

SoftFluent

Comment le développement logiciel
piloté par les modèles peut vous
aider à réussir vos projets

Livre Blanc CodeFluent Entities

Daniel COHEN-ZARDI

Version 2.0

Juin 2011

Sommaire

Ce livre blanc a pour objectif de détailler le défi que représente le développement logiciel, et en quoi SoftFluent parvient à le relever grâce à la fabrique logicielle pilotée par les modèles CodeFluent Entities et la méthodologie que nous lui avons associée.

La première partie de ce document expose les défis posés par le développement et pourquoi il est difficile d'y répondre :

- Au Chapitre 1, nous expliquons la nature du défi du développement logiciel,
- Au Chapitre 2, nous commentons les approches le plus fréquemment rencontrées sur le marché,
- Au Chapitre 3, nous détaillons nos observations relatives aux échecs de certains projets, particulièrement liés aux nouvelles technologies (dont l'ouverture comporte plus de risques que les solutions traditionnelles, plus encadrées),
- Au Chapitre 4, nous proposons notre propre "Equation du succès" comme modèle pour expliquer ces échecs mais aussi les composantes d'un succès possible.

Il est intéressant de noter que cette partie est largement applicable et qu'elle concerne toutes les personnes qui s'intéressent au développement logiciel, indépendamment de notre offre.

Dans la seconde partie, nous exposons la solution que nous avons élaborée pour répondre structurellement au défi mentionné :

- Au Chapitre 5, nous présentons les convictions intimes qui nous ont guidés dans l'élaboration de notre solution afin de dépasser les limites des approches traditionnelles,
- Au Chapitre 6, nous détaillons les différentes étapes de notre méthodologie liées à l'outil CodeFluent Entities pour modéliser et industrialiser le développement logiciel,
- Au Chapitre 7, nous listons des situations typiques pour illustrer les bénéfices concrets liés à l'utilisation de CodeFluent Entities.

Finalement, le but de CodeFluent Entities est d'être une recette concrète pour le succès des projets de développement logiciel. Elle a déjà démontré qu'elle pouvait apporter une valeur considérable.

En conclusion au chapitre 8, nous expliquerons pourquoi le développement piloté par les modèles est la seule solution durable à l'enjeu de l'évolution tel qu'il se pose aux directions informatiques.

© SoftFluent, 2010-2011 - Ne pas dupliquer sans autorisation.

CodeFluent Entities est une marque déposée de SoftFluent. Tous les autres noms de société et de produit sont des marques déposées de leurs propriétaires respectifs.

Les éléments présentés dans ce document sont par nature synthétiques, sujet à évolution, non contractuels et communiqués uniquement à des fins d'information.

Table des matières

| | | |
|-------|--|----|
| I. | Le défi du développement logiciel | 4 |
| II. | Approches traditionnelles | 5 |
| A. | Outils RAD | 5 |
| B. | Ateliers de Génie Logiciel | 5 |
| C. | Off-shore | 6 |
| III. | Notre expérience "terrain" des échecs | 8 |
| A. | Manque de compétences | 8 |
| B. | Manque d'expérience et "geek attitude" | 8 |
| C. | Sur-ingénierie et 'frameworks' | 9 |
| IV. | Proposition de modèle pour une "Equation du succès" | 11 |
| V. | Nos convictions | 12 |
| VI. | Méthode Agile CodeFluent Entities | 13 |
| A. | Les principes de CodeFluent Entities | 13 |
| B. | Aperçu de CodeFluent Entities | 14 |
| C. | L'agilité avec CodeFluent Entities | 15 |
| D. | Etapes recommandées pour un usage basique | 16 |
| E. | Etapes recommandées pour une utilisation avancée | 20 |
| VII. | Bénéfices de CodeFluent Entities: 12 exemples de scénarios | 23 |
| VIII. | Conclusion | 26 |
| IX. | Annexes | 28 |

I. Le défi du développement logiciel

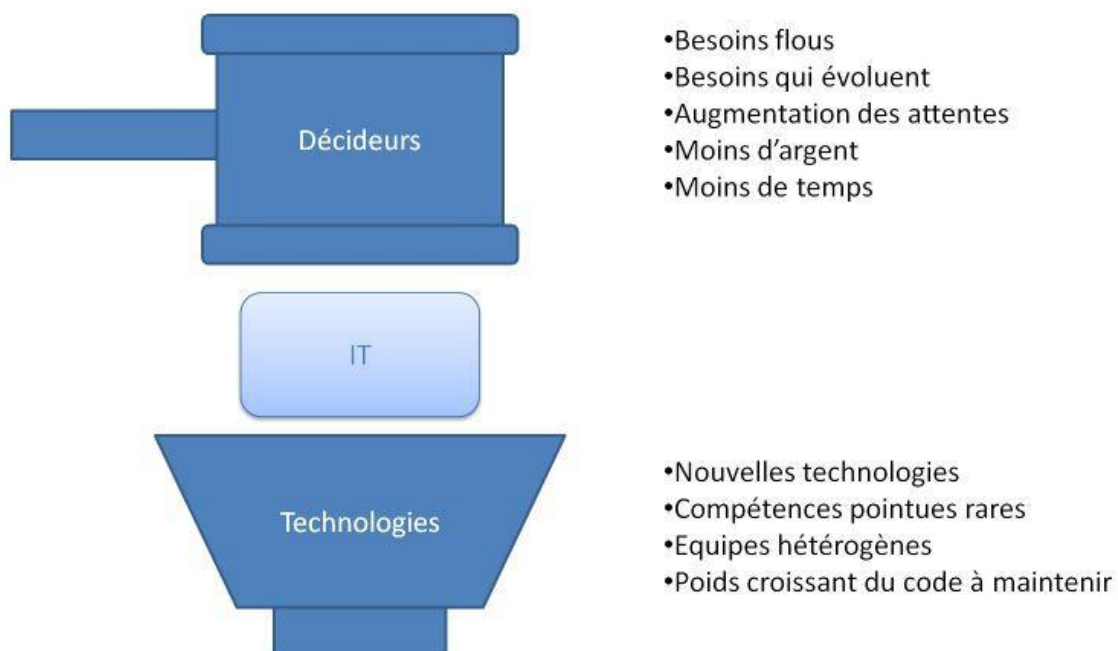
Etre en charge d'une équipe de développement logiciel dans les années 2010 n'est pas le poste le plus enviable qui soit. Alors que les attentes des utilisateurs ne cessent d'augmenter, les moyens s'accompagnent de plus en plus de contraintes.

Quand un secteur progresse, cela se traduit inévitablement par des besoins fonctionnels de plus en plus sophistiqués ; parallèlement, les utilisateurs professionnels attendent de plus en plus des technologies. Le fait qu'ils utilisent des logiciels packagés (produits pour des millions d'utilisateurs avec des investissements conséquents) dans leur vie quotidienne n'aide pas l'équipe IT à contenir les attentes des utilisateurs professionnels.

En fait, l'équipe de développement doit **digérer l'innovation technologique**, et apporter des solutions prêtes à l'emploi. Le défi est de plus en plus difficile à relever dans la mesure où le rythme de l'innovation technologique s'accélère avec une complexité accrue difficilement gérable.

A cela il convient d'ajouter une pression à laquelle les équipes IT doivent faire face, notamment les réductions de budgets et de délais, et l'on comprend alors que les développeurs sont structurellement coincés entre des attentes fonctionnelles croissantes et l'évolution de la technologie.

Défi des développeurs



A ce stade, vous pourriez être tenté d'embrasser une autre carrière, et nous vous recommanderions peut-être de le faire si ce n'est pas trop tard ! Mais si, comme c'est notre cas, il est trop tard, nous vous invitons à poursuivre votre lecture.

II. Approches traditionnelles

Le développement logiciel est une discipline âgée d'une cinquantaine d'années, on peut donc s'étonner du fait qu'aucune solution n'existe à ce jour. Passons en revue les différentes approches qui ont été tentées et faisons la lumière sur leurs principales limites.

A. Outils RAD

Le temps du développeur c'est de l'argent, c'est pourquoi les outils de génie logiciel ont longtemps misé sur l'accélération du processus de codage, et ce, par différents moyens (en particulier la génération de code).

Certains outils connus, catégorisés sous la notion de RAD (Rapid Application Development), tels que PC Soft Windev ou 4D, procurent actuellement un bon moyen de livrer une application dans un délai relativement court. Différents prototypes peuvent rapidement aboutir à des résultats alléchants avec un processus simple.

Cependant, si ces applications atteignent un certain niveau de complexité, les outils se révèlent souvent incompatibles avec la sophistication des fonctionnalités demandées, dans la mesure où ces outils sont très efficaces pour répondre à leur fonction de base, mais incapables d'une quelconque valeur ajoutée dès lors que celle-ci n'a pas été prévue au départ.

Le principal problème pour les clients est alors d'être incapable de répondre aux besoins fonctionnels de l'application. C'est un risque inacceptable pour les clients qui ont des besoins avancés.

B. Ateliers de Génie Logiciel

Les Ateliers de Génie Logiciel (AGL) existent depuis longtemps, et ont été largement vulgarisés pendant la vague client/serveur des années 90.

Les outils tels que Powerbuilder, Progress, Magic, NS-DK ou Centura ont pris leur essor à cette époque. Ils ont beaucoup investi pour fournir un modèle de programmation pour les applications client/serveur et ont rencontré un réel succès à la fois sur le marché mais aussi en gains de productivité. Ils simplifiaient notamment les langages complexes tels que C++, langage standard à l'époque.

Mais, ces AGL ont perdu de leur popularité essentiellement en raison de l'émergence d'un tout nouveau modèle de programmation avec le Web. La conséquence a été que bon nombre de clients, poussés par les vendeurs de plateformes logicielles, se sont à nouveau focalisés sur des langages tels que Java ou C# perdant ainsi le gain qu'ils avaient acquis en utilisant des outils à haut niveau d'abstraction.

Une autre raison était la crainte de certains clients d'être verrouillés dans l'environnement d'un fournisseur, dans la mesure où ces outils fournissaient leur propre langage de programmation.

Finalement le principal problème pour les clients relevait de leur incapacité à se maintenir à jour des technologies, dans la mesure où la contrepartie de ces offres était la lenteur de leur évolution. En fait, à ce jour il n'existe pas d'AGL efficace pour les applications Web. Ce n'est pas une situation tenable à long terme pour les clients.

C. Off-shore

Une autre approche que beaucoup de sociétés ont tentée notamment dans les années 2000 a été de réduire le coût des ressources via l'off-shore.

Bien que cela semble économiquement alléchant avec des coûts de main d'œuvre allant de 1 à 10 suivant les pays, cette solution est loin de tenir ses promesses.

En fait – lorsque cela fonctionne¹ – cela ne répond pas fondamentalement au défi, mais diminue simplement le coût apparent des ressources de développement, tout en ajoutant par ailleurs de nouvelles difficultés et en faisant peser de nouveaux risques sur le projet.

On pourrait écrire un livre sur l'off-shore, mais pour résumer, éloigner l'équipe de développement l'empêche d'augmenter son savoir-faire et ne répond pas à notre défi.

Au contraire, les personnes expérimentées en développement logiciel savent que le coût total d'une application doit inclure les coûts de maintenance. Cette phase est bien souvent plus longue que la phase de développement initial.

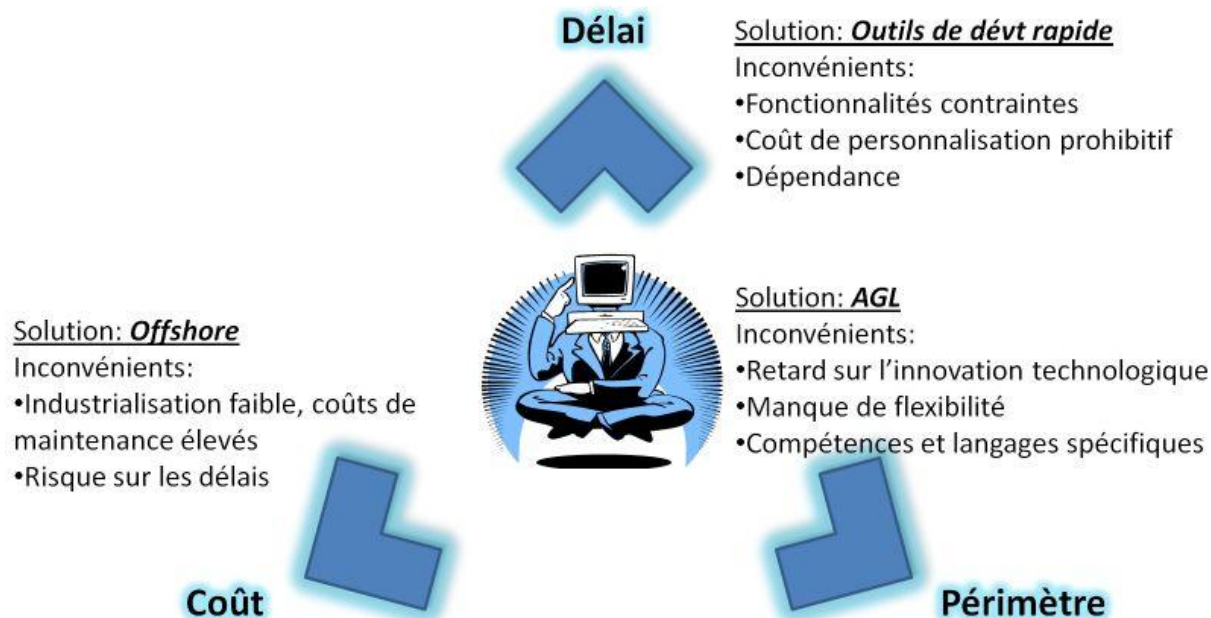
Et ce qui peut paraître économique à court terme se transforme souvent en coûts additionnels et en perte d'agilité à long terme. Tout développeur expérimenté sait que les possibilités d'évolution d'une application mal conçue sont faibles alors que toute évolution, même mineure, devient de plus en plus coûteuse.

Le principal problème avec l'off-shore est qu'il est souvent guidé par une pure recherche de réduction des coûts, avec au final un projet livré au détriment de l'industrialisation et de la maintenance.

Selon nous, ces 3 approches comportent des limites intrinsèques en plaçant l'accent de façon exagérée sur respectivement les délais, le périmètre et les coûts.

¹ Voir notamment le livre blanc de la commission R&D de l'AFDEL (Association Française des Editeurs de Logiciels) intitulé "La conduite de l'innovation logicielle" qui y consacre un chapitre.

Approche générale : focalisation sur un composant de l'ambition



❏ *Aucune de ces solutions ne relève réellement le défi structurel de l'innovation technologique...*

Ces trois paramètres doivent être équilibrés de manière appropriée et c'est pourquoi nous privilégions une autre méthode pour relever le défi du développement logiciel.

III. Notre expérience "terrain" des échecs

Parce que nous avons de l'expérience dans le développement logiciel, il nous est souvent demandé d'effectuer des missions d'audit sur le développement .NET pour analyser la qualité d'une application, sa capacité d'évolution, sa flexibilité et son exploitabilité dans un contexte de production.

En premier lieu, il est important de préciser que nous sommes étonnés par la grande proportion des projets qui échouent et l'accumulation de défauts. Nous rencontrons souvent des échecs se chiffrant en millions d'euros et il est assez commun d'en conclure qu'une bonne moitié du code pourrait être évitée en utilisant des bibliothèques appropriées, la plupart du temps disponibles dans le Framework .NET lui-même !

Il est intéressant de noter l'existence de syndromes typiques que nous avons observée chez certaines équipes de développement et que nous résumons ci-dessous.

A. Manque de compétences

La première raison est le simple manque de compétences.

Dans la plupart des cas, même les développeurs expérimentés peuvent manquer de compétences dans les nouvelles technologies. Alors que la complexité des couches techniques était généralement 'masquée' avec les AGL, ce n'est désormais plus le cas avec le nouveau défi posé par la large ouverture des environnements, qui mène à encore plus d'écueils.

Dans cette situation, nous constatons que le code fonctionne... jusqu'à ce que le projet se complexifie en périmètre ou en volumétrie. Comme le code est produit manuellement, avec des styles hétérogènes fluctuants, la maintenance est de plus en plus compliquée avec une inflation des coûts pour la moindre modification jusqu'à ce que l'équipe se retrouve finalement face un mur.

Dans ce cas, les **compétences** sont le 1^{er} facteur d'échec.

B. Manque d'expérience et "geek attitude"

A contrario, nous rencontrons souvent de jeunes développeurs plutôt intelligents et à l'aise avec les nouvelles technologies. Mais ils peuvent manquer d'expérience entreprise et la plupart d'entre eux ont tendance à adopter la 'geek attitude', un comportement qui consiste à se focaliser sur chaque nouvelle technologie, dès qu'elle apparaît.

Dans ces situations, les développeurs se concentrent sur la beauté du code, une seule ligne de code pour une instruction très compliquée, qui satisfait son ego de développeur mais n'apporte aucune valeur intrinsèque. En fait, cela peut même porter une valeur potentiellement négative et accroître les coûts de maintenance.

Ces jeunes développeurs perdent souvent le sens des objectifs fonctionnels dans la mesure où les utilisateurs ne sont pas leur principal centre d'intérêt ; de plus, ils n'ont pas le niveau d'expérience pour comprendre les contraintes d'exploitabilité et les nécessités d'instrumentation du code.

En raison de ce manque d'expérience, il n'est pas rare qu'ils sous-estiment les enjeux de flexibilité, et les applications qu'ils conçoivent sont généralement peu performantes avec l'augmentation du volume de données et du nombre d'utilisateurs.

Ce genre d'applications est généralement difficile à maintenir et nécessite d'être optimisé voire réécrit dans les domaines nécessitant de la performance.

Ce type d'échec traduit en fait un manque de **méthode éprouvée**.

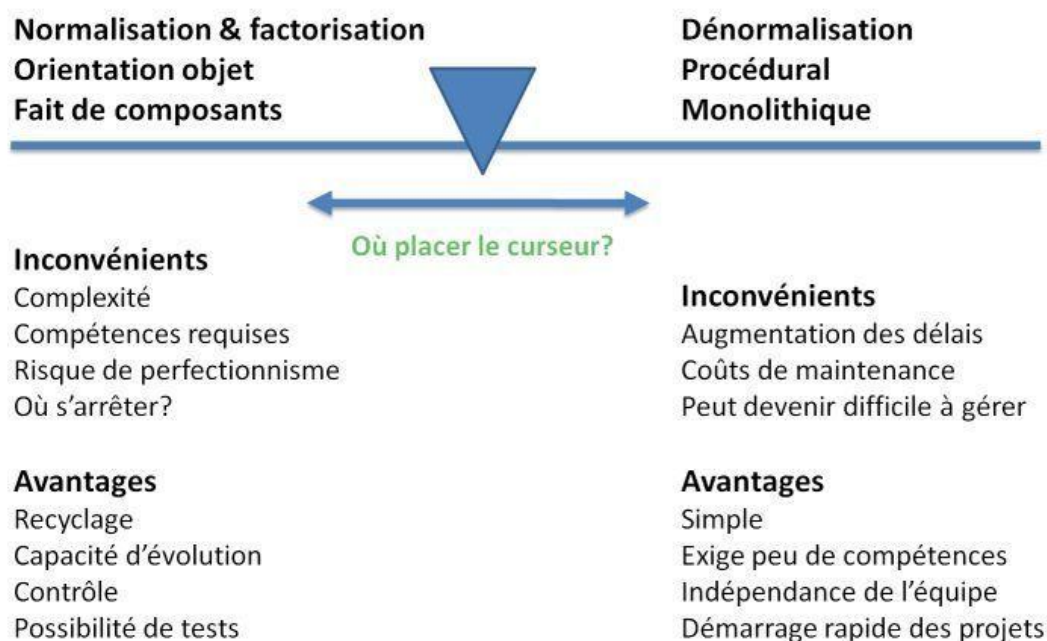
C. Sur-ingénierie et 'frameworks'

Ce type de cas mène aux situations les plus coûteuses et concerne essentiellement les grands comptes.

Dans ce 3ème syndrome, un (ou plusieurs) architecte de génie est mandaté pour inventer une solution personnalisée pour l'équipe de développement. Généralement cet architecte est très intelligent, et c'est en partie le problème, car il se croit capable d'inventer toutes les solutions pertinentes pour le projet. Il commence donc à concevoir – potentiellement avec une équipe de développeurs – un 'framework' supposé être la recette optimale pour le projet mais elle va souvent bien trop loin.

Mettre le bon niveau d'abstraction pour un projet nécessite de placer le curseur au bon endroit sur le schéma ci-après :

Mettre le bon niveau d'abstraction



Le "Où s'arrêter?" à gauche est une question cruciale, car selon notre expérience, c'est là que la situation se détériore alors même que l'idée de concevoir un framework peut paraître bonne.

L'expérience montre que :

- Concevoir un framework flexible susceptible de faire face à toutes les situations avec les nouvelles technologies évolutives est ardu et risqué,
- Le risque d'échec avec un framework qui ne fonctionne pas est énorme, avec potentiellement un "effet tunnel" pendant le développement du framework, cette phase est alors perdue inutilement,
- Dans les cas les plus favorables, le framework apporte de la valeur mais ne résiste pas à une nouvelle vague technologique, à moins de réinvestir fortement pour être toujours en adéquation avec les nouvelles technologies et prévenir un autre échec,
- Dans la plupart des cas, le framework ne peut pas être maintenu par une personne autre que l'architecte lui-même.

Bien que n'étant pas toujours admise, cette situation provoque des échecs économiques significatifs.

Si, parfois, les AGL sont perçus comme un problème en plaçant le client dans une situation de dépendance vis-à-vis d'un fournisseur, le syndrome décrit ici, à savoir la dépendance vis-à-vis d'un architecte, est bien pire. Le risque consistant à dépendre d'une seule personne est autrement plus inquiétant encore que de dépendre d'un fournisseur établi dont la stratégie repose sur une offre conçue pour cela.

Ce cas d'échec est caractérisé par un surinvestissement dans l'**outillage**, un facteur qui est difficile à maîtriser et dont la complexité est souvent fortement sous-estimée.

IV. Proposition de modèle pour une “Equation du succès”

Beaucoup d'études menées sur le développement logiciel font le même constat d'un gros pourcentage d'échecs des projets. Dans une étude internationale récente citée par l'AFDEL, il s'avère que seuls 30 % des projets sont considérés comme concluants et seulement 13 % aboutissent dans les délais impartis.

En fait, la probabilité de succès d'un projet peut se résumer à l'équation suivante :

$$\% \text{ Succès} = \frac{\text{Savoir faire}}{\text{Ambition}}$$

Si le savoir faire est supérieur aux ambitions, le projet a toutes les chances de réussir. Cela apparaît finalement simple à comprendre.

Si on ajoute à cela les considérations liées aux attentes évoquées dans l'introduction, on peut considérer que:

$$\text{Ambition} = \frac{\text{Périmètre fonctionnel}}{\text{Budget} \times \text{délai}}$$

Notons que la tendance sur chaque composante fait évoluer défavorablement la situation, et **l'ambition ne cesse d'augmenter de façon cubique !**

Il est clair que les projets de développement logiciel ne peuvent pas réussir sans une approche permettant au savoir-faire d'évoluer plus vite que l'ambition

En fait sans changement structurel d'approche, les équipes de développement **n'ont aucune chance de réussir sur le long terme.**

C'est ce que nous avons pu constater sur le terrain où – bien que souvent masqués – les coûts explosent et le volume de code antérieur obsolète ne cesse de croître, faisant peser le poids de la maintenance sur les équipes de développement.

Donc y a-t-il un espoir de maîtriser l'ambition ou est-ce peine perdue ?

Si l'on considère le savoir faire comme la combinaison de 3 éléments clés constituant les facteurs majeurs de succès des projets de développement :

$$\text{Savoir faire} = \text{Compétences} \times \text{Méthode} \times \text{Outil}$$

Notre équation est **soluble à la seule condition d'améliorer en permanence les compétences, la méthode et les outils.**

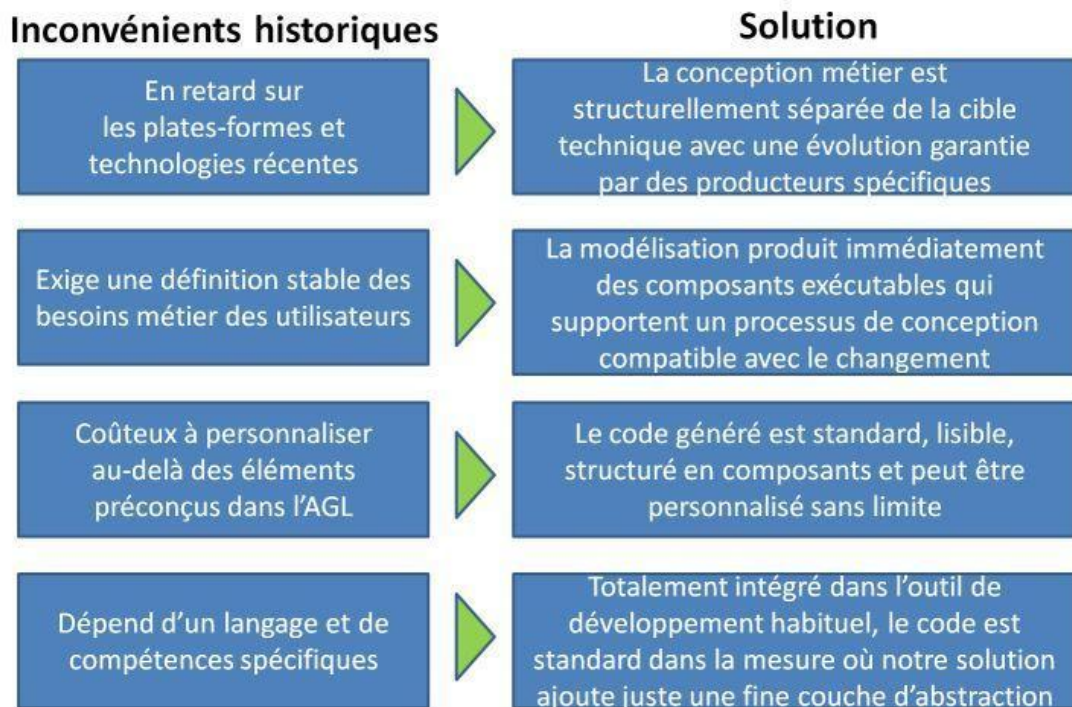
C'est exactement ce que propose notre approche du développement logiciel, basée sur une méthode itérative pilotée par les modèles, appuyée par un outil qui améliore le processus de développement. La responsabilité de l'équipe de développement est, par conséquent, focalisée sur le maintien des compétences, un processus d'autant plus simplifié qu'une partie de la complexité des technologies est automatiquement digérée.

V. Nos convictions

Parce que chez SoftFluent, nous avons passé beaucoup d'années sur des projets de développement, nous avons tiré parti de cette expérience pour concevoir une nouvelle recette susceptible de garantir le succès des projets malgré l'évolution des technologies.

Notre approche du développement est construite sur le succès des recettes passées, telles que les AGL, mais avec quelques différences structurelles ciblées pour relever le défi.

Améliorations de l'approche AGL



L'analyse des inconvénients historiques des AGL nous a aidés à forger nos convictions intimes et constitue le cœur de notre méthode de développement pilotée par les modèles.

1. Les entités métier ont des cycles de vie plus longs que la technologie et doivent donc être définies dans un format qui **résiste aux évolutions technologiques**, qui puisse les supporter sans avoir à reconcevoir l'application.
2. Les règles métier et les changements de processus doivent avoir des **cycles courts** compatibles avec un schéma de maintenance continu.
3. Un **couplage** efficace entre les couches de données, de modèle métier et de présentation incluant tout ou partie du code personnalisé, doit être **garanti par conception**.
4. Les **patterns** de code doivent être homogènes et l'**implémentation automatisée** de sorte que la maintenance puisse être effectuée par des **compétences standardisées**.

VI. Méthode Agile CodeFluent Entities

Maintenant que nous avons abordé nos convictions et les paramètres clés sur lesquels nous avons bâti notre offre, nous allons expliquer, dans ce chapitre, comment travailler avec CodeFluent Entities pour concevoir et produire du logiciel.

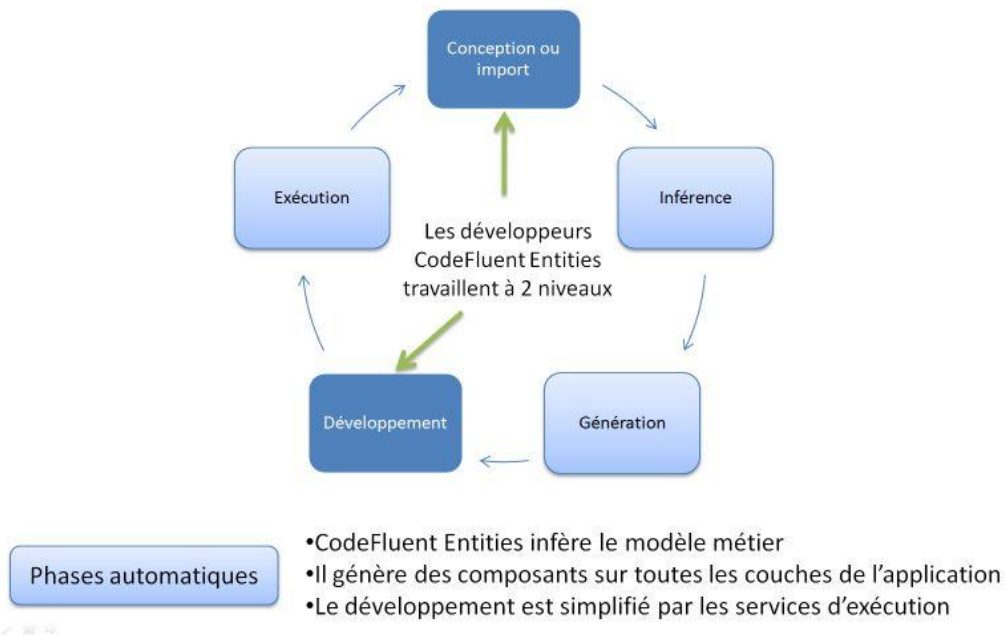
A. Les principes de CodeFluent Entities

CodeFluent Entities est construit sur nos convictions et propose de concentrer l'effort à deux niveaux :

- **Conception des éléments fonctionnels** de sorte qu'ils soient indépendants de la technologie,
- Développement d'un **code personnalisé**, principalement dans les couches métier et les templates de personnalisation des IHM et rapports générés.

Comme décrit dans le schéma ci-après, toutes les autres phases de l'application sont automatiques :

CodeFluent Entities Cycle de vie de l'application



Au-delà du processus qui nécessite de concentrer les efforts à 2 niveaux, la valeur ajoutée des phases d'automatisation intervient dans 3 phases :

- **Analyse du modèle et inférence:** bien qu'invisible, cette phase d'automatisation est extrêmement puissante dans la mesure où elle rend possible le processus de génération élaboré qui suit, ainsi que les combinaisons de comportements complexes.
- **Génération:** les producteurs génèrent des composants prêts à l'emploi sur toutes les couches souhaitées, en totale cohérence avec l'intégration des composants entre eux.
- **Exécution:** les bibliothèques 'runtime' fournissent des services mutualisés sur lesquels s'appuient les composants générés, afin que le processus de développement soit encore plus productif.

B. Aperçu de CodeFluent Entities

Bien que l'objectif de ce livre blanc ne soit pas de donner une vue détaillée du produit, nous savons que nombre de nos lecteurs familiers du développement logiciel voudront comprendre au travers d'une courte démonstration ce que le produit fait exactement et en quoi il se différencie des offres traditionnelles.

Une démonstration complète durerait des heures car le produit couvre de très nombreux aspects du développement et comprend des fonctionnalités complètes pour couvrir tous les besoins. Néanmoins, pour ceux qui veulent matérialiser le produit "en action", nous vous recommandons de regarder cet [aperçu du produit](#) (22 minutes).

Il est également possible de visualiser directement un passage correspondant à un sujet en particulier

| Sujet | URL |
|-------------------------------|---|
| Define Entities | http://www.youtube.com/watch?v=CV5py9C1laQ |
| Define Properties | http://www.youtube.com/watch?v=L-4CQyEBBFc |
| Define Enumerations | http://www.youtube.com/watch?v=nI5QKxpJHlo |
| Collaborate | http://www.youtube.com/watch?v=lo2K8gXeB4Y |
| Meta-Model | http://www.youtube.com/watch?v=WEh17S0tiYA |
| Generate | http://www.youtube.com/watch?v=SXAeJlpKHQ |
| Instances | http://www.youtube.com/watch?v=Hp4a90QlGjU |
| Continuous Generation | http://www.youtube.com/watch?v=lqZHn3AwSqE |
| Methods | http://www.youtube.com/watch?v=0Z4Ju4-5FD4 |
| Define Rules | http://www.youtube.com/watch?v=5oG6KqXRBEs |
| Extend Generated Code | http://www.youtube.com/watch?v=On3GkLAYIMg |
| XML View | http://www.youtube.com/watch?v=K1E4baWe-lQ |
| WPF Smart Client Producer | http://www.youtube.com/watch?v=uw_n11aTNqo |
| SharePoint Web Parts Producer | http://www.youtube.com/watch?v=l7vudNA5fcl |
| Develop Custom Uis | http://www.youtube.com/watch?v=wsZa2I0zCPA |
| Import | http://www.youtube.com/watch?v=VoLCj2uaJws |
| Form Editor | http://www.youtube.com/watch?v=cORcfjRuEgo |
| Navigating in Models | http://www.youtube.com/watch?v=LcNoEWZqucE |
| Customizing Namespaces | http://www.youtube.com/watch?v=daSyJjj9JZ8 |
| Adding Notes | http://www.youtube.com/watch?v=Lio7FVEba_0 |
| Model Grid | http://www.youtube.com/watch?v=3mS84IkJAMs |

Ces vidéos illustrent quelques exemples de fonctionnalités du produit, et nous vous recommandons de suivre le [blog de CodeFluent Entities](#) pour un aperçu plus complet et régulier de ses possibilités.

C. L'agilité avec CodeFluent Entities

CodeFluent Entities ne requiert pas formellement de suivre une méthode agile dans la mesure l'on peut tirer parti du produit en utilisant une méthodologie classique de type "cycle en V". En fait, plusieurs projets utilisant CodeFluent Entities ont été livrés en utilisant l'approche traditionnelle basée sur une phase de spécification préliminaire, essentiellement pour des raisons d'engagement contractuel lié à la tradition de fonctionnement "au forfait" fortement ancrée dans la culture française.

Cependant, CodeFluent Entities est particulièrement efficace pour mener à bien des projets conduits selon une méthode agile. Les méthodologies agiles sont basées sur le principe du développement itératif, qui nécessite que le développement logiciel fonctionne à chaque fin d'itération. Itération après itération, le périmètre fonctionnel s'étend jusqu'à ce que le logiciel présenté aux utilisateurs soit suffisamment satisfaisant pour entrer en production ou être mis en vente sur le marché.

CodeFluent Entities propose plusieurs attributs qui se révèlent particulièrement intéressants lorsqu'ils sont utilisés conjointement à la méthode agile :

- Les éléments fonctionnels conçus dans CodeFluent Entities génèrent immédiatement des **composants exécutables**,
- Les spécifications n'ont pas besoin d'être définies en amont, le modèle peut être enrichi et re-généré autant de fois que nécessaire sans rien perdre du code spécifique,
- Les architectures ciblées peuvent être choisies plus tard dans le cycle en changeant simplement le(s) producteur(s) et le code spécifique des couches concernées.

Par conception CodeFluent Entities propose un processus de développement **itératif** et **intégré** reposant sur une **modélisation continue**.

Pour beaucoup de clients, la modélisation métier a commencé avec les utilisateurs et s'est transformée rapidement en première version de modèle grâce à l'opportunité que l'équipe de développeurs CodeFluent Entities a saisi de leur montrer immédiatement des interfaces utilisateurs représentant le besoin.

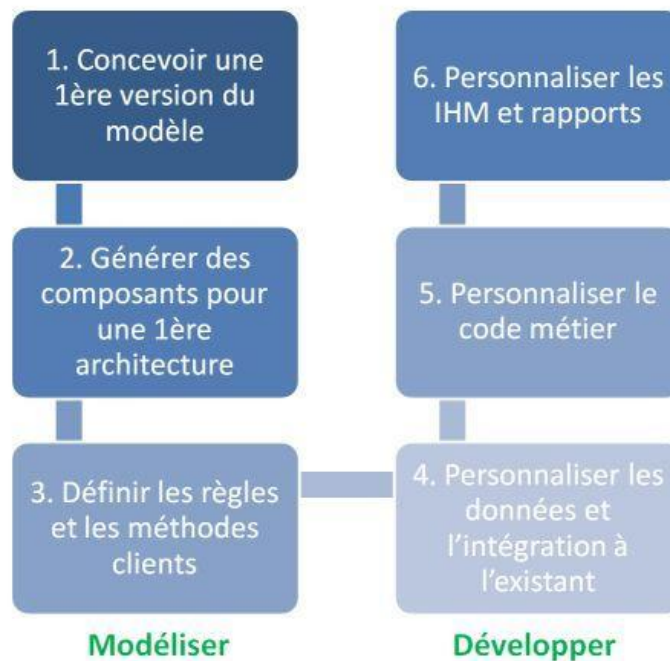
D. Etapes recommandées pour un usage basique

Dans ce chapitre, nous décrivons les étapes habituelles que nous recommandons pour livrer des applications du meilleur niveau en utilisant CodeFluent Entities.

Les trois premières étapes sont focalisées sur l'élaboration du modèle, permettant la génération de prototypes susceptibles d'être rapidement présentés aux utilisateurs pour validation.

La seconde série d'étapes est davantage focalisée sur le développement du code spécifique ; elle requiert plus de travail et est susceptible d'être plus dépendante de la plateforme cible.

Etapes d'une utilisation basique (itératif)



Note importante : comme expliqué dans le précédent paragraphe, CodeFluent Entities est particulièrement approprié pour un travail en mode agile, ce qui signifie que ces étapes n'ont nul besoin d'être séquentielles au niveau de la globalité du projet. Au contraire, ces étapes sont généralement séquentielles au sein d'une itération, avec notamment la première itération plutôt concentrée sur l'étape de modélisation.

1. Conception d'une première version du modèle

La première étape avec CodeFluent Entities consiste à concevoir une première version du modèle fonctionnel. Dans cette version initiale, **la majorité des entités et des propriétés pertinentes**, incluant les énumérations, sont identifiées.

Cette étape formalise les entités métier clés pertinentes pour votre domaine, leurs relations et les éléments structurels majeurs pour votre application. En utilisant un modèle classique de relations entre entités et héritage, on peut créer un modèle métier assez complet à l'aide uniquement de concepts relativement simples.

Même les données d'instances peuvent être définies dans le modèle, ce qui peut s'avérer utile dans la mesure où il est souvent nécessaire d'avoir des enregistrements spécifiques dans la base de données.

Ces éléments sont généralement très stables à l'échelle de la décennie.

2. Génération des composants vers une première architecture

Une fois les entités métier modélisées dans CodeFluent Entities, l'architecte choisit l'architecture appropriée à son besoin : web, smart client, Office, SharePoint, client/serveur traditionnel ou même Cloud.

Grâce à l'approche pilotée par les modèles CodeFluent Entities, l'architecte peut immédiatement générer une première version de l'application qui va implémenter le modèle fonctionnel.

Cette version peut être présentée aux utilisateurs pour validation des éléments fonctionnels du modèle.

De plus, CodeFluent Entities permettant une flexibilité au niveau des architectures cibles, il est également possible d'utiliser ces premiers prototypes pour orienter le(s) choix d'architecture.

Il n'est pas rare que le choix de l'architecture soit un vrai enjeu pour certains clients, notamment lorsque ces applications sont censées remplacer une version antérieure.

Faire du pur Web est la tendance naturelle mais, finalement, les conséquences ne sont pas vraiment mesurées, et en particulier les coûts pour atteindre la même ergonomie qu'une application classique sous Windows. Donc, partir sur une version Web sans avoir préparé les clients, peut se traduire par un échec au niveau du test d'acceptation des utilisateurs, ce qui est la dernière chose dont a besoin une équipe de développement.

En investiguant complètement les exigences de nos clients, nous trouvons généralement la meilleure combinaison :

- Une version pur web généralement proposée aux utilisateurs occasionnels avec le mérite de minimiser les défis de déploiement,
- Une application client intelligent pour les utilisateurs expérimentés, qui manipulent les données métier toute la journée et qui ont besoin de flexibilité et d'outils visuels,
- Une application mobile avec des caractéristiques spécifiques liées à des scénarios identifiés.

C'est pourquoi être capable de montrer des résultats rapidement avec la méthode pilotée par les modèles CodeFluent Entities procure une très grande valeur ajoutée aux dires de nos clients.

Note: De la même manière, il est possible de choisir un système de stockage à ce stade en conservant la possibilité de le modifier plus tard dans le cycle, par exemple pour déployer sur un autre système de gestion de base de données.

3. Définition de règles et méthodes personnalisées

Une fois que les éléments principaux ont été validés par les utilisateurs, il est en général fréquent dès la 2e itération de définir des règles métiers du modèle.

Cela inclut principalement les règles déclaratives. CodeFluent Entities propose un jeu élaboré d'options à décrire :

- Règles d'existence,
- Règles de formats de base et de types,
- Règles de formats plus complexes comme 'email' par exemple,
- Règles d'intégrité avec des options en cascade pour la mise à jour ou la suppression,
- Propriétés calculées.

A la fin de cette étape, le modèle métier CodeFluent Entities inclut déjà une part importante des notions fondamentales du métier de façon totalement indépendante de la technologie.

Dans la mesure où le format de stockage de ce modèle est un format XML simple, il est ouvert et durable quelles que soient les futures tendances technologiques.

4. Personnalisation des schémas de données et d'intégration

De par sa méthode de modélisation, CodeFluent Entities propose des mécanismes élaborés d'accès aux données :

- Un langage simple et intuitif : CodeFluent Query Language,
- Les vues personnalisées,
- La possibilité d'utiliser des procédures stockées natives dans le langage de la base de données.

Il est également possible de tirer partie des options suivantes :

- Un mappage de noms pour préserver les schémas de la base de données antérieures,
- Une connexion au travers de composants encapsulant l'accès aux systèmes existants.

5. Personnalisation du code métier

CodeFluent Entities contient bon nombre de concepts de modélisation qui simplifient le développement logiciel et permettent une grande flexibilité et puissance dans la définition de règles métier.

De par la philosophie du produit, il n'est pas possible (ou plutôt non pertinent économiquement) d'introduire des concepts de modélisation trop compliqués pour les utilisations les plus élaborées. C'est pourquoi CodeFluent Entities est conçu pour **permettre** une implémentation "classique" du **code spécifique** sur les différentes couches. En particulier, les applications CodeFluent Entities concentrent habituellement les comportements et les règles personnalisées dans le BOM, la couche centrale qui assure la cohérence.

Cette étape consiste donc à réaliser des **classes partielles** pour compléter le BOM généré de façon à garantir les exigences du besoin métier.

6. Personnalisation de l'interface utilisateur et des rapports

Bien que CodeFluent Entities fournisse les producteurs standards qui produisent directement des interfaces utilisateurs opérationnelles pouvant être utilisées pour manipuler des données, les applications métier exigent des écrans personnalisés ou des pages qui optimisent les processus.

CodeFluent Entities facilite cette conception grâce à la couche BOM évoquée précédemment car elle est facilement connectable aux contrôles visuels disponibles sur le marché. Cette approche fonctionne très bien avec les contrôles standards fournis dans la plateforme Microsoft tout comme ceux des acteurs classiques comme Infragistics, DevExpress, Telerik ou ComponentArt.

Concevoir une interface utilisateur pour les processus métier exige un maximum de flexibilité et c'est pourquoi les développeurs CodeFluent Entities ont l'habitude d'assembler les composants générés, tout en bénéficiant d'une série de services de runtime utiles (fournis par CodeFluent Entities), pour les différentes architectures supportées (Web, client-riche, Silverlight, SharePoint, Office).

Bien que des interfaces utilisateurs spécifiques soient généralement nécessaires, nous constatons dans beaucoup de projets, qu'il y a une proportion significative d'écrans qui requièrent une structure et un comportement standard. C'est pourquoi, dans les scénarios élaborés, nous allons décrire comment automatiser la création de modèles d'interface utilisateurs.

CodeFluent Entities intègre un moteur de "template" qui permet la production de tout fichier texte basé sur le principe du mix de "text output" avec exécution des instructions (comme les pages Active Service fonctionnaient avant .NET). Bien que constituant en réalité une relative petite partie de CodeFluent Entities, cet élément est comparable à de nombreux générateurs de code qui fournissent, aujourd'hui, uniquement ce mécanisme.

Le moteur de Template de CodeFluent Entities est même plus puissant à plusieurs égards car il perçoit le modèle objet de CodeFluent Entities comme une API, permettant **de tirer facilement parti de tous les éléments modélisés** par le développeur dans le concepteur graphique.

De plus, ce moteur comprend un **traitement natif du format RTF** permettant une stratégie de mise en oeuvre de rapport au format Word avec un haut niveau de performances et une mise à jour de ces modèles par les utilisateurs.

Le producteur de listes Microsoft Office fournit aussi une solution pour intégrer les interfaces utilisateur dans l'environnement classique Microsoft Excel ou Access.

E. Etapes recommandées pour une utilisation avancée

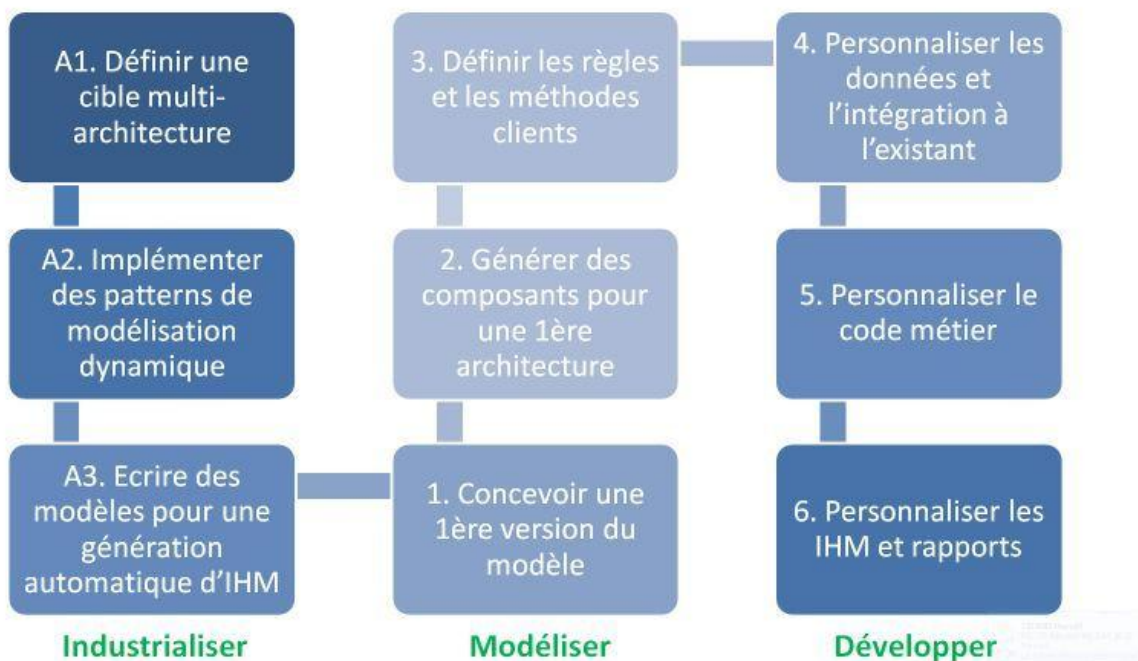
Bien qu'il soit facile de démarrer avec CodeFluent Entities, il inclut pourtant des caractéristiques avancées permettant de répondre aux projets de développement les plus exigeants. Véritable fabrique logicielle, CodeFluent Entities industrialise la production de code à un niveau jamais atteint.

Par conséquent, nous vous recommandons d'implémenter le 1er projet avec le scénario basique de sorte de vous familiariser avec l'approche pilotée par les modèles et le produit. Ensuite, sur un projet plus ambitieux, un énorme effet de levier peut être atteint lorsque l'on passe à un usage avancé du produit.

Les scénarios d'utilisation avancés incluent :

- Applications multi-architecture,
- Modélisation dynamique et patterns,
- Génération automatique d'interfaces utilisateurs.

Etapes d'une utilisation avancée (itératif)



11 / 11

1. Définir une cible multi-architecture

Aujourd'hui, les applications sont généralement conçues pour une architecture spécifique comme client-serveur, client riche, web ou web riche. Parfois, des extensions spécifiques supportent un scénario particulier comme la mobilité, souvent perçu comme une autre version de l'application.

Les applications couronnées de succès incluent souvent différentes architectures, du moins pour une partie. C'est tout-à-fait possible et à un coût raisonnable grâce à notre vision du développement logiciel et à notre approche pilotée par les modèles CodeFluent Entities.

Pour exemple, on peut imaginer pour un ERP Ressources Humaines que :

- Les utilisateurs d'un métier spécialisé vont disposer d'un **client intelligent** pour manipuler des données avec un maximum de flexibilité,
- Tous les utilisateurs dans l'entreprise accèderont à leurs données via un portail 'self-service' **pur web**,
- Certains scénarios d'accès aux informations personnelles seront directement intégrés sur **mobile** où les caractéristiques pertinentes seront déployées.

Ce type de scénarios sera de plus en plus fréquent ainsi que leur combinaison avec le Cloud... si l'on anticipe un peu le futur.

Utiliser CodeFluent Entities en **ajoutant ces scénarios de nouvelles architectures** est juste une question de :

- Ajouter de **nouveaux producteurs** pour générer des composants pertinents,
- **Compléter** simplement le code concerné, particulièrement le **code spécifique à l'IHM**.

2. Implémenter une modélisation dynamique avec des patterns

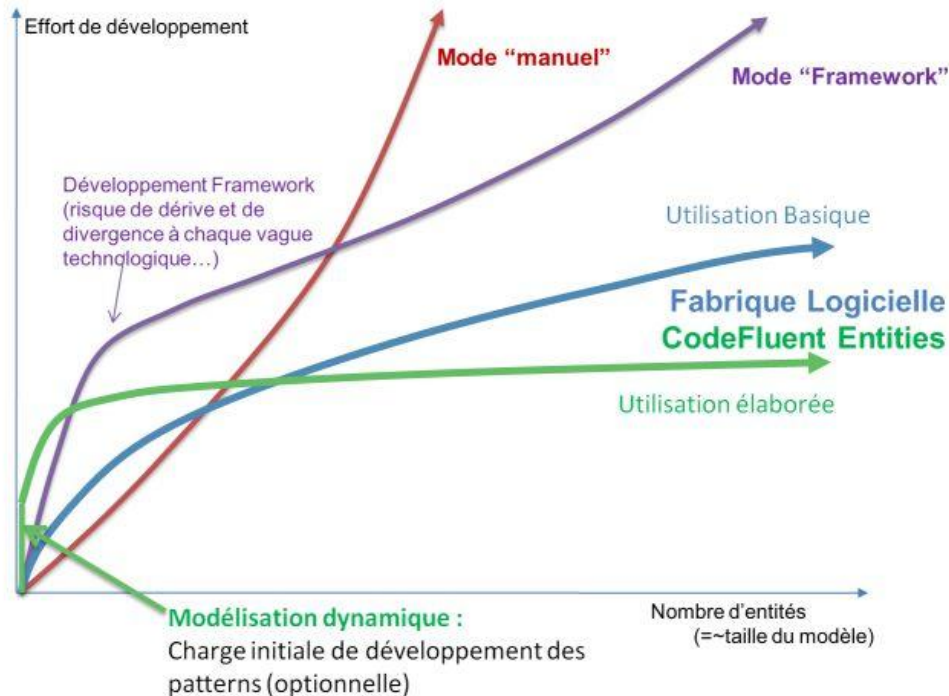
Comme expliqué dans le chapitre 3, se placer au bon niveau d'abstraction nécessite de trouver le juste équilibre. La bonne nouvelle avec CodeFluent Entities, c'est que vous pouvez **commencer très simplement** tout en bénéficiant de concepts puissants à une étape ultérieure.

Le produit étant **complètement conçu comme une API**, donnant aux développeurs un maximum de flexibilité, l'approche basée sur les templates décrite ci-dessus peut être appliquée au modèle lui-même ce qui produit un effet de levier puissant : la **modélisation dynamique**.

En appliquant les principes des templates au modèle lui-même, CodeFluent Entities vous permet de définir des **patterns**, une manière industrielle d'ajouter des comportements à vos entités. Cela inclut des comportements tels que : localisation, historisation, recherche de texte ou n'importe quel besoin client.

Cependant des générateurs basés simplement sur les templates ne peuvent pas combiner ces comportements sans apporter à ces templates un tel niveau de complexité qu'il n'est plus possible de les maintenir (bien connu sous le nom d'effet spaghetti dans les pages Active Server). L'approche différente proposée par CodeFluent Entities rend la production du BOM industrielle comme représenté ci-dessous.

Production logicielle industrialisée



L'utilisation basique du produit procure déjà une solution viable économiquement et durablement (ce qui n'est pas le cas des modes 'manuel' et 'framework' que nous avons observés sur le terrain) et l'utilisation avancée avec une modélisation dynamique procure un ROI encore supérieur pour les projets comprenant des modèles de centaines d'entités.

3. Concevoir des modèles d'interface utilisateur pour les générer automatiquement

Dans beaucoup de projets nous constatons que bien que des interfaces utilisateurs spécifiques soient généralement requises, une proportion significative d'écrans exige une structure et un comportement standard. C'est notamment le cas pour les interfaces utilisateurs back office ou pour la gestion de dates de référence.

Dans ce cas, il est pertinent d'investir dans un **template d'interface utilisateur** qui va décomposer la structure du UI et les comportements communs. Ce template servira de base pour la génération d'interfaces utilisateur du type de celle livrée avec le produit.

De cette manière **les coûts de développement et de maintenance ne seront pas corrélés au nombre d'entités**. Pour les projets de centaines d'entités, cela peut engendrer des économies substantielles au-delà du cycle complet de vie du produit.

Grâce à la grande flexibilité de CodeFluent Entities, cette approche peut également être fournie pour toutes les interfaces utilisateurs qui utilisent des caractéristiques élaborées, rendu client et code client dans les templates. Cependant, pour les écrans compliqués, cela peut être plus difficile à maintenir, c'est pourquoi nous proposons les 2 approches, l'architecte étant capable **d'équilibrer** les 2 au niveau optimal pour le projet.

VII. Bénéfices de CodeFluent Entities: 12 exemples de scénarios

Maintenant que vous avez compris pourquoi nous avons conçu CodeFluent Entities et comment vous pouvez livrer des applications avec CodeFluent Entities, vous trouverez ci-après quelques exemples classiques de scénarios qui illustrent comment vous pouvez profiter des apports de CodeFluent Entities.

| Scénario | Sans CodeFluent Entities | Avec CodeFluent Entities | Mots clés |
|--|--|---|--|
| Le chef de produit ou les utilisateurs ont oublié une propriété importante dans l'entité. | Il vous faut changer le schéma de la base de données, les procédures stockées, modifier les couches business, services et interface utilisateurs. | Vous ajoutez simplement la propriété au modèle et vous re-générez. Si votre interface utilisateur est spécifique, vous ajoutez simplement la propriété. | Pilotage par les modèles Cohérence Couches |
| Votre application est réussie. Mais vous avez besoin de supporter 10 000 utilisateurs courants avec 1 milliard d'enregistrements. | Il vous faut implémenter des mécanismes de pagination et de tri sur les écrans pertinents et trouver comment appliquer l'approche sur toutes les couches. | En tirant partie du mécanisme de pagination et de tri inclus dans CodeFluent Entities, votre travail se résume à leur configuration sur les parties pertinentes de l'application. | Extensibilité Meilleures pratiques Cohérence |
| Le département juridique vous demande de tracer toutes les modifications de données liées à l'application. | Vous aurez besoin d'ajouter un processus spécifique à différents endroits dans un schéma de base de données évolutif pour stocker les données pertinentes. | Vous avez juste à modéliser les données à historiser et à implémenter un pattern qui sera ajoutée dans la couche métier pour tous les composants une fois pour toutes. | Patterns Dynamique Modélisation Cohérence |
| Les utilisateurs souhaitent entrer des données en mode liste directement dans Excel. | Vous devrez développer un processus d'import spécifique intégrable dans Microsoft Excel. | En utilisant le producteur Excel, vous pouvez générer immédiatement des listes modifiables incluant la vérification des règles métier. | Office Producteur Ergonomie |
| Tous vos objets doivent désormais respecter certains standards tels que : les noms de propriété doivent être précédés du nom de l'entité. | Vous devrez renommer tous les éléments manuellement ou élaborer un outil qui automatisera le processus et le tester. Il est probable que ce travail ne soit pas juste à 100 %. | Vous pouvez utiliser des conventions de nommage proposées par CodeFluent Entities et vous obtiendrez 100% de conformité par conception. | Cohérence Convention Qualité |
| Votre application s'exporte, vous devez supporter 10 | Vous devrez vérifier le respect des règles d'internationalisation et | En utilisant les caractères Unicode, les messages de ressources et le pattern de | Localisation Meilleures pratiques |

| | | | |
|---|--|---|--|
| langues. | trouver le moyen de localiser les données dans la base. | localisation CodeFluent Entities, vous aurez la solution pour les exigences d'internationalisation. | Patterns |
| Faisant partie du partenariat Microsoft, vous devez supporter Silverlight. | Vous devrez reconfigurer votre application pour inclure cette nouvelle architecture intégrant des aspects complexes comme la nature asynchrone du modèle de programmation de Silverlight. | En utilisant le SCOM de CodeFluent Entities, votre travail sera limité à la configuration d'un nouveau producteur et l'ajustement des éléments d'IHM. | Architecture Silverlight Innovation |
| L'application doit maintenant être déployée sur une autre base de données. | Vous devrez comprendre quelle partie du code d'accès aux données est spécifique à votre système de base de données et reconfigurer ces éléments et potentiellement, trouver un mécanisme qui supporte les 2 (si besoin). | CodeFluent Entities a été conçu de façon à permettre différents scénarios de déploiement de bases de données tout en optimisant le système de code de chaque SGBDR. Seules les méthodes natives codées pour optimiser un processus spécifique ont besoin d'être revues. | Pilotage par les modèles Architecture Oracle/SQL |
| Les utilisateurs souhaitent envoyer des courriers aux clients en exploitant des données extraites de l'application. | Vous devrez automatiser Word ou intégrer un composant spécifique pour livrer cette caractéristique. | En utilisant le producteur de modèles avec les templates RTF, vous disposez immédiatement d'une solution permettant aux utilisateurs d'être capables d'éditer des modèles de lettres dans Word. | Producteur RTF Templates |
| Une nouvelle politique change les règles de contrôle d'une propriété. | Vous devrez changer cette règle partout où elle s'applique voire même sur le client et le serveur. | Vous ajoutez cette contrainte dans le modèle et vous régénérez. Les règles peuvent être contrôlées à la fois sur client et serveur pour une meilleure expérience utilisateur. | Pilotage par les modèles Règles Architecture |
| Votre dernière application est tellement couronnée de succès que l'on vous demande de re-développer l'application vieille de 20 ans avec les mêmes standards | Vous devrez repartir de zéro bien que le modèle ait été conçu il y a plus de 10 ans. C'est un travail colossal. | En utilisant l'importeur de CodeFluent Entities sur une base de données utilisée pour une application existante, vous disposez immédiatement d'un premier modèle pour commencer à générer rapidement des écrans plus modernes pour votre application. Vous pouvez | Legacy Importeur Modernisation |

| | | | |
|--|---|---|-------------------------------------|
| d'ergonomie . | | reconfigurer progressivement l'application en améliorant le modèle fonctionnel et en travaillant sur une nouvelle architecture cible. | |
| Les investisseurs ont exigé de votre application qu'elle supporte une architecture Cloud dans les toutes prochaines semaines. | Vous devrez trouver les différents composants d'une architecture Cloud disponible et peut être écrire une version totalement différente de votre application. | En utilisant CodeFluent Entities, vous serez capable de générer le modèle vers des nouvelles plateformes incluant le Cloud sans écrire une version différente de l'application. | Innovation Cloud Architecture |

VIII. Conclusion

L'innovation technologique va vite, et le rythme s'accélère alors que l'on doit faire face à la mondialisation et à une concurrence féroce. Chaque jour qui passe, une ou plusieurs nouvelles technologies sont annoncées et cette tendance n'a pas l'air de ralentir. Le développement de logiciels en couches, s'il a augmenté la flexibilité et l'ouverture, a également fortement accru la complexité du développement logiciel.

Avec la multiplication des plates-formes en ligne, l'avènement du "cloud" et des appareils mobiles, ainsi que les besoins croissants de se connecter à des réseaux sociaux, nous sommes convaincus que cette complexité va continuer de croître à l'avenir.

Parallèlement, depuis plusieurs décennies, les entreprises ont développé des systèmes d'information opérationnels conséquents, pour aller plus loin et plus en profondeur dans les fonctionnalités afin de se maintenir sur le marché. L'héritage applicatif est bien plus important qu'il y a quelques années et continue d'augmenter. Une récente [étude du Gartner](#) a montré que ce que les analystes appellent désormais la 'dette technologique' croît à un rythme inquiétant.

Dans toutes les entreprises, la pression pour réduire les coûts, tout en offrant de nouveaux services afin de rester compétitif dans un environnement plus complexe, est forte ; s'ajoute à cela la nécessité de gérer un héritage croissant, évoqué ci-dessus, avec son lot de contraintes liées à l'obsolescence. Certains systèmes vont même jusqu'à continuer à fonctionner sans être supportés avec le risque d'une refonte contrainte et forcée à un moment ou à un autre.

Pour les entreprises qui ont besoin de maintenir et d'améliorer leur système d'information obsolète pour poursuivre leur activité, c'est la quadrature du cercle. Ce n'est pas une solution viable de se contenter de redévelopper les applications à chaque nouvelle vague technologique.

En fait, à de très rares exceptions près, aucune entreprise peut se permettre un tel coût. Et même si vous étiez très riche, il serait sans doute plus judicieux de dépenser votre argent pour augmenter votre chiffre d'affaires.

En même temps, le maintien en place des applications très anciennes tend à être la réalité du terrain, mais on peut légitimement se demander combien de temps cela peut encore durer sans une crise majeure. Une telle stratégie 'statu quo', qui est évidemment moins risquée à court terme, est probablement le plus sûr chemin vers la mort à long terme. Dès l'instant où votre concurrent aura effectué le virage, il sera déjà trop tard pour entamer une refonte de vos systèmes.

C'est la raison pour laquelle nous pensons qu'il y a 2 phases critiques sur lesquelles toute entreprise devrait travailler de manière proactive :

- A. Fixer des objectifs mesurables de réduction de la maintenance de l'héritage des systèmes existants,
- B. Utiliser les économies pour réaliser les nouveaux projets selon une approche aussi indépendante que possible de la technologie.

Pour le point A, il est essentiel d'évaluer précisément les coûts de maintenance, en termes de matériel, licences de logiciels et jours/homme pour faire fonctionner ses systèmes et corriger les bugs. Pour les applications personnalisées qui sont en mode de maintenance, le coût est fortement corrélé au nombre de lignes de code et à la qualité de l'application. Moderniser des applications existantes sans aucune modification fonctionnelle peut être source d'économies considérables et n'est pas une tâche difficile en soi, à condition de ne pas tomber dans le piège de l'évolution des fonctionnalités en parallèle.

Pour le point B, le développement piloté par les modèles vous permet de pérenniser votre travail en le plaçant à un niveau d'abstraction plus élevé que le code. Nous et nos clients réalisons des projets sur cette base depuis plus de 6 ans, et cette capacité d'évolution devient évidente dès l'instant où vous l'avez expérimentée.

En outre, elle présente l'avantage considérable de permettre une prévision fiable des coûts et des délais. Après une utilisation de la méthodologie, vous êtes à même de faire une estimation précise de chaque projet en utilisant des éléments factuels tels que les nombres d'entités, de propriétés, de méthodes, de règles métier, d'interfaces utilisateur ou de rapports. Nous publierons des chiffres détaillés et la méthodologie d'estimation associée dans une prochaine version de ce livre blanc, ce qui vous permettra de comparer votre projet en cours avec nos estimations.

Nous vous invitons donc à suivre notre actualité au travers de nos différents canaux de communication.

Daniel COHEN-ZARDI
Président de SoftFluent
<http://blog.softfluent.com>

IX. Annexes

Voici une liste des informations clés pour tout savoir, à tout moment, sur CodeFluent Entities et les meilleures pratiques de développement .NET vues par SoftFluent :

| Url | Information |
|---|---|
| http://blog.softfluent.com | Blog dédié au partage de notre expérience et des meilleures pratiques sur tous les aspects de développement logiciel |
| http://blog.codefluententities.com | Blog apportant un éclairage régulier sur les fonctionnalités de CodeFluent Entities. |
| http://www.codefluententities.com/rss.aspx | Liste des nouvelles fonctionnalités et des corrections de bogues pour chacune des nouvelles versions. |
| http://forums.softfluent.com | Portail d'accès aux forums SoftFluent. |
| http://www.youtube.com/softfluent | SoftFluent TV : chaîne qui diffuse régulièrement des vidéos de démonstrations du produit. |
| http://www.codefluententities.com/documentation | Documentation complète du produit. |
| http://www.softfluent.com | Site de la société SoftFluent. |
| http://www.codefluententities.com | Site du produit CodeFluent Entities. Il vous permet de télécharger gratuitement la licence CodeFluent Entities dont une version illimitée en fonctionnalités destinées aux usages personnels. |
| http://store.softfluent.com | Boutique en ligne de SoftFluent. |