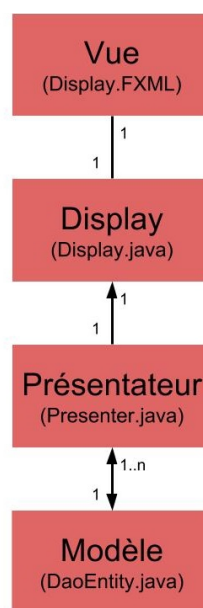


FIGURE 4.7 – Architecture MVP associé à JavaFX

À l'ouverture d'un écran, le présentateur est initialisé en premier et s'occupe ensuite de créer la vue. Après cela, il fera appel au modèle grâce au *DAO* qui renverra des données brutes qui seront ensuite retranscrite sous un *Bean FX* et lié à la vue via des *Data Binding* afin de présenter les données à l'écran (flux orange).

Le présentateur peut envoyer des notifications à travers le bus d'évènements. Ce dernier notifiera alors tout les objets abonnés à ce type de notification. Cela permettra entre autres de synchroniser l'avancée de processus ou de mettre à jour des données par un autre présentateur (flux vert).

Dans ce cadre-là, voici un schéma de cette architecture :

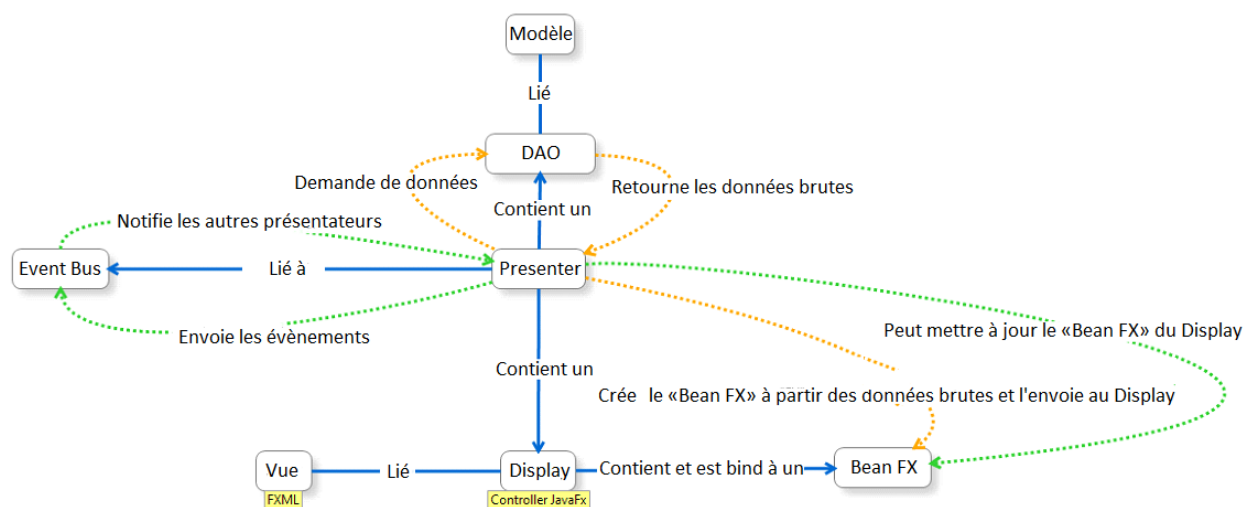


FIGURE 4.8 – Architecture MVP associé à GIPSI

4.4 Intervention

Cette dernière section propose des exemples d’actions réalisées pour permettre de visualiser concrètement ce qui a pu être accompli durant ce stage. Allant de la plus simple à la plus complexe, chacune sera expliquée pas à pas comme cela a été fait lors du traitement.

4.4.1 Exemple simple de correction d’anomalie

Le traitement des anomalies prend une part importante sur le temps de travail. Elles sont causées principalement par une modification mal implémentée ou dont les impacts n’ont pas été correctement évalués, faisant généralement suite à un retour du client ou des testeurs.

Une anomalie est classée selon un degré prenant trois valeurs : mineure, majeure et bloquante. De par son nom, une anomalie bloquante se doit d’être traitée immédiatement, impactant de manière critique l’expérience de l’utilisateur qui ne peut plus utiliser l’application normalement, obligeant parfois à effectuer une nouvelle livraison non programmée sur l’environnement du client pour qu’il puisse poursuivre ses tests. Une anomalie majeure perturbe fortement le déroulement de l’application et se doit d’être traitée avec attention mais n’en empêche pas son fonctionnement. Enfin, les

anomalies mineures perturbe l'utilisation sans pour autant être incapacitante. Ce sont en général des oublis lors du développement d'autres tickets.

Pour illustrer ce traitement de correction et se plonger progressivement dans l'architecture de GIPSI Nomade, prenons dans un premier temps l'exemple d'une anomalie mineure portant sur l'activation d'un bouton.

Constat du client

Sur l'écran d'affichage d'un formulaire, un bouton ne s'active pas correctement lorsque les champs requis sont remplis.

Cause

La gestion d'activation de ce bouton est faite sur chaque champ où un *listener* y a été ajouté dans le présentateur de la vue contenant le bouton, vérifiant à chaque changement de valeur si tous les champs requis sont remplis en appelant la méthode d'activation du bouton si tel était le cas. Ce fonctionnement est clairement périlleux sur plusieurs points : il faut préalablement avoir ajouté un *listener* similaire sur chaque champ obligatoire, impliquant de la répétition de code, un risque d'avoir oublié d'ajouter un *listener* et un dernier risque d'avoir oublié de vérifier l'état de chaque composant dans la condition, qui elle-même est d'un point de vue logique incohérente puisque qu'au lieu de vérifier que tous les champs requis soient remplis, il serait plus judicieux de vérifier qu'un seul ne soit pas vide.

Dans ce cas présent, la condition de vérification n'englobait pas tous les champs obligatoires concernés.

Solution

Deux solutions se sont dévoilées lors du traitement : corriger l'état actuel de la condition dans le présentateur en y ajoutant le composant concerné, ou simplifier le code en déplaçant cette gestion du bouton du présentateur jusqu'à la vue et lier dès l'initialisation de cette dernière, l'état d'activation du bouton aux champs obligatoires.

Choix et réalisation

Malgré les gains de performance et de lisibilité du code qu'apportent la seconde solution, le chiffage et les risques ont dû être pris en compte. Dans la pratique, il ne

faut pas chercher à vouloir tout refaire à sa manière mais savoir utiliser et adapter la solution qui a déjà été mise en place, cela même si ce choix n'est pas le plus optimisé. Sachant que la première solution n'était qu'un ajout de l'état du composant manquant dans la condition, alors que l'autre aurait impliqué la suppression de nombreuses lignes de code, le choix de la première s'est rapidement rendu évident.

4.4.2 Exemple avancé de correction d'anomalie

Pour ce nouvel exemple, l'anomalie en question traite sur l'affichage de texte riche dans l'application. En effet, malgré que le composant utilisé affiche correctement le texte, il n'adapte pas sa taille en fonction du contenu. Contrairement à l'exemple précédent, cette anomalie majeure a nécessité plus de temps durant la conception et la réalisation que sur le choix de la solution. Environ huit heures ont ainsi été nécessaires à son traitement.

Constat du client

Les composants affichant les textes riches ne s'adaptent pas suivant leur contenu, ce qui entraîne d'avoir des composants à taille statique, trop grand si aucun texte n'est présent, ou au contraire si le texte est trop important, d'être trop petit.

Cause

Pour comprendre le souci, il faut avoir une notion de l'architecture d'un langage balisé. En effet, quand il sera facile de compter le nombre de caractère d'un texte simple pour adapter la taille du composant qui l'affiche, il en sera autrement pour les textes riches. Les caractères formant les balises ne doivent pas être pris en compte et quand bien même cela est possible, le contenu affichable n'est pas homogène (taille et format de police, couleur du texte, ...).

Le composant utilisé est une *WebView* basique, permettant d'afficher le texte riche en bonne et due forme, sans pour autant ajuster sa taille en fonction du contenu.

Solution

La solution apportée est la création d'un nouveau composant en réutilisant l'ancien, le but étant d'utiliser le moteur initialement présent pour exécuter un script JavaScript « *(getDocument().getElementById(mydiv).getoffsetHeight)* » renvoyant la hauteur d'une balise précise.

Analyse

Plusieurs problèmes ont dû être préalablement résolus avant la réalisation concrète du composant. En effet, une des premières difficultés étant d'insérer la balise « `<div id="mydiv">` » au bon endroit quand les données représentant les textes riches en base n'ont pas la même structure.

Le second souci fut d'exécuter le script une fois que la page était entièrement chargée et pas avant, renvoyant une exception si tel est le cas et de ne pas pouvoir ainsi s'adapter au contenu.

Réalisation

La création du composant a pris un peu plus de deux heures. Le but étant de créer une classe étendant le composant pré-existant nommé *Region* pour y placer la *WebView* en lui indiquant d'occuper tout l'espace nécessaire.

Le traitement du texte passé en paramètre avant injection dans la *WebView* est géré par la méthode suivante :

```
/**
 * Formate le texte riche passé en paramètre
 *
 * @param content
 *      Le texte riche à formater
 * @return Le texte riche contenant une balise "div" identifiable
 */
protected static String getHtml(final String content) {
    if (content.startsWith("<html")) {
        // si le texte est bien formé, detecte la balise body et insere une balise div
        return content.replace("<body>", "<body><div id=\"mydiv\">").replace("</body>", "</div></body>");
    } else {
        // Sinon on insère le contenu dans une page HTML
        return "<html><body><div id=\"mydiv\" >\" + content + "\"</div></body></html>";
    }
}
```

FIGURE 4.9 – Méthode formatant le texte riche

Elle permet de détecter la forme générale du document et d'ajouter une balise avec un identifiant au bon endroit.

Le second problème est de s'assurer que le contenu est entièrement chargé par le moteur de la *WebView* avant de calculer la hauteur. Pour cela, on ajoute un *listener* sur l'état du chargement de ce moteur, et on exécute la fonction permettant l'ajustement de la taille uniquement lorsqu'il passe dans l'état de succès :


```
// Permet de traiter avec javascript une fois que le load content est terminer
this.webview.getEngine().getLoadWorker().stateProperty().addListener(new ChangeListener<Worker.State>() {
    @Override
    public void changed(final ObservableValue<? extends Worker.State> observable,
        final Worker.State oldValue, final Worker.State newValue) {
        if (newValue != Worker.State.SUCCEEDED) {
            return;
        }
        adjustHeight();
    }
});
```

FIGURE 4.10 – *Listener assurant le séquençage des actions*

Enfin, la dernière étape fut de remplacer l'ancien composant par le nouveau.

4.4.3 Exemple d'évolution

Une fois la majeure partie des anomalies traitées, les développements basculent peu à peu sur les tickets d'évolutions. Les évolutions sont le souhait du client d'ajouter de nouvelles fonctionnalités à l'application existante.

Expression des besoins

Nativement sur GIPSI Nomade, un inspecteur peut être amené à initialiser un objet localement en étant connecté ou non, en ayant par la suite seulement la possibilité de le consulter tant qu'une synchronisation via le réseau ne s'est pas faite.

Cette évolution porte sur l'ajout d'une fonctionnalité permettant à l'inspecteur d'avoir la main sur ce processus et choisir lui-même quand l'objet doit remonter sur le réseau en le *marquant*, tout en ayant la possibilité de le modifier tant qu'il ne l'aura pas fait. Une fois marqué, si le poste est connecté et que la synchronisation s'enclenche, l'objet sera remonté sur le réseau et supprimé localement, permettant à l'inspecteur de finaliser sa création sur le client web.

Spécifications

Étant une imposante modification de l'application, il s'en est suivit une longue phase de spécification. Il a fallu tout d'abord identifier le nouveau processus qu'allait suivre cet objet, de sa création à sa suppression. Dès lors qu'il est créé localement, l'objet détient un « *flag* », initialisé à *false*, permettant au processus de synchronisation de reconnaître s'il doit être pris en compte ou non. Ce « *flag* » ne sera uniquement modifié à *true* lorsque l'inspecteur lui en aura donné l'ordre, mais sera remis automatiquement à *false* s'il tente de le modifier à nouveau. Une fois la synchronisation

enclenchée, l'objet devient inaccessible et sera supprimé lorsqu'elle s'achèvera.

S'en suit le diagramme d'état suivant :

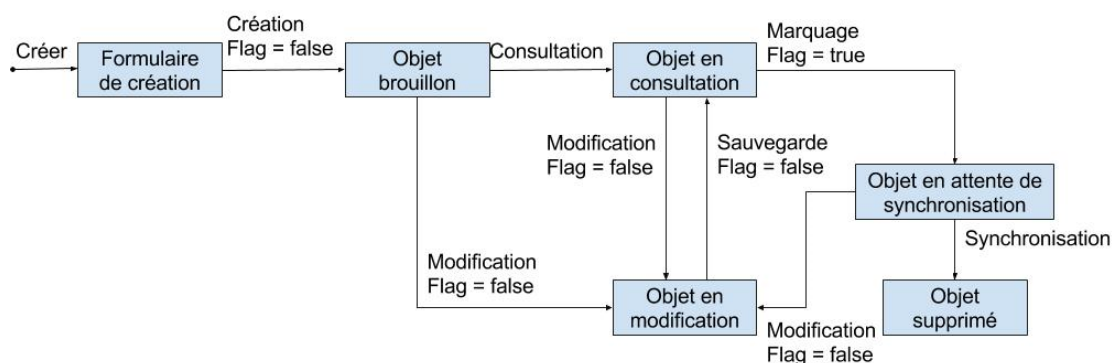


FIGURE 4.11 – *Diagramme d'état de l'évolution*

Le nouveau processus étant défini, il reste à spécifier les actions à mettre en place :

- Modifier le processus de création existant
- Adapter l'écran de consultation actuel
- Ajout de contrôles
- Notifier l'état de la synchronisation
- Supprimer les objets

Réalisation

La réalisation s'est déroulée suivant les étapes décrites précédemment. Le « *flag* » existant déjà auparavant, il n'a pas été nécessaire de modifier l'objet, uniquement le processus de création en initialisant sa valeur à *false*.

Ensuite, plusieurs solutions sont possibles pour basculer du mode consultation en modification. Celle choisie fut l'ajout d'un attribut *BooleanProperty* ayant le même fonctionnement qu'un booléen standard, mais permettant d'y lier directement le comportement d'autres objets comme suit :

```

/**
 * Identifie le mode d'affichage en cours
 */
private BooleanProperty modeConsultation;

/**
 * @return the modeConsultation
 */
public BooleanProperty isModeConsultation() {
    if (this.modeConsultation == null) {
        this.modeConsultation = new SimpleBooleanProperty();
    }
    return this.modeConsultation;
}

/**
 * Effectue l'initialisation générale de l'affichage de l'objet
 *
 * @param mode
 *         Le mode d'affichage : consultation (true) ou modification (false)
 * @param ale
 *         L'AbstractLocalEntity concernée
 */
public void initGenerale(final AbstractLocalEntity ale) {
    setAleEntity(ale);
    // Liaison de la visibilité/éditabilité/désactivation des éléments en fonction du mode
    // Boutons inférieur
    this.btnAnnuler.visibleProperty().bind(isModeConsultation().not());
    this.btnRetour.visibleProperty().bind(isModeConsultation());
    this.btnModifier.visibleProperty().bind(isModeConsultation().and(this.bdsEntity.etatProperty().isEqualTo("0")));
    this.btnEnregistrer.visibleProperty().bind(isModeConsultation().not());
    //TextArea
    this.textAreaFourniture.editableProperty().bind(isModeConsultation().not());
    this.textAreaCommentaireDeSurveillance.editableProperty().bind(isModeConsultation().not());
}

```

FIGURE 4.12 – Exemple de binding des composants dans la vue

L'ajout de contrôles fut ensuite nécessaire pour gérer la cohérence des écrans. Vu que l'on vient d'ajouter un *BooleanProperty*, il suffit de modifier sa valeur pour que les composants y étant liés changent leur mode d'édition et de visibilité :

```

/**
 * Binder du présentateur
 */
@Override
public final void bind() {

    /**
     * Gestion du bouton annuler (annule les modifications et recharge les données initiales)
     */
    getDisplay().getBtnAnnuler().addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
        @Override
        public void handle(final ActionEvent event) {
            getDisplay().isModeConsultation().set(true);
        }
    });
}

```

FIGURE 4.13 – Exemple de gestion d'évènement

Un dernier bouton fut ajouté permettant de marquer l'objet synchronisable. Dès que la synchronisation est enclenchée, on va lancer une notification dans le bus d'évènement, étant un objet dans lequel on peut placer ce que l'on souhaite comme suit :

```
this.eventBus.post(new SynchronisationIsOverEvent());
```

FIGURE 4.14 – Exemple de lancement d'une notification

Pour lancer une notification, il faut avoir injecté au préalable l'objet « *EventBus* » de la librairie *Guava*, dans lequel on va passer cette notification :

```
/**
 * Injection de l'EventBus
 */
@Inject
private EventBus eventBus;

/**
 * Initialisations post injections.
 */
@PostConstruct
public void initMasterPresenter() {
    this.eventBus.register(this);
}

/**
 * Création du service pour la synchronisation des données
 */
getDisplay().getProgressBarSynchro().progressProperty().bind(this.serviceSynchronisation.progressProperty());
this.serviceSynchronisation.stateProperty().addListener(
    (final ObservableValue<? extends Worker.State> observable, final Worker.State oldValue,
     final Worker.State newValue) -> {
        switch (newValue) {
            case READY:
                // La tâche est prête (état par défaut)
                MasterPresenter.this.getDisplay().getProgressBarSynchro().setVisible(false);
                MasterPresenter.this.getDisplay().getLabelPbSynchro().setVisible(false);
                // Lance un event à toute l'appli quand la synchro se termine
                MasterPresenter.this.eventBus.post(new SynchronisationIsOverEvent());
                break;
            case SCHEDULED:
                // La tâche va être exécutée.
                break;
            case RUNNING:
                // La tâche est en cours d'exécution.
                MasterPresenter.this.getDisplay().getProgressBarSynchro().setVisible(true);
                MasterPresenter.this.getDisplay().getLabelPbSynchro().setVisible(true);
                // Lance un event à toute l'appli quand la synchro débute
                MasterPresenter.this.eventBus.post(new SynchronisationIsRunningEvent());
                break;
            case FAILED:
                // La tâche a échoué.
                break;
        }
    });
}
```

FIGURE 4.15 – Lancement d'une notification au démarrage de la synchronisation

Toutes les classes connectées à l'« *EventBus* », et ayant souscrites à cette notification via l'annotation « *@Subscribe* », seront alors alertées de cette manière :

```
/**
 * Souscription aux événements de la synchro: démarrage
 */
@Subscribe
@SuppressWarnings("unused")
public final void handleSynchronisationRunning(final SynchronisationIsRunningEvent event) {
    getDisplay().isSynchronisationEnCours().set(true);
}
```

FIGURE 4.16 – *Souscription à un événement*

Dans ces deux derniers morceaux de code, la notification est lancée au début de la synchronisation qui permet de verrouiller l'accès aux objets assurant l'intégrité des données entre le client web et le client riche. Une autre notification est ensuite utilisée pour avertir l'arrêt de la synchronisation.

Enfin, la dernière étape consiste à parcourir tous les objets englobés par la synchronisation et de les supprimer.

4.4.4 Exemple de proposition de solution et chiffrage

Même si les évolutions viennent d'un désir du client d'ajouter quelque chose de nouveau au logiciel, il arrive parfois que lors de développements ou de chiffreages, les équipes proposent d'elles-mêmes au client une nouvelle fonction qu'elles jugent intéressante.

Nous allons ainsi voir ici la proposition que j'ai faite au client qui a suivi une demande d'évolution de sa part.

Expression des besoins initiaux

Sur la partie client web de GIPSI, les inspecteurs ont accès à une page d'accueil présentant les informations principales de leur compte. La MOA a donc exprimé le besoin d'avoir une page similaire sur le client riche et en a demandé le chiffage à l'équipe.

Proposition

Une fois l'analyse faite pour savoir si l'on pouvait récupérer les informations identiques à celles présentes sur le client web, une première maquette a rapidement été réalisée :

The diagram illustrates a web interface layout. At the top is a header bar containing the text "accueil Ressources Doctrine ...". Below the header is a tab bar with a single tab labeled "Accueil x". The main content area contains a form with the following fields: "Nom : XXX", "Prénom : XXX", "Quad : ABCD", "NNI : XXXX", and "Rôle : ?". The form is enclosed in a rectangular border, and the entire interface is framed by a larger border.

FIGURE 4.17 – *Proposition initiale*

On constate qu'une fois les informations affichées, il reste énormément de place inutilisée. L'initiative a été prise d'ajouter à ce bas de page les objets que l'inspecteur en cours avaient préalablement verrouillés sur son poste ou sur lesquels il devait effectuer des actions, présentés dans des tableaux à l'intérieur d'onglets comme l'illustre la figure ci-dessous :

accueil Ressources Doctrine ...

Accueil x

Nom : XXX Prénom : XXX Quad : ABCD NNI : XXXX Rôle : ?

CRA Verouillé BDS Brouillon FCE Brouillon

cra N°	Etat	...

⋮

< 1 2 3 4 >

consulter

FIGURE 4.18 – Proposition avancée

Une fois validée par le chef de projet, cette proposition fut transmise à la MOA qui, après y avoir rajouté ses remarques, en a particulièrement apprécié le contenu et nous en a demandé son chiffrage.

Chiffrage

Compte tenu des éléments à ajouter et à afficher dans la vue ainsi que des événements à lier dans le présentateur, s'en est suivi le chiffrage suivant :

- Bouton « Accueil » à la barre des tâches supérieure ouvrant un nouvel onglet : 20 minutes
- 4 libellés et champs à afficher en fonction de l'inspecteur courant : 40 minutes
- *TabPane* avec 3 onglets statiques contenant chacun un bouton et un tableau dans lequel chaque colonne doit être initialisée : 3 heures
- Un bouton « Actualiser » à la place du libellé « Rôle » permettant d'actualiser les données de tout les tableaux : 1 heure
- Tests : 1 heure
- Analyse : 2 heures

Le chiffrage total estimé comprenant les développements et tests unitaires transmis au chef de projet a donc été de 8 heures pour traiter cette évolution.

4.4.5 Atelier ergonomique

Ce stage ayant principalement porté sur l'aspect ergonomique de l'application GIPSI Nomade, il a été convenu avec la MOA de réaliser un atelier ergonomique le 3 juin.

Intérêt

Plusieurs visites de la MOA se sont déroulées dans les locaux de Sopra Steria lors de ces mois de stage. Ces visites révèlent de nombreux avantages pour les deux parties leur permettant d'établir et conserver une relation pérenne. Cela a l'avantage entre autre de rassurer la MOA sur l'avancement du projet tout en donnant l'occasion à l'intégrateur d'acquérir de nouvelles informations sur des processus attendus de l'application ou même de faire du business en proposant de nouvelles fonctionnalités.

Réalisation

L'atelier ergonomique s'est ainsi déroulé avec une personne de la MOA en coopération d'un autre membre de l'équipe. Durant une journée entière, de nombreuses anomalies ont été passées en revue, permettant d'en réaliser le traitement en direct pour certaines, et d'avertir du coût en temps pour la correction des plus complexes. Une fois terminé, des propositions d'amélioration de la qualité d'utilisation de l'application ont été abordées permettant d'intriguer le client sur de potentielles idées de futures évolutions.

Chapitre 5

Bilan & conclusion

5.1 Bilan

L'appréhension que l'on ressent au commencement du stage laisse rapidement place à l'envie d'en apprendre plus lorsque l'on est plongé au sein du projet et des richesses qui s'offrent à nous. Étant à l'aise avec Java, j'ai été rapidement affecté à la correction de JIRA après une rapide prise de connaissance du contexte et des objets manipulés par l'application.

Apprenant au fil des jours la méthodologie de travail, essayant du mieux possible d'assimiler les réflexes qu'un développeur se doit d'avoir, je me suis vite vu affecté de nouvelles tâches plus complexes et intéressantes. Me voir confier la réalisation d'évolutions m'a permis de confirmer mes efforts et de me motiver davantage.

Ayant énormément travaillé sur l'ergonomie de l'application et suite au départ de ressources humaines, j'ai assuré de manière autonome la prise en charge des correctifs y faisant référence. Cela m'a permis notamment de co-animer un atelier ergonomique d'une journée avec la maîtrise d'ouvrage.

Lorsque je regarde le déroulement de ce stage, je suis fier de ce que j'ai accompli et de la manière dont ma vision du monde du travail a mûri. À l'heure actuelle, j'aurai participé à quasiment toutes les étapes d'un projet informatique, et je compte bien en apprendre davantage sur le temps qu'il me reste.

5.2 Conclusion

Ce stage de fin d'études aura été un véritable enrichissement, autant sur le plan professionnel qu'humain. En effet, j'ai pu renforcer ma formation sur les technologies Java dans le cadre de mes différentes interventions, mais j'ai aussi pu m'imprégner de la qualité de la méthodologie de travail de Sopra Steria.

Cette méthodologie est primordiale si l'on veut aborder ce monde du travail du mieux possible, et ce stage aura eu l'immense intérêt de m'y préparer. J'aurai appris à constamment me remettre en question, que ce soit sur mes choix ou la quantité de tâches restant à faire.

La communication est la notion qui m'a le plus marqué, se révélant être un puissant outil indispensable à chaque développeur, que ce soit en écoutant l'expérience de ses pairs ou de faire valoir ses idées. Elle ouvre notamment de nouvelles voies de recherche lors de réflexion commune.

Durant ces cinq mois, je me suis vu évoluer et mûrir, au sein d'une équipe unie, qui m'a accueillie et mis à l'aise dès le premier jour, partageant leur expérience et leur vision du travail et me donnant la confiance nécessaire pour aujourd'hui me confronter à de plus grands défis.

Annexe A

Lexique

A.1 trigramme/etc

A.2 Lexique

Annexe B

Annexe

Explication des technologies

B.1 Outils

eclipse,plugins (sonarQube, mylyn, subclipse) maven , jpa PostgresQl et pgadmin
versionning : SVN

B.2 Langages et format de données

Java,sql,html,css,javascript1

Bibliographie

- [1] Wikipédia. Ergonomie informatique. *https://fr.wikipedia.org/wiki/Ergonomie_informatique*, 2016.
- [2] Wikipédia. HP Quality Center. *https://en.wikipedia.org/wiki/HP_Quality_Center*, 2016.
- [3] Wikipédia. Jira. *<https://fr.wikipedia.org/wiki/Jira>*, 2016.