Une Histoire de Migration Agile

Présenté par Jaxio au Model Driven Day 2010

Cas client Banque de France

Auditorium Prairie de 14h45 à 15h40

JAXIO info@jaxio.com http://www.jaxio.com/

Introduction

Les applications de gestion ont souvent un modèle métier proche du modèle de base de données associé. Une partie des spécifications de ces applications est classique: il faut permettre aux utilisateurs d'effectuer des recherches, de naviguer à travers les entités du modèle en suivant les relations entre elles, de faire des mises à jour, etc.

Cette particularité rend ces applications fastidieuses à développer si l'on ne dispose pas d'outils pour automatiser la partie technique.

Ainsi, dès les années 70 des éditeurs ont su proposer des AGL puissants aux directions informatiques. Mais aujourd'hui ces AGL arrivent en fin de vie.

Dans ce document nous partageons notre expérience de la migration de plusieurs applications de gestion client-serveur développées avec un AGL propriétaire vers une application Web développée en Java à la Banque de France.

Pour rester maître de la partie purement métier durant la migration et conserver une architecture propre, il a été décidé sciemment de ne pas utiliser d'outils de transformation de modèle à modèle.

Pour optimiser la tâche des développeurs et assurer une maintenance sur le long terme nous utilisons trois principaux leviers :

- Le modèle entités-relations existant,
- Une architecture évolutive adaptée au développement des applications de gestion basée sur Spring Framework, JPA, JSF, Spring WebFlow, RichFaces, Ajax,
- Le générateur de code Celerio.

Enfin nous décrivons de manière objective les avantages et les contraintes de notre approche et la vision à plus long terme.

Le Contexte

Notre Client a des centaines d'applications en production. Ces applications ont été développées au fil des années avec des technologies hétéroclites arrivant pour certaines en fin de support. Il est donc devenu critique pour des raisons de risques et de coûts liés à la maintenance de réécrire certaines applications.

La direction technique a fixé les règles suivantes dès le départ:

- Les applications client-serveur doivent être migrées vers des applications Web,
- Il faut rationnaliser la diversité technologique en définissant une architecture commune,
- Il faut privilégier lorsque l'on a le choix les technologies Open Source ainsi que les standards.

Ajoutons que les différentes applications ont des durées de vie assez longues (> 10 ans) et que leur développement est externalisé.

Il est donc primordial que la migration se fasse vers une architecture technique qui dans les dix années à venir ait encore des chances d'être comprise par les développeurs du marché.

Existant

Pour valider la faisabilité de la migration, la direction a souhaité partir d'un cas concret. Elle a fait réaliser par l'un de ses architectes un pilote d'application sur base d'une nouvelle architecture technique à partir d'une application existante bien connue en interne.

L'application d'origine a été développée à l'aide de l'AGL français Natstar en fin de vie et doit être régulièrement mise à jour pour prendre en compte des réglementations européennes et mondiales ainsi que des nouvelles demandes utilisateurs.

C'est une application client-serveur. Elle comporte des centaines d'écrans complexes permettant de manipuler plus de 900 entités métiers. Entre ces entités existent de nombreuses relations (1-1, 1-n, n-1, n-n) ainsi que de l'héritage. La base de données associée comprend 250 tables et une centaine de giga octets de données.

Les utilisateurs de l'application actuelle sont exigeants:

- Ils souhaitent conserver lors de la migration les données existantes ainsi que le modèle de données,
- Ils souhaitent reconduire l'ergonomie de l'application pour simplifier leur formation et la phase de recette.

L'AGL Natstar qui a permis de développer cette application a su résoudre en son temps de nombreux problèmes techniques et ergonomiques toujours d'actualité en offrant:

- Une bibliothèque de composants graphiques adaptés aux applications de gestion,
- Un ORM (Object Relational Mapping) très puissant qui n'a rien à envier aux Framework Open Source actuels et au standard JPA2.

Pour ce pilote l'objectif était de choisir une architecture permettant de reproduire à l'identique quelques écrans de l'application d'origine. Le principal challenge pour notre Client s'est immédiatement révélé être la gestion des conversations en web avec une ergonomie riche.

Conversations Web

Dans l'application d'origine, la plupart des fonctionnalités s'articulent autour de l'enchainement de plusieurs écrans (une conversation).

D'un point de vu technique chaque écran permet de modifier une partie du modèle métier. Les liens entre les écrans sont souvent le prolongement des liens entre les entités. Les modifications ont lieu en mémoire coté client. Une fois les différentes modifications effectuées et validées, l'utilisateur peut décider, ou non, de les propager en base de données de manière transactionnelle, grâce à l'ORM fourni par Natstar, et ainsi terminer la 'conversation'.

En web classique, pour reproduire les fonctionnalités d'origine, il faut:

- Etre capable de définir la notion de conversation, dont la portée est intermédiaire entre une requête http et une session http,
- Mémoriser coté serveur les modifications envoyées par le navigateur, en cours de conversation, par exemple lors des changements d'écran sans les propager en base de données,
- Propager en base les modifications de manière transactionnelle.

Ergonomie Riche / MDI

L'application d'origine comporte des centaines d'écrans. Il est possible de démarrer plusieurs conversations en parallèle pour mener de front plusieurs opérations disjointes. Il est possible de basculer d'une conversation à l'autre via un menu de type MDI[1]. Il faut pouvoir rafraichir certaines parties des écrans.

[1] MDI : Multiple Document Interface

Les écrans utilisent des composants graphiques riches avec de nombreux raccourcis clavier, panneaux, etc.

Tout ceci est loin du web classique 1.0 que nous avons connu à nos débuts.

Vision pour les choix technologiques

Notre client a une vision très claire de l'architecture. Elle doit avant tout être adaptée aux besoins des applications de gestion. Indépendamment de l'utilisation de générateur de code, le Client veut minimiser le code technique à écrire à la main.

Le Client souhaite également privilégier sur le long terme l'utilisation de standards matures sans être pénalisé sur le court terme par les limitations des standards actuels. Pour cela il a décidé de s'appuyer sur les bonnes parties des standards actuels et sur des technologies complémentaires qui sont déjà ou deviendront, selon lui, des sources d'inspiration pour les futures versions des standards.

Ainsi, au fil des sorties des nouveaux standards, leur adoption doit être relativement simple.

Le langage Java ayant été retenu[2], le Client a pris comme point de départ de son architecture le standard Java EE tout en sachant qu'il souffrait encore de limitations. Nous allons voir maintenant comment certaines limitations ont pu être levées.

Spring + JPA

Le « Spring Framework » s'est imposé comme standard de facto supplantant le standard JEE (<5) devenu bien trop lourd. Java EE6[3] a depuis rattrapé en partie son retard mais au début du projet, aucun serveur d'application ne proposait d'implémentation Java EE6 solide. Le Client a donc naturellement choisi d'utiliser Spring tout en sachant qu'une migration progressive vers Java EE6/7 serait possible par la suite.

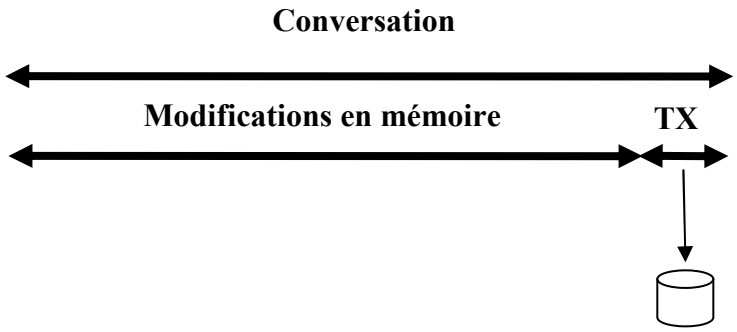
Les points forts de Spring qui nous intéressent ici sont:

- La gestion déclarative des transactions,
- le tissage des dépendances,
- la gestion des conversations via Spring WebFlow.

[2] Les raisons du choix de Java vs .Net par exemple ne sont pas détaillées dans ce document.
[3] Livre « Java EE 6 », d'Antonio Goncalves, voir http://www.antoniogoncalves.org/

Spring permet d'utiliser en partie des standards Java EE, notamment pour toute la partie ORM rendue populaire par le Framework Open Source Hibernate et standardisé par la spécification JPA 1.0 puis 2.0.

Les contextes de persistance étendus en JPA permettent de persister[4] en bloc (transaction courte) les changements appliqués aux objets métiers au cours d'une conversation (longue), ce qui répond bien au besoin. Ceci est supporté par Spring WebFlow et permet de ne jamais travailler avec des objets détachés.



Conversation longue et transaction (TX) courte

JSF

Coté IHM, le monde Open Source n'a pas su fournir une solution qui mette tout le monde d'accord comme Spring ou Hibernate coté backend. Il existe en réalité de nombreux Framework pour certains très innovants: Wicket, GWT, ExtJS, Struts 2, Spring MVC, Flex, Play! etc.

Chaque Framework a ses avantages, l'idée ici est de choisir un Framework adapté au contexte:

- Gestion des 'conversations' web,
- IHM construite avec des composants riches,
- Manipulation des objets métiers dans la couche de présentation,
- Maintenance de l'application > 10 ans.

JSF 1.2 (JEE <6) n'a pas bonne presse auprès des développeurs: les fichiers de configurations sont trop verbeux, la technologie est trop web 1.0 (le standard JSF 1.2 ne définit par exemple rien autour d'Ajax). Cependant après presque 10 ans, JSF est toujours présent en entreprise et le standard JSF 2.0 (Java EE6) comble enfin certaines lacunes (prise en compte d'Ajax par exemple) et semble assez prometteur.

[4] Sauvegarder en base de données

Par ailleurs des projets Open Source comme RichFaces ou IceFaces ont construit au dessus de JSF des composants riches (supportant Ajax), répondant en grande partie aux besoins.

Un choix de raison est donc de s'aligner sur le standard JSF qui est toujours mis en avant dans la spécification Java EE6.

Mais JSF souffre encore de quelques lacunes, notamment au niveau de la configuration de la navigation et surtout de l'absence (version < 6) de scope «conversation».

Le Framework SEAM, développé par JBoss, comble une partie des lacunes de JSF. Il a été créé pour développer le type d'application que notre Client cherche à migrer. Il utilise également RichFaces.

Cependant, au lancement du projet, SEAM ne fonctionne pas sur le « serveur d'applications » Tomcat et la mise en œuvre d'un contexte de persistance étendu dans les EJB3 Session Stateful bean peut s'avérer être une véritable barrière technique pour les développeurs et source de bugs subtils.

Spring Web Flow

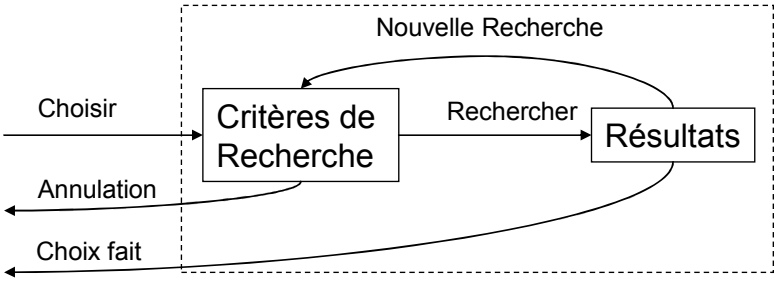
Pour rendre JSF adapté au besoin de conversation notre Client a choisi **Spring Web Flow** (SWF), une alternative plus simple à utiliser que SEAM.

SWF est un framework Open Source qui permet de configurer les enchainements entre écrans d'une application web et les actions associées à l'aide d'un DSL intuitif.

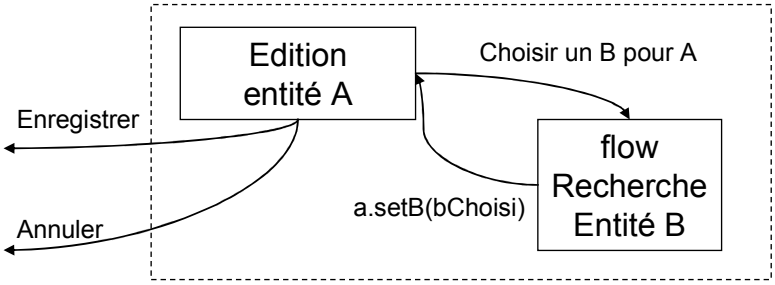
En tout point SWF surpasse JSF dans la simplicité de la configuration des enchaînements d'actions et d'écrans. Enorme point positif, SWF peut s'intégrer avec JSF via l'extension Spring-Faces et ainsi offrir le meilleur des deux mondes:

- Utilisation d'un standard, pour la partie composant de JSF avec un accès à des librairies de composants graphiques telles que RichFaces/IceFaces,
- Règles de navigations simples et très puissantes,
- Support des conversations (flow),
- Réutilisation des flows

La réutilisation des flows permet de modulariser des pans entiers d'IHM et de les réutiliser comme des « boites noires » à l'intérieur d'autres flows.



Exemple de « flow » de recherche



Exemple de « flow » d'édition réutilisant un « flow » de recherche

Par ailleurs SWF propose une encapsulation élégante et simple de la gestion du contexte de persistance étendu, clé pour le développement de notre pilote, au travers de ce que SWF appelle le FlowScoped PersitenceContext.

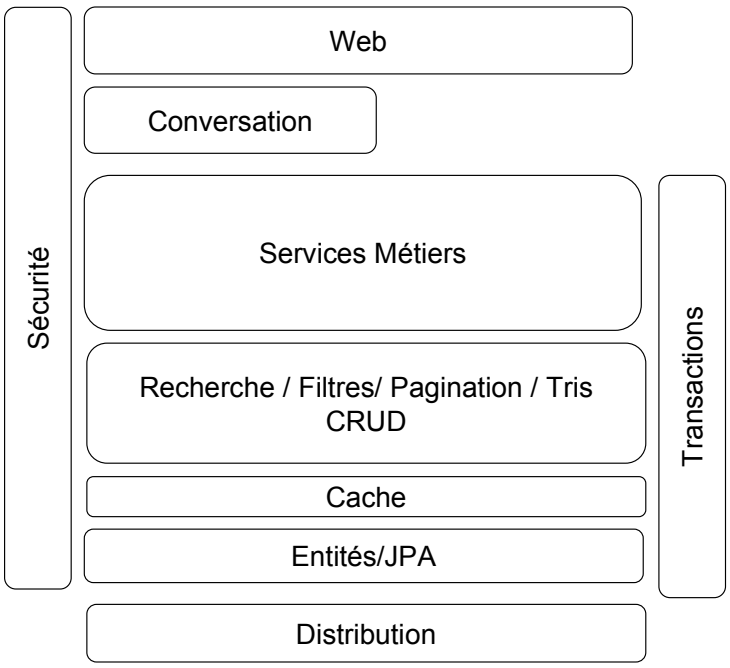
Ce pattern permet de reproduire le comportement Natstar et a l'avantage d'éviter plusieurs écueils :

- le pattern Data Transfert Object,
- les fameuses LazyInitializationException,
- la gestion manuelle de rattachement au contexte de persistance (méthode 'merge'). Les objets restent en effet attachés en permanence au contexte de persistance durant toute la conversation.

A armes presque égales

Ces choix techniques permettent d'avoir des armes similaires à celles des développeurs Natstar à la différence qu'elles sont standardisées ou en passe de l'être et qu'elles sont Open Source.

Ces choix conduisent naturellement vers une architecture classique en couche:



Architecture en couche

De plus ces technologies ne requièrent pas l'utilisation d'un serveur d'application JEE-compliant. La cible de déploiement choisie est Tomcat.

A cette étape le Client a une vision claire sur l'architecture à mettre en place. Le développement d'un POC avec des cas concrets non-triviaux a permis de valider la pertinence de ses choix.

Reste maintenant la problématique de l'industrialisation à traiter.

Industrialisation du développement

L'application à migrer comporte 900 objets métiers, 250 tables, des centaines d'écrans, des centaines de milliers de lignes de code. Par où commencer?

Les développeurs Natstar disposaient d'un environnement de développement avec un générateur de code puissant.

Le modèle métier contient le savoir et l'expérience accumulés sur des années par les concepteurs de l'application d'origine. En le conservant, nous accélérons la phase de conception et sécurisons la couverture fonctionnelle.

Par extension, nous conservons, à quelques changements près, la base de données d'origine et simplifions grandement la migration des données existantes, ce qui est primordial pour la validation.

Partir de la base de données

Le premier réflexe en interne a été d'utiliser le générateur de code fourni par Hibernate. Celui-ci a été rapidement écarté car trop lent (plus de 3h pour faire le reverse engineering de la base de données de 250 tables).

Suite à des essais concluants avec le service en ligne SpringFuse (www.springfuse.com), nous (Jaxio) avons été sollicités pour réaliser un prototype à l'aide de notre générateur de code Celerio.

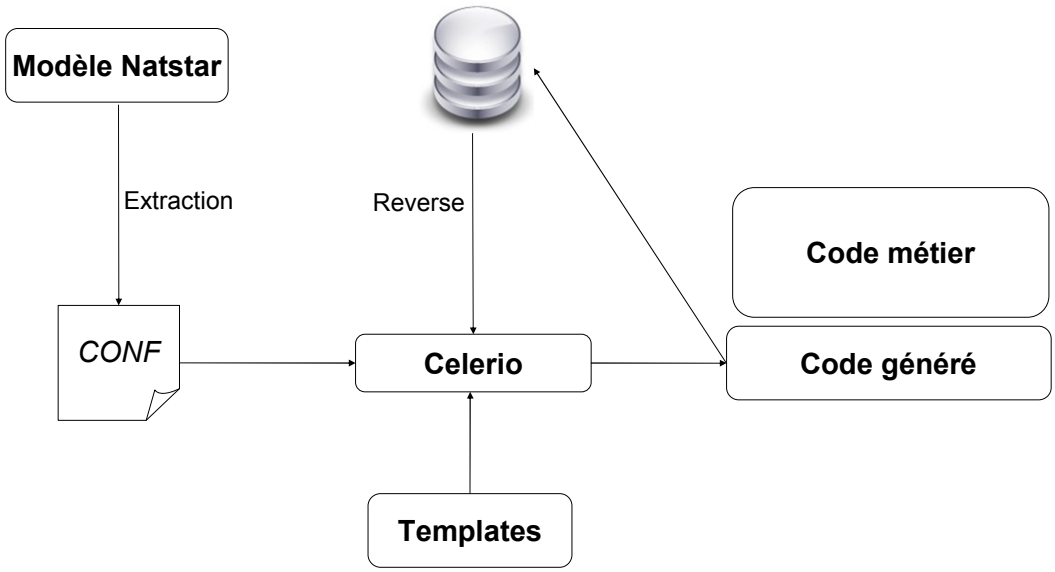
La première étape pour Jaxio fut de tout d'abord bien comprendre l'importance des choix d'architecture, l'intégration entre les technologies choisies et de prendre comme point de départ le POC réalisé par le Client.

Jaxio a ensuite mis à jour ses packs de templates de génération pour prendre en compte JSF et certains patterns clés.

Le reverse engineering de la base de données Oracle (250 tables) avec Celerio a pris moins de 3 minutes. Nous avons depuis amélioré ce temps d'extraction. La génération est aussi rapide. Cette rapidité permet d'intégrer le générateur au processus de build de manière quasi-transparente ce qui rend les développeurs beaucoup plus agiles.

Tout aurait été simple si à une table correspondait un objet métier. En réalité l'ORM de Natstar permettait déjà entre autres de faire du mapping d'héritage.

Pour que Celerio puisse générer du code qui soit le plus fidèle au modèle métier d'origine, un expert ayant la double compétence Natstar/Java et notamment JPA a créé un premier fichier de configuration Celerio à partir de méta données pertinentes présentes dans l'application d'origine (héritage entre les entités, nom logique des entités, noms des variables, types de liens entre les entités, etc.)



Alimentation de Celerio

Note : cette étape a demandé des améliorations sur le moteur Celerio en particulier pour le support de toutes les typologies d'héritage et la gestion des liens qui ne sont pas toujours représentés en base de données. Par exemple, les contraintes NOT NULL, les clés étrangères, les contraintes d'unicité, utilisées par Celerio, ne sont pas souvent déclarées dans le schéma de base d'origine alors qu'elles le sont au niveau du modèle métier.

Le prototype ayant donné toute satisfaction le véritable projet de migration a pu commencer avec une équipe de développeurs externalisée outillés avec Celerio.

Code métier comme référence

Le Client a écarté l'option de reprise de code métier par un traducteur automatique.

Il a préféré repartir des spécifications fonctionnelles d'origines et réécrire à la main la partie métier. Cette approche permet de mettre à jour les spécifications fonctionnelles et de repartir sur une base de code saine, bâtie au dessus du code généré.

Lorsque les spécifications ou les demandes fonctionnelles ne sont pas suffisamment précises alors que la fonctionnalité donnait entière satisfaction dans l'application d'origine, le développeur doit étudier l'ancien code. La reprise du schéma de base de données d'origine dans la nouvelle application et l'utilisation des mêmes noms de variables permet aux développeurs de mieux appréhender l'ancien code.

Vie projet

Peu après le lancement du premier projet, d'autres projets ont suivi.

Durant la vie de chaque projet il faut faire face à de nouveaux challenges. L'architecture et les technologies choisies ont permis aux équipes de s'adapter rapidement aux nouvelles demandes.

Amélioration continue des templates de génération

Les projets ont tous démarré sans avoir accès au code source des templates de génération pour éviter d'avoir à maintenir des templates différentes d'un projet à l'autre.

Sur la partie IHM chaque projet a des besoins très particuliers. Nous avons donc ouvert les templates IHM et aidé les projets à les adapter à leurs besoins, pour par exemple prendre en compte:

- l'utilisation de tags spécifiques,
- des règles de validations métiers,
- les recherches par intervalle de date,
- les spécificités de la mise en page, etc.

Coté backend nous avons pérennisé dans les templates les excellentes idées remontées par les développeurs.

Audit

Les projets peuvent introduire l'historisation des changements en base de données en utilisant ENVERS de JBoss qui s'intègre naturellement avec les technologies choisies.

Etude Architecture et Haute Disponibilité

Certaines applications du Client sont critiques et tout arrêt de service, même très temporaire, peut avoir des conséquences majeures: il est donc nécessaire que les applications développées permettent techniquement de survivre à des pannes matérielles, y compris des pannes critiques (perte d'un data-center complet).

Il s'agit là de l'un des nets avantages des nouvelles technologies choisies par rapport aux solutions client-serveur basées sur des mainframes, qui par nature ne peuvent pas proposer ce type de fonctionnalité.

La haute disponibilité d'une application Java est une problématique multiple: pour l'obtenir il ne faut pas de "Single Point of Failure", et donc que:

- L'infrastructure réseau supporte la haute disponibilité et permette de rediriger automatiquement des requêtes HTTP d'un serveur à un autre,
- La base de données soit répliquée sur plusieurs machines en temps réel,
- Les serveurs d'applications soient eux aussi en cluster, de manière à ce que la perte d'un serveur Java n'ait pas d'impact: il faut pour cela répliquer les sessions HTTP et les caches Hibernate sur plusieurs nœuds.

Ce dernier point est certainement le plus complexe: les applications du Client étant des migrations de clients lourds, réalisées avec JSF, elles ont tendance à utiliser de manière importante les sessions HTTP et les caches Hibernate. De plus, la mise en haute disponibilité d'applications JSF est une problématique rarement prise en compte: la plupart du temps, les entreprises se contentent de répliquer le cache Hibernate, et considèrent que les données en session ne sont pas d'une importance majeure.

Le Client ayant des contraintes particulièrement élevées, une étude "Haute Disponibilité" a été menée sur une application générée par Celerio. Cette étude a permis de montrer que les applications Java basées sur l'architecture choisie pouvaient être mises en cluster grâce à deux technologies de cache distribué: TerraCotta et Oracle Coherence.

Montée de version des Frameworks

Les montées de versions des Framework sont d'abord réalisées à la main sur un projet d'exemple généré par Celerio.

Les modifications requises sont ensuite reportées dans les templates de génération.

Les projets peuvent ensuite utiliser les nouvelles templates de génération pour monter de version sereinement.

La prochaine mise à jour concernera le passage de JSF 1.0 à JSF 2.0. Nous attendons pour cela que RichFaces sorte la version 4 compatible avec JSF 2.

Limitations

Manque de littérature technique

Prise individuellement chaque technologie utilisée est largement documentée (Internet, livres, forums, etc.)

En revanche il y a peu de littérature sur l'intégration simultanée des technologies choisies:

- Spring Web Flow avec le pattern FlowScope PersitenceContext,
- Spring Faces et JSF,
- La démarcation déclarative des transactions avec Spring.

L'équipe des architectes a écrit des normes et mis en place des formations pour sensibiliser les nouveaux venus aux subtilités et à la pertinence de cette intégration.

L'IHM reste le centre de coût le plus important.

La réalisation de comportements avancés, proches de ceux d'un client lourd, ou de widget non supportés par RichFaces font perdre un temps précieux aux développeurs et ne sont pas maintenables sur le long terme.

Ces développements auraient pu parfois être évités en privilégiant le dialogue avec les équipes fonctionnelles.

Au fil des projets nous avons réussi à mieux anticiper ces dérapages en sensibilisant les équipes projets et en proposant des solutions alternatives.

Conclusion

L'utilisation d'un AGL comme Natsar permettait d'améliorer la productivité, de limiter les possibilités offertes aux développeurs et en quelque sorte d'homogénéiser les développements. En contrepartie, tout était très fermé.

La nouvelle architecture choisie permet de réaliser des applications de gestion complexes comparables, mais en web, sur des technologies ouvertes.

Celerio est utilisé pour améliorer la productivité et cadrer les développeurs en générant du code conforme aux normes définies en interne.

L'expérience acquise sur chaque projet est réinjecté dans les templates de génération. Les nouveaux projets démarrent ainsi sur des bases extrêmement solides. Nous tâchons d'entretenir ce cercle vertueux sans fin, dans la mesure où les technologies ne cessent d'évoluer.

Les développeurs doivent cependant avoir un niveau confirmé pour pouvoir appréhender dans leur ensemble les technologies utilisées. Une fois ce cap franchi les développeurs écrivent essentiellement du code métier et IHM.

Les difficultés rencontrées sur la partie IHM ne sont pas rédhibitoires mais montrent que cette partie n'est pas encore assez mûre.

Remerciements

Nous tenons à remercier, pour leur patience et leurs feedbacks

- Bernard Pons, architecte à la Banque de France
- Hervé Le Morvan, consultant, Ceinture noire[5] Java, expert Natstar et autres systèmes
- Tous les développeurs qui travaillent avec Celerio ou www.springfuse.com, la version en ligne de Celerio

Nous remercions également Julien Dubois, co-auteur du livre « Spring par la pratique », pour sa contribution sur la haute disponibilité.

[5] Voir http://www.blackbeltfactory.com/