Prospective

Une approche relocalisante et économe pour fournir des logiciels de qualité.

Ref: MZZ/09/R&D/ECÒ
Date: 29/04/09

mezzÒnomy
6 rue des Tamaris – 31700 Blagnac

Pierre Gradit
Gérant

Pierre Gradit a protégé ce document en version 1.0 le 29/04/09
Pierre Gradit a modifié ce document en version 1.1

Mots-clés: Economie, Coût, Développement logiciel, Externalisation, Relocalisation

Sommaire

Table des matières

1 Introduction....3
2 Contexte et objectifs....3
2.1 Vie d'un programme....3
2.1.1 Double-compétence d'un programme....4
2.1.2 Divergence d'un programme....4
2.1.3 Usure d'un programme....4
2.2 La phase interne : le règne du développeur....5
2.3 Phase externe : le règne du contrat....6
2.3.1 Irruption de l'aspect temporel....7
2.3.2 La diffusion des connaissances....8
2.3.3 Dispersion de l'effort d'écriture....9
3 Modèle économique relocalisé....9
3.1 Logiciel programmable....9
3.1.1 L'auteur....9
3.1.2 La méthode....9
3.2 Dynamique des programmes....10
3.3 Pourquoi ça marche ?....11
3.3.1 Séparation des compétences....11
3.3.2 Suppression de l'effet bouton....12
3.3.3 Suppression de l'effet tunnel....12
3.4 Réorganisation isofonctionnelle....13
4 Exemples de réalisations....13
4.1 Réalisation passées....13
4.1.1 Lancer des calculs depuis une feuille de calcul....13
4.1.2 Passer des tests embarqués sur un PC....14
4.1.3 Mutualiser des méthodes de calcul....14
4.2 Réalisation possibles....14
4.2.1 Un outil programmable sur mesure....14
4.2.2 Toolzòne: Logiciel programmable de première génération....15
4.2.3 Mirza: Logiciel programmable de deuxième génération....15

1 Introduction

L'utilisation de l'informatique est un facteur puissant d'augmentation de la productivité individuelle. A titre d'illustration de la puissance de ce facteur on peut constater que si son utilisation a commencé dans les années 60, à ce jour, concevoir un outil productif sans outillages informatiques n'est plus envisageable.

L'intégration de l'outil informatique dans la production a connu deux phases principales. Dans un premier temps, les entreprises ont développés en interne leurs outillages, et ce développement ne suivait pas de schémas pré-établis. Dans un deuxième temps, les décideurs ont repris le contrôle de leur outillage logiciel par l'externalisation massive des outillages informatiques.

Les deux solutions, **interne** et **externe**, ont des avantages et des défauts. Notre approche consiste dans une troisième voie, à la fois **interne** et **externe** dont nous pensons qu'elle permet d'une part une relocalisation des activités des entreprises et d'autre part qu'elle est économe.

- Par **relocalisante**, nous entendons qu'elle permet une meilleure maîtrise des flux de connaissance dans le développement logiciel et par là un meilleur meilleur contrôle des compétences métiers critiques pour l'outil productif.
- Par **économe**, nous entendons qu'elle augmente significativement la productivité du développement logiciel face aux deux modèles passés **interne** et **externe**, et surtout que son impact sur la productivité de l'opérateur productif est meilleure que celle des deux modèles antérieurs

L'objectif de ce papier prospectif est de montrer comment notre approche de développement informatique à la fois **interne** et **externe** peut être **relocalisante** et **économe**. Cette approche de développement s'appuie sur des plateformes qui ont deux capacités : **programmables** et **certifiables**.

2 Contexte et objectifs

Cette première partie définira les concepts élémentaires associé aux programmes et à leur existence dans les entreprises qui les utilisent, ensuite nous montrerons les deux principales formes d'organisation

2.1 Vie d'un programme

Pour raisonner, nous assimilerons *un* logiciel à *un* seul programme. Dans ce cadre ainsi simplifié, le processus de développement logiciel s'assimile au processus d'**écriture d'un texte** en « *langage artificiel* » : le « *programme* ».

Nous allons nous intéresser à trois phénomènes majeurs qui sont à l'origine de la plupart des problèmes rencontrés par l'intégration des logiciels dans la production :

- **La double-compétence** : un programme est écrit par la coopération de deux compétence, « métier » et « informatique ».
- **La divergence** : un programme ne satisfait jamais complètement l'intention qui à présidé à

son écriture dans le cas général.

- **L'usure** : un programme finit pas être manipulé par des personnes qui ignorent son fonctionnement.

Ces trois phénomènes ont une telle réalité qu'il est possible pour un expert en étude de programmes existants de déterminer – à la manière d'un oenologue – les paramètres clés d'un programme : quand ce programme a été crée et comment il a vieilli.

2.1.1 Double-compétence d'un programme

La particularité du logiciel est de faire intervenir deux types de compétences dans le processus d'écriture d'un « *programme* » :

- la compétence « *métier* » qui permet de cerner de « *quoi* » parle ce texte
- la compétence « *informatique* » qui permet de cerner « *comment* » ce texte « *parle* »

Un « programme » est *toujours* **un texte écrit à deux mains** et fait intervenir deux types de connaissances. Un « programme » résulte *toujours* d'un processus coopératif entre deux mondes qui ne partagent pas le même vocabulaire technique.

C'est dans cette dynamique de « *nommage* » entre les mondes « *informatique* » et « *métier* » que se situe le **gisement de productivité** que compte exploiter la société mezzÒnomy.

2.1.2 Divergence d'un programme

Un programme est réalisé pour remplir une fonction mais un programme ne remplit dans les cas réel jamais complètement cette fonction. Un modèle de cette situation peut être exprimé avec les deux infinis : une intention est un nombre réel (comme π ou e) et un programme est un rationnel (une fraction).

- Lorsque l'intention est une fonction calculable, il est possible d'écrire un programme qui atteindra exactement l'intention. En effet, en vertu de la Thèse de Church, l'intention est *de facto* un programme dans ce cas particulier.
- Mais lorsque l'intention est réelle et ne correspond pas *a priori* à un programme, il est parfaitement courant qu'aucun programme ne puisse atteindre exactement l'intention. Donc inévitablement, des acteurs feront un constat de divergence entre l'intention et le programme. Cette non calculabilité arrive très vite en particulier comme l'a constaté Wegner lorsque l'on considère des système interactifs.

Le temps séparant deux constat de divergence entre l'intention et le programme est appelé « *temps caractéristique* ». En pratique plus le temps passe plus l'écart se réduit et plus le « *temps caractéristique* » augmente.

2.1.3 Usure d'un programme

Nous avons vu que dans les cas réels un programme doit être constamment amendé. Ceci génère des retouches continuelle sur un support fragile. En outre un phénomène rajoute à la difficulté, la vitesse à laquelle on oublie comment on a articulé la double-compétence pour produire un programme. Le temps de demi-vie de cette connaissance active et transformante – une *praxis* – est

nommé « *temps d'évaporation* ».

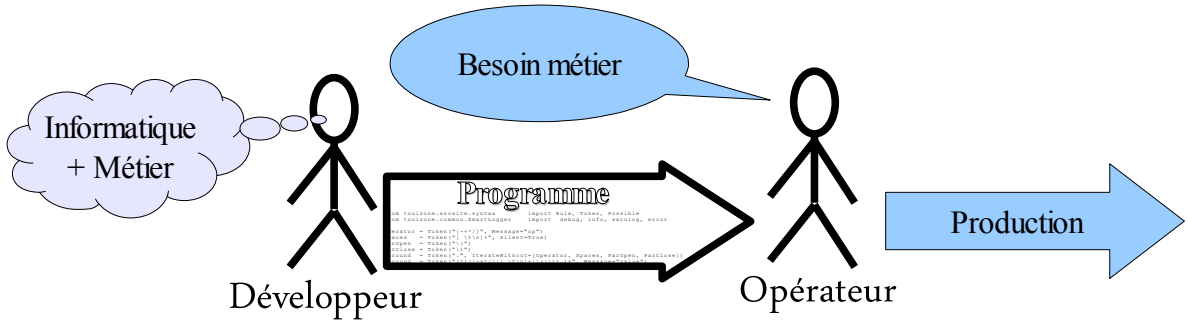
Inévitablement, le « *temps caractéristique* » dépasse un jour le « *temps d'évaporation* » et à partir de ce moment là, le programme s'use : c'est le temps de la « *bidouille* ». Il est retouché par des personnes qui ne savent plus vraiment comment il est construit et à quoi servent ses différents éléments constitutifs. Leur programmation est gouverné non plus par une *praxis* mais par une *hexis* – une pratique figée et utilitaire.

L'action sur le programme est à base d'interventions syntaxiques simples, des coupures et des duplications et des surcharges : le programme est utilisé comme un formulaire. Cette action sur le programme ne parvient plus à régénérer sa *praxis* et le programme finit par étouffer sous les redites et les incohérences.

2.2 La phase interne : le règne du développeur

L'outillage informatique s'est développé dans un premier autour des outils de production industrielle de données scientifiques. L'émergence du web au début des années 1990 au sein du CERN, avec ses deux composantes HTML et HTTP, illustre parfaitement cette proximité naturelle entre outillage informatique et industrie de la production de données scientifiques. Cette date marque aussi la fin de la phase **interne** de l'intégration informatique dans l'outil de production.

Pendant cette phase, le développement d'outillage informatique concernait des personnes potentiellement isolées qui détenaient les deux compétences « *métier* » et « *informatique* », et qui écrivait avec leurs « *deux* » mains, rendant cette « *coopération* » plus simple mais par là même inobservable à des acteurs extérieurs.



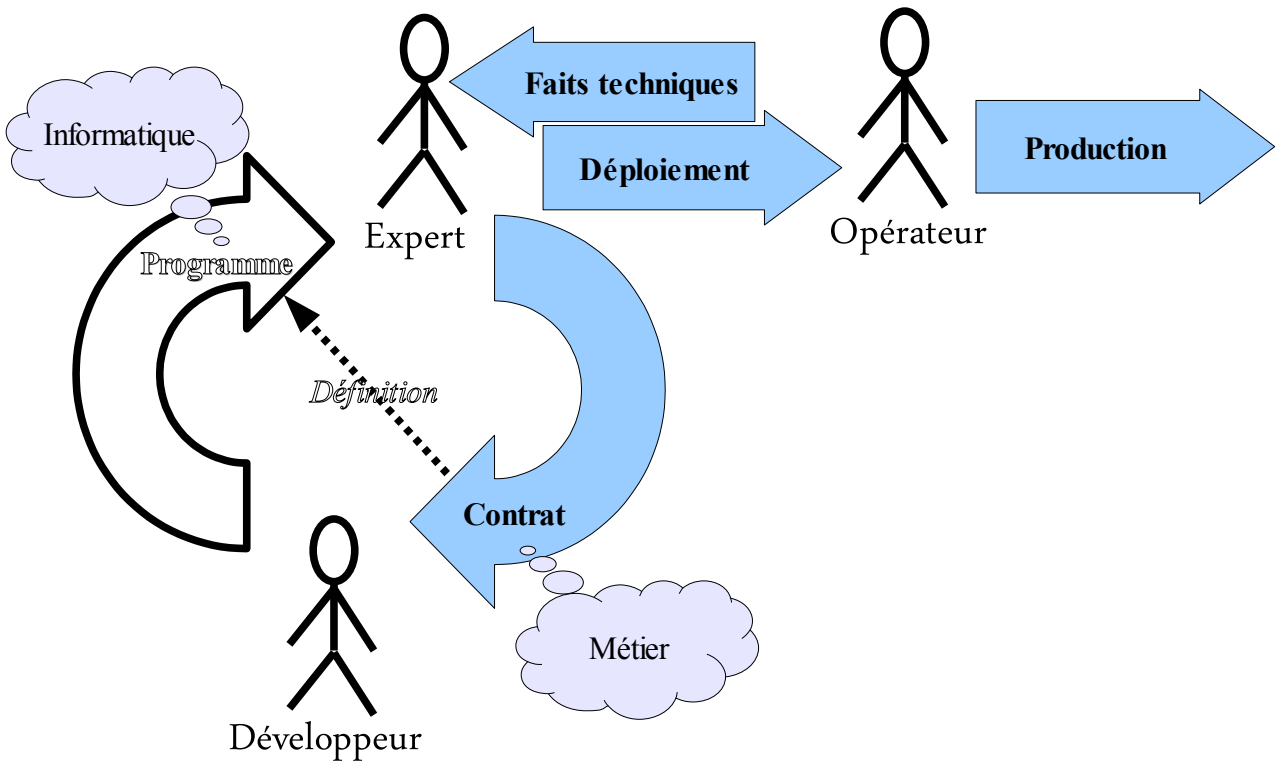
La manufacture de programmes était l'œuvre d'artisans renommés dont le savoir faire confinait à une pratique magique. Cette situation privilégiée, associé aux retombées productives que cette intégration provoquait eut pour effet de donner à ce « *producteur de programmes* » un rôle éminent dans l'entreprise qu'attestent les grilles salariales de l'époque et que le film TRON des studios Disney, produit au début des années 80, illustre sur un mode symbolique.

Cette situation éminente posait de lourds problèmes à l' « *entreprise hôte* ». En particulier, le risque associé à la disparition de cette personne dépassait tous les standards possibles et quand il fut clair à tous que l'informatique faisant durablement partie du paysage productif, le phénomène d'usure des programmes commença à devenir une réalité économique. Enfin, l'irruption de la notion de partage – remise au goût du jour par cette innovation fantastique qu'est le web – donna le coup de grâce idéologique au monde des pionniers.

2.3 Phase externe : le règne du contrat

La phase « *externe* » correspond à la première réponse collective du corps social qu'est l'« *entreprise hôte* » au phénomène d'intégration de l'informatique. Cette réponse tient en un seul mot : *externalisation*.

Des acteurs de l'entreprise hôte, les « *experts* », furent chargés d'identifier les éléments « *métier* » à intégrer dans le logiciel sous forme contractuelle puis ce contrat fut transmise à des « *développeur* » chargés de « *développer* » le programme.



Ceci eut pour effet principal de rajouter un texte en « *langage naturel* », toujours plus élaboré, **définissant** le programme attendu contenant principalement des connaissances « *métier* ». Ceci a deux effets majeurs qui forment les deux axes d'améliorations que nous proposons :

- **L'irruption de l'aspect temporel** : l'« *effet tunnel* » ou l'« *effet bouton* » sont des exemples de comportements temporels ressenti comme déficients. Il s'agit de cas particuliers résultant des liens de dépendance temporels complexes qui se tissent dès que le modèle **externe** entre en fonction. Non triviaux, ces dépendances peuvent amener cette solution à créer des blocages réduisant la productivité des opérateurs.
- **Le problème de la diffusion des connaissances**: le contrat contenant exclusivement des informations métiers parfois critiques, cette sortie d'informations a un impact économique qui n'est pas à négliger et qui pourrait être une des sources des capacités accrues de délocalisation constatées depuis une dizaine d'années.
- **Dispersion de l'effort d'écriture** : cette architecture de coopération détourne l'attention générale du texte *cardinal* – le programme – au profit d'un énoncé *auxiliaire* en langage

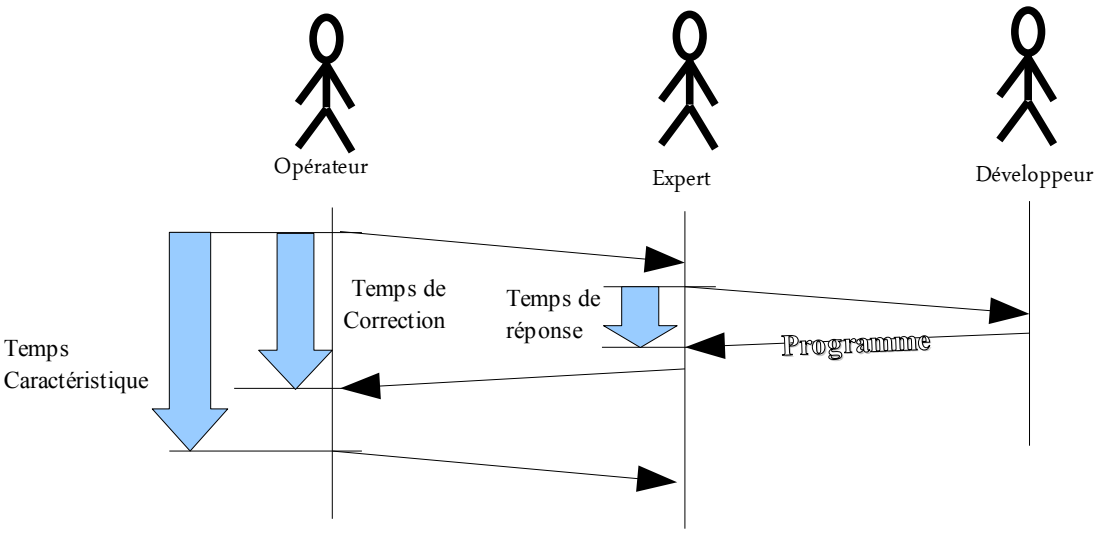
naturel – le contrat – qui mobilise des ressources de plus en plus conséquentes au détriment du programme. Notons au passage que cette « *diversion* » peut créer massivement de la valeur, au sens fiscal et économique du terme, sans forcément créer une réelle augmentation de la productivité individuelle.

Ce découplage entre croissance mesurée par l'administration fiscale ou les agences de notation et la réalité de la productivité individuelle, ainsi que les aspect relatif à la maîtrise des connaissances critiques sont des aspects essentiels de la crise économique actuelle et des modalités de sa sortie.

2.3.1 Irruption de l'aspect temporel

Le premier problème identifié pour ce type d'architecture est l'« *effet tunnel* ». En effet, un « *temps de réponse* » sépare le moment où un contrat C est défini et le moment où le programme P validant les termes de C existe. Pendant la période – le tunnel – où le contrat C est fixé et le logiciel P n'existe pas encore, il apparaît absurde de modifier C qui reste donc figé pendant un temps de l'ordre de la demi-année : ceci est l'« *effet tunnel* ».

Pourquoi en arrive-t-on là ? La raison fondamentale est qu'un programme ne remplit jamais exactement la fonction *intentionnelle* pour laquelle il est écrit. Il est possible de définir des édifices mathématiques rendant cette réalité démontrable. Et donc, il est *inévitable d'observer* que le programme ne fournisse plus une fonction *adéquate*. Pour faire simple, tout programme devient faux – en fait présente une nouvelle *divergence* – au bout d'un « *temps caractéristique* » et ceci quel qu'en soit la raison d'un point de vue contractuel.



Le « *temps de correction* » est le temps pendant lequel un constat de « *divergence* » reste actuel. Or ce « *temps de correction* » intervient directement dans le calcul de la productivité de l'opérateur.

La caractéristique structurelle de l'architecture externe est de le rendre strictement supérieur au « *temps de réponse* » quel que soit la nature de la divergence. Voici un autre effet pervers de l'architecture externe : l'« *effet bouton* ». Toute modification même mineure d'un programme requiert un temps qui peut être sans commune mesure avec la nature de la divergence : rajouter un bouton dans une interface peut prendre six mois ! En outre, pour les décideurs, son coût est en rapport avec la mobilisation des ressources juridiques et expertes nécessaire à ce processus de haute technicité coopérative. Il est possible d'arriver à des coûts de l'ordre de la centaine de milliers

d'euros pour une opération ressentie comme du niveau de celle que tout un chacun réalise à titre de loisir dans ses activités de bricolage perçue, à tort, comme gratuites. L'« *effet bouton* » est aujourd'hui une cause majeure des critiques du modèle externe.

A l'issu de ce tour d'horizon, notre modèle temporel, qui ne concerne que l'entreprise cliente, est complet :

- « *temps de réponse* » : temps mis pour mettre un programme en *conformité* avec un contrat
- « *temps caractéristique* » : temps mis par un programme pour présenter une nouvelle *divergence* (i.e. ne plus être en mesure d'être utilisé sans surcout unitaire à une fonction donnée dans la production)
- « *temps de correction* » : temps de validité d'un constat de *divergence* (i.e. être en mesure à nouveau d'être utilisé sans surcout unitaire à une fonction donnée dans la production)

Ces notions permettent de raisonner même si elle nécessiteront une formulation mathématique plus élaborée pour servir de base à des modèles économiques pertinents. Par exemple, il suffit que le « *temps caractéristique* » passe sous le « *temps de correction* » pour que le programme soit en « *perpétuelle divergence* ». Cet état est régulièrement rencontré dans les programmes actuels développé selon la méthode **externe**. Pour les opérateurs en bout de chaîne, cet état s'apparente à une réalité où le programme ne fonctionne *jamais* complètement, même si cette perception n'est pas forcément fondée sur des éléments objectifs mais une perception subjective issue de l'effet « *trains en retard* » : la société discours toujours plus des faits *disruptifs* que des faits *normatifs*.

2.3.2 La diffusion des connaissances

Un autre aspect paradoxal de la stratégie externe est qu'elle est conçue en théorie pour que chacun travaille dans son domaine, mais se pose en pratique le problème de la réception des éléments: Le développeur doit réceptionner le contrat et l'expert doit réceptionner le programme

Or dans le registre technique existe le « *paradoxe du lecteur* », en effet dans ce registre à la différence du registre littéraire , « *lire est plus compliqué que d'écrire* ». Donc il faut de sérieuses compétences métier pour réceptionner un contrat et de sérieuses compétences informatiques pour réceptionner un programme : ceci va complètement à l'encontre du but poursuivi.

De ce constat initial paradoxal se décline divers type de problèmes qui entravent l'efficacité du modèle externe :

- **Le problème des droits** : vu le prix payé pour le service de développement, l'expert se vit comme propriétaire des sources. Ceci entre en conflit avec l'objectif de réutilisation propre au développeur. Or cette réutilisation lui permettrait de réduire ses coûts en mutualisant les développements pour différents client.
- **La rupture de la barrière d'entreprise** : les « *développeurs* » acquièrent une compétence non négligeable en termes de métier et même si il est blindé en surface par des clauses juridiques, ce transfert est une réalité humaine incontestable. Des connaissances critiques passent ainsi la barrière d'entreprise dans des proportions qui ne peuvent être négligées.

Bref, l'objectif théorique de recentrer chacun sur son cœur de métier n'est absolument pas atteint. Non seulement, nous y reviendrons, tout le monde doit développer des compétences de nature juridique, mais en plus, à cause du « *paradoxe du lecteur* », les compétences migrent allègrement de

part et d'autre de la barrière d'entreprise rendant cette cloison poreuse en termes de connaissance à tous les niveaux. Cette porosité pose de graves problèmes de propriété industrielle dans tous les sens.

2.3.3 Dispersion de l'effort d'écriture

Ecrire un contrat qui permettent de spécifier un logiciel « *pour toujours* » présente la même impossibilité théorique que de faire un logiciel qui fonctionne « *pour toujours* ». Cela n'empêche pas toutes les parties de rendre ce document de plus en plus sophistiqué pour essayer d'en maximiser l'utilité et la durée de validité.

En conséquence, toutes les parties se mettent à établir des normes de nature juridique de part et d'autre de la barrière d'entreprise pour se prémunir de la responsabilité en cas d'incident ou d'accident. Mais tout ceci à un coût et tout cet effort sur le contrat est divertit de celui l'écriture du programme. Le ratio entre l'effort total et la partie de l'effort qui contribue *directement* à l'augmentation de la productivité de l'opérateur est appelé « *facteur de qualité* ».

Aujourd'hui, le « *facteur de qualité* » ne peut inférieur à 3 sans risque de panne majeure du service rendu par le logiciel. Au maximum un tiers de l'activité de développement peut être consacré à l'effort de développement dont profitera l'opérateur en bout de chaîne. La moyenne se situe à 4, des valeurs de 5 ne sont pas exceptionnelles.

3 Modèle économique relocalisé

Notre propos est de montrer que notre conception logicielle donne la possibilité de concevoir un modèle global de développement logiciel qui garantit une meilleure réponse opérationnelle, un meilleur contrôle de la connaissance et une augmentation de l'effort consacré au développement du logiciel.

3.1 Logiciel programmable et certifiant

Un logiciel **programmable** est un logiciel qui n'est pas paramétré par un ensemble de valeurs mais par un authentique programme. Notre modèle repose sur un logiciel **programmable** que l'on appelle un plateforme. Cette plateforme doit en outre avoir la capacité d'être **certifiante**, c'est à dire la possibilité de rendre inviolable des jeux de programmes dans des modules protégeant ce contenu.

Par exemple, le logiciel programmable le plus utilisé est le « *tableur* », il se présente comme une grille ne contenant aucune information où il est possible de coder dans chaque cellule un programme. Ce type de logiciel programmable n'est pas certifiant car on ne peut protéger uniquement les formules tout en préservent les capacités de calcul.

3.1.1 L'auteur

Un logiciel programmable permet l'existence d'un acteur supplémentaire, qui à la fois et programme réside dans le monde « *métier* »

Nous appelons cet acteur l'« *auteur* ». Sa façon de programmer est caractéristique : il procède par duplication/surcharge de programmes existants.

3.1.2 La méthode

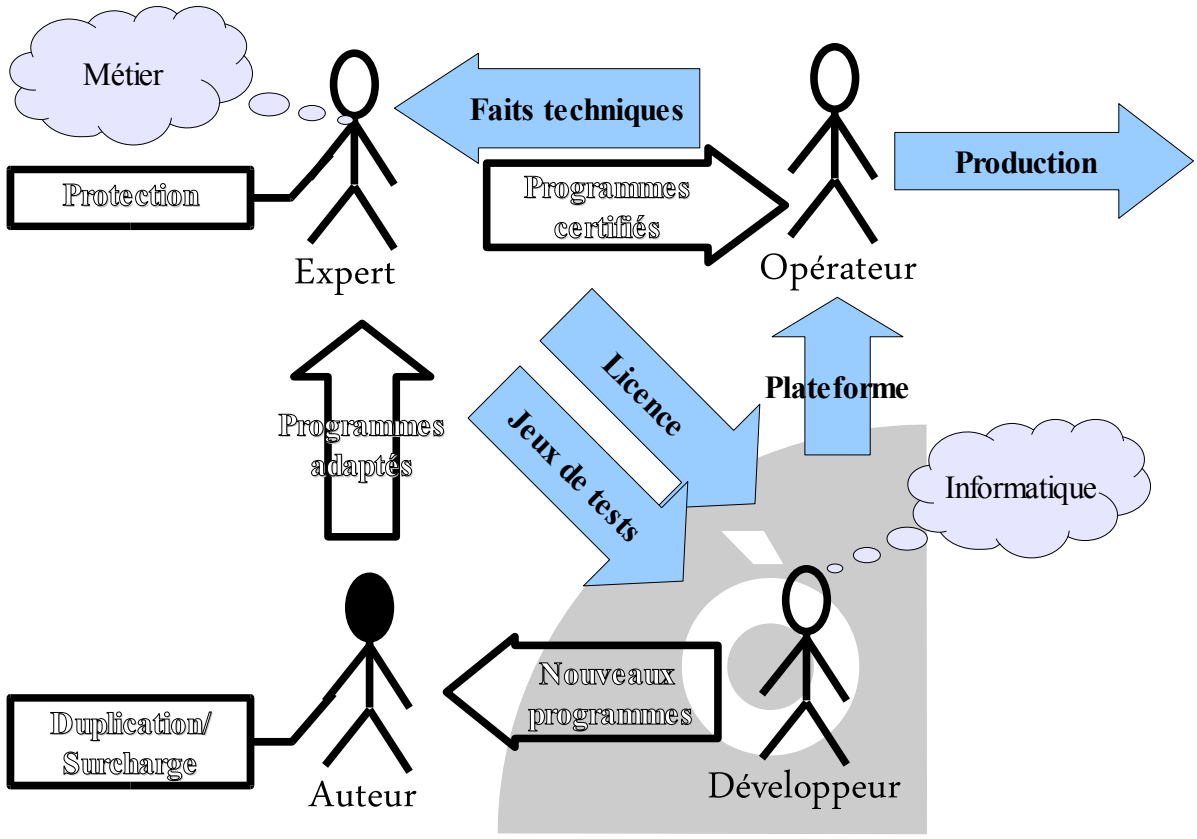
La cœur de notre méthode est de fournir à l'auteur des programme modifiables qui minimise la connaissance nécessaire en informatique pour être adaptés à son besoin. Il est ensuite possible de rendre la modification de ces programmes impossible.

- Le Développeur fournit des langages
- L'Auteur fait des programmes avec ces langages
- l'Expert peut protéger des programmes pour en faire des modules certifiés.
- L'utilisation par l'opérateur de modules non certifiés peut être détecté et donner lieu à des notifications plus ou moins contraignantes.

La double capacité de la plateforme nécessaire pour appliquer la méthode est donc d'être « *programmable et certifiante* ».

3.2 Dynamique des programmes

Le modèle relocalisé fonctionne comme ceci dans sa version standard :



- L'ensemble des acteurs partagent la même plateforme qui est réalisée par les développeurs.
- L'utilisation de la plateforme est associée à une licence définissant un « tarif d'utilisation »
- Un besoin plateforme est définie par un « *jeu de tests* »
- Le fournisseur livre des plateformes à date régulière via un dispositif de mise à jour automatique
- Le fournisseur livre avec les plateformes des programmes potentiellement « nouveaux » résolvant les jeux de tests
- L'auteur à partir des programmes des jeux de tests compose par duplication/surcharge des programmes adaptés aux besoins de l'expert.
- L'expert peut rendre le contenu d'un programme inviolable et le distribuer aux opérateurs par le même moyen que les plateformes.

La protection est un étape essentielle du processus. Cette protection est la fonctionnalité supplémentaire par rapport aux logiciels programmables existant que la plateforme doit détenir pour pouvoir être intégré dans ce type de modèle économique.

A cette étape-clé, il est tout à fait possible de réaliser toutes sortes de calculs permettant d'optimiser

à la fois le produit et le processus.

- Le programme peut être *compilé* dans un langage plus proche de la machine hôte en garantissant que son comportement restera identique. Ceci permet de réaliser toutes sortes d'optimisation statique au moment de cette compilation.
- En outre, il est possible à ce moment de vérifier la qualité du code ainsi protégé pour détecter les signes avant-coureurs de blocages d'usure par l'utilisation d'indicateurs.

La protection s'apparente à une compilation et donc ne pose pas de problème technique informatique, la compilation est une compétence cardinale de l'informatique, enseignée en maîtrise.

3.3 Pourquoi ça marche ?

Cela fonctionne essentiellement parce que ce modèle traite plus rationnellement les problématiques de la fourniture de logiciel :

- **Séparation des compétences** : chacun se concentre sur son métier, la coopération est plus claire, plus sereine, la productivité globale s'en ressent.
- **Suppression de l'« effet bouton »** : les corrections peuvent dans le cas général être fait en interne, ceci permet d'abaisser considérablement le « *temps de correction* » en le découplant du temps de réponse qui concerne les jeux de tests et les programmes associés.
- **Suppression de l'« effet tunnel»** : en optant pour une définition opérationnel des attendus sous forme de jeux de tests, notre méthode simplifie et rationalise l'expression du besoin.

Plus généralement, penser au niveau des langages donne les moyens penser les problèmes contemporains de fourniture de logiciel. En particulier, le développeur ne fournit plus un programme mais un langage, un ensemble de constructions qui vont pouvoir être combinées par l'auteur pour couvrir son besoin « métier ».

3.3.1 Séparation des compétences

Le premier problème que règle cette pratique de développement est le problème de la séparation des compétences et le problème des dépendances croisés de propriété industrielle :

- **La plateforme appartient au fournisseur**, elle concentre la compétence informatique. L'entreprise client n'est pas propriétaire des sources de la plateforme.
- **Les programmes appartiennent au client**, ils concentrent la compétence métier. Le fournisseur peut n'y avoir aucun accès direct sans que cela nuise au processus de développement.
- **Les jeux de test et les programmes de réponse sont publics** (les « *nouveaux programmes* »). Le client peut, à titre exceptionnel, souhaiter qu'un jeu reste confidentiel mais ceci ne peut former une pratique habituelle.

Le système peut fonctionner avec une étanchéité totale de compétences entre les deux mondes, cette étanchéité a un coup pour le client mais elle est techniquement réalisable et des processus peuvent être déployés chez le client à cette fin. A toutes fins utiles, l'approche résout aussi un problème annexe mais souvent épineux de l'obtention des supports commerciaux pour le fournisseur.

Cette séparation des compétences a plusieurs effets importants. En particulier la compétence métier

peut être relocalisé dans l'entreprise cliente. Si l'entreprise souhaite délocaliser cette compétence, par exemple en sous-traitant le travail des auteurs à des sociétés off-shores, ceci forme un choix qui n'est plus du ressort des politiques du fournisseur mais seulement du client.

Nous pensons que la dispersion des connaissances critiques par le biais de la sous-traitance de l'outillage informatique est une cause qui n'est pas à négliger dans nos problèmes économiques actuels. Une façon de remettre un frein à la phase « *tout le monde il est beau, tout le monde il est gentil* » promu *de facto* par le modèle externe et combattu de façon artificielle par une agressivité juridique et une méfiance inter-personnelle au jour le jour qui n'est pas sans avoir un impact sur la santé du corps productif et donc sa productivité.

3.3.2 Suppression de l'effet bouton

D'avoir relocalisé une partie de la production de programme au sein de l'entreprise cliente permet aux acteurs de cette entreprise de reprendre la main sur leur développements « métier ». Mais ce contrôle est à deux niveaux :

- Les auteurs ont une maîtrise des programmes définissant la partie métier du logiciel final. Si cette partie est sujette a des divergences, ils peuvent y remédier par leurs propres moyens. Les expériences sur le terrain montre que si le langage de programmation est bien fait, ils peuvent faire vivre des programmes pendant des années sans difficultés majeures.
- Les experts conservent les capacités de validation et de certification acquise pendant la période externe. Le fournisseur peut fournir les outils et les connaissances nécessaires pour que la gestion de la flotte des programmes se passe au mieux.

L'avantage majeur de la technique, outre la relocalisation des activités, est la disparition de l'« effet bouton ». Des améliorations mineures sont pensées pour être réalisées par les auteurs. Et de fait l'expérience montre qu'ils sont parfaitement autonomes en l'absence d'experts.

3.3.3 Suppression de l'effet tunnel

Pour attaquer ce dernier point il faut détailler un peu comment se déroulent les premières phases. L'objectif de ces premières phases est de parvenir à exprimer un « *jeu de test élémentaire* » permettant à l'expert de couvrir ses besoins fonctionnels.

L'idée est de toujours discuter sur du concret, et éviter au maximum l'écriture d'éléments annexes qui n'ont aucune utilité fonctionnelle. Les jeux de test ont de multiples utilités fonctionnelle comme par exemple de permettre la qualité de toute livraison par passage automatisé des jeux de tests. Même dans le cadre de sous-traitance multi-niveaux, la discussion sur les jeux de tests est beaucoup plus productive et créatrice de valeur que d'introduire un énoncé supplémentaire n'ayant aucune utilité fonctionnelle.

A partir de ce moment là, l'expert à une réponse quasiment immédiate qui n'est pas adapté à son besoin métier mais avec laquelle il peut « *jouer* » et ainsi évaluer « *en conscience* » si les capacités qui lui sont démontrées sont à même d'être combinées pour « *rencontrer son besoin* ». Il n'a pas à attendre pour disposer de cette confiance, il peut l'acquérir par ses propres moyens très rapidement en expérimentant des énoncés qui peuvent être simple mais qui combinés donne les potentialités souhaitées.

3.4 Réorganisation isofonctionnelle

Notre approche ne supprime pas les effets de l'usure du code, au contraire elle identifie expressément le lieu de l'usure et par cette identification se donne les moyens de la contrôler et d'y remédier.

Lorsque le logiciel final vient à s'user, il peut être intéressant de procéder à une « réorganisation isofonctionnelle », en remplaçant en bloc un jeu de programmes certifiés par un autre en garantissant la non-régression sur un jeu de test – qui dans ce cas particulier n'est pas publiables. Il est même intéressant de constater que cette capacité de refonte isofonctionnelle s'applique aussi à la plateforme.

Ceci nous permet d'identifier l'ensemble du cycle de vie d'une ligne de code dans un logiciel programmable : une ligne de code est écrite par un développer puis dupliquée et surchargée par des auteurs pour finalement disparaître lors d'une « *réorganisation isofonctionnelle* ».

4 Exemples de réalisations

4.1 Réalisation passées

Dans cette partie nous identifions nos différentes expériences de logiciel programmables et comment nous avons tenté de mettre en place cette méthode à travers divers essai pour laquelle elle forme un indiscutable fil rouge.

4.1.1 Lancer des calculs depuis une feuille de calcul

En 2004, suite à un contact au sein d'une importante société aéronautique, je parviens à vendre un tout petit logiciel pour un tout petit prix mais sans livrer les sources. Ce logiciel permet de post-traiter des feuilles de calcul sous forme de fichier texte devant partir au calcul. Son originalité est d'être paramétré par un template qui permet de piloter le texte produit au caractère près.

En 2009, le logiciel s'est répandu dans l'entreprise. Comme il compte quelques centaines de lignes, elles sont fiables depuis longtemps et les auteurs ont appris à fonctionner sans développeur car j'ai quitté la société en question au printemps suivant ! Je n'arrive même pas à identifier les utilisateurs qui craignent d'être en tort... C'est en forgeant que le devient forgeron. L'erreur consiste à ne pas avoir muni ce logiciel d'un système élémentaire de contrôle de flotte.

Sans doute l'expérience la plus aboutie, tellement économe pour le client qu'elle s'est révélée sans intérêt commercial à court terme pour le fournisseur !

4.1.2 Passer des tests embarqués sur un PC

Historiquement la première application industrielle que j'ai développée. En remplaçant une partie d'un code embarqué par un code de simulation, on peut arriver à faire tourner un logiciel embarqué sur un PC. La plateforme sert à faire tourner les tests propose un langage pour les scénarios et une API pour définir le « *code de simulation* ».

Après six années passées à développer cette solution, je n'arrive toujours pas à convaincre mes interlocuteurs du caractère « *métier* » du « *code de simulation* », de leur devoir s'y intéresser car cela relève de leur compétence. Lors de la refonte du logiciel qui a permis en partie la création de

mezzÒnomy, ce problème avait été identifié et répété constamment à toutes les réunion.

Mais rien n'y fait, prisonnier de leur statut d'expert rédigeant des documentations absconses et détachés des contraintes opérationnelles, toucher à un code leur apparaît une dégradation de leur condition. Moralité, c'est un jeune Indien qui passait par là qui a acquis ces connaissances critiques sur le composant central de la technique de simulation !

Et ceci fut parfaitement justifié par mon interlocuteur par le fait que le logiciel allait être déployé en Inde. Ce faisant il ne se rendait même pas compte qu'après lui avoir abandonné son rôle d'auteur dans le système il lui abandonnait *de facto* celui d'expert...

A part ça, tout va très bien le logiciel fonctionne à merveille, la plateforme n'a plus été pris en défaut depuis Janvier 2009 et un problème alambiqué de conflit d'optimisations...

4.1.3 Mutualiser des méthodes de calcul

Changement d'échelle, ma dernière expérience de développement d'une plateforme programmable concerne des milliers de postes potentiels. Il s'agissait de mettre en commun les méthodes de calculs de l'ensemble des bureaux d'études d'un grand groupe aéronautique.

Là, je me suis heurté à un obstacle de nature différente, l'impossibilité pour les décideurs du système à distinguer le « *développeur* » de l'« *auteur* », dans leur esprit toute personne touchant des programmes est un développeur, comme dans le modèle **externe**. Alors que la plateforme est **programmable** et pouvait être rendue **certifiante** sans grande difficulté technique.

Cette impossibilité à distinguer la double-compétence et les flux de responsabilité dans le système, a mené à des confusions entre les rôles, la nature des langages à promouvoir, la capacité à répondre à des demandes. Le résultat est un coût de développement pharaonique sans aucun rapport avec l'augmentation de productivité effective, l'aspect d'harmonisation permettant seul de justifier de telles dépenses.

Plus généralement, l'impossibilité de pouvoir réfléchir au niveau global, contrainte fixé par le client pour de raisons de contrôle opérationnel a *de facto* induit des contraintes très fortes de cloisonnement qui ne sont pas les plus à même d'optimiser les conditions de productions de données scientifiques.

Cette expérience nous amène à penser que pour attaquer ce type de compte, il faut déjà avoir une plateforme opérationnelle pour tout de suite amener un contenu à l'expert et tenir ferme sur la relocalisation. Or sans soutien politique, cette dernière condition est impossible. Nous sommes encore très loin du compte pour approcher ces grands comptes.

4.2 Réalisation possibles

Aujourd'hui nous avons une vision claire de l'environnement et disposant d'une méthode ayant déjà été utilisé permettant de réduire significativement les coût de fourniture logiciel et de relocaliser les connaissances métier chez le client.

Nous pouvons appliquer cette méthode selon trois modalités principales :

1. **A la demande** : le client souhaite faire développer un logiciel programmable qui lui est propre.
2. **Toolzòne** : une plateforme orientée fichier et transformation de données. Une première

version déployée à la rentrée.

3. **Mirza** : une plateforme étendant les tableurs du marché pour les rendre coopératifs et certifiants. Un prototype existe mais il reste encore beaucoup de travail.

Toute plateforme programmable et certifiante peut être utilisée comme support à la méthode relocalisante.

4.2.1 A la demande

Un client souhaite disposer d'un outil programmable et certifiant sur un domaine où il n'y a pas de réutilisation possible ou pour lequel il souhaite disposer du droit de propriété sur les sources.

Ce cas de figure est le cas de lancement pour notre société car elle ne suppose pas de capital important antérieur au contrat.

La méthodologie consiste à

1. se focaliser sur la notion de jeu de test en faisant écrire et dessiner le client son besoin jusqu'à arriver à exprimer des scénarios de test.
2. établir la première liste de programme capable de couvrir ces jeux de tests.
3. réaliser les interpréteurs pour l'auteur et le compilateur pour l'expert et le système peut commencer à itérer.

La viabilité du modèle repose sur notre savoir-faire en matière de compilateur et d'interpréteur. C'est cette compétence technique qui associé au modèle de développement donne sa puissance à l'approche générale.

4.2.2 Toolzòne: Logiciel programmable de première génération

mezzÒnomy a développé sur fond propre un petite suite de logiciel permettant de transformer des fichiers de donnée par une « grappe de logiciels programmables ». Ce jeu d'outil sera mis sur le marché à la rentrée 2009, sans être très « *sexy* », ils peuvent rendre de bons services et disposent de toutes les capacités nécessaires pour tester en vrai grandeur le modèle économique.

Nous avons investi de l'ordre de 5000 € dans la réalisation de cette « grappe d'outil » et espérons être proche de pouvoir le distribuer pour un cout de licence et un coût de certification. Nous sommes en train de mettre en place le système de tarification.

Pour des développements dédiés dans la Toolzòne, la base du contrat-type consiste en une promesse d'acquisition de licence si tel ou tel le jeu de test est implémenté. Si le système n'est pas rentable pour la Toolzòne, nous revenons au mode sur mesure.

4.2.3 Mirza: Logiciel programmable de deuxième génération

mezzÒnomy a développé sur fond propre (3500 €) un prototype non encore fonctionnel d'un logiciel programmable très ambitieux pouvant se substituer aux tableurs du marché tout en étant utilisable dans le cadre du modèle relocalisant.

Le coût de développement de cette plateforme est estimé à 400.000 €.