

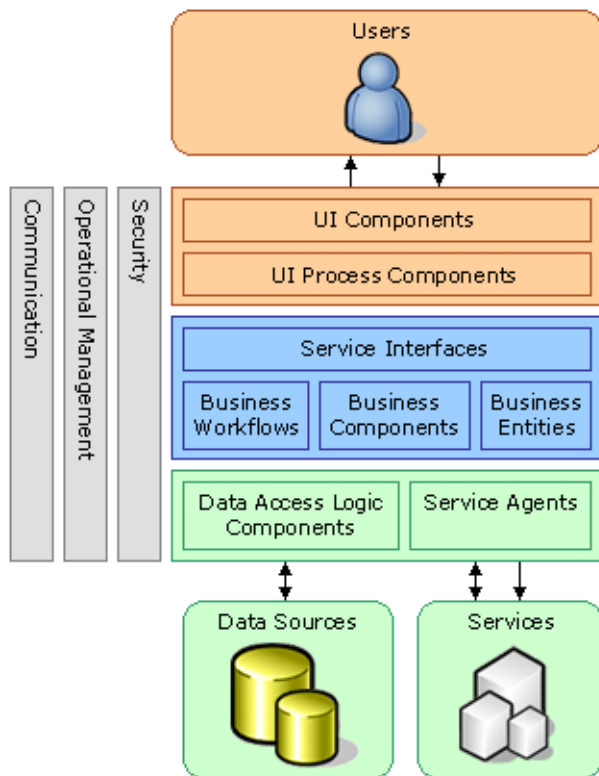


**Pourquoi la solution Aspectize vous permet-elle de réduire les couts et maitriser les budgets de vos projets .Net ?**

## Qu'est-ce qu'une bonne architecture ?

C'est une architecture en plusieurs couches indépendantes.

La nécessité de bâtir des architectures en couches n'est plus à démontrer. Plus qu'un standard, la séparation des 3 couches bien connues que sont **Présentation**, **Traitements** et **Données** est aujourd'hui une évidence pour qui veut bâtir un Système d'Information pérenne et maintenable. Toute la littérature sur le sujet converge donc vers cette bonne pratique, et Microsoft ne fait pas défaut en illustrant ainsi les *Best Practise* d'architecture :



On y trouve tous les éléments classiques mis en œuvre par les architectes dans les projets. Leur principale difficulté va être de reproduire ce schéma-là dans tous les contextes métiers des projets. Pourquoi ? Parce que la mise en œuvre de cette architecture n'est pas indépendante du métier. Les *UI Components* et les *Business Entities* vont fortement lier le métier et l'implémentation.

Ce couplage est doublement dommageable pour les Systèmes d'Information. Premièrement, si l'architecture était vraiment indépendante du métier, elle serait faite une bonne fois pour toutes, et nul n'aurait besoin de la refaire dans chaque projet. Deuxièmement, puisqu'elle n'est pas indépendante, elle nécessite de connaître d'avance beaucoup trop de détails métier pour la mettre en œuvre correctement. Des choix structurants sont fait trop tôt, les projets sont livrés trop tard, et il suffit que le besoin utilisateur évolue un tout petit peu pour que l'impact vis-à-vis de l'existant se paie au prix fort. Les projets sont souvent en retard et plus cher que prévu parce que le besoin évolue, alors qu'il a déjà impacté la réalisation.

Arrêtons-nous un instant sur les raisons qui font que **l'architecture est difficilement indépendante du métier** avec les approches actuelles.

La case *Business Entities*, qui porte bien son nom, indique l'endroit du modèle objet de l'Application. Par Modèle Objet, on entend un typage fort du contexte métier.

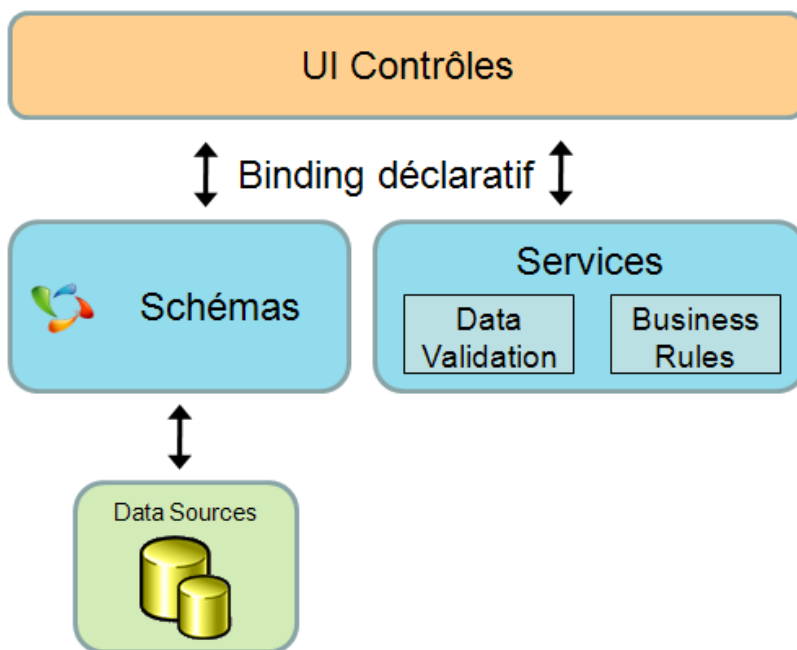
Si l'application gère une librairie, il y a de fortes chances pour qu'on y trouve un objet *Livre* et un objet *Auteur*. Le design de ces objets doit se faire très tôt, car c'est toute la logique métier qui est présente dans ce design. Il est quasiment impossible à un développeur d'imaginer sa couche logique sans des objets métiers spécifiques. Là où les difficultés arrivent, c'est que cette approche vieillit très mal car elle est rigide. En effet, il y a de fortes chances pour que l'objet *Livre* contienne une référence vers l'objet *Auteur*, et que l'objet *Auteur* contienne une liste de *Livres*. Ce lien étant représenté par du code, il va être exploité partout où l'Application en aura besoin. Si on imagine que la couche IHM veut tirer profit de ce lien entre *Auteur* et *Livre* (représenter la liste des *Livres* écrits par un *Auteur*), le lien va être présent dans les contrôles de l'IHM. Le typage fort du modèle métier est donc intrinsèquement lié aux contrôles de la couche de présentation.

Imaginons un instant que le modèle évolue et qu'un *Livre* puisse avoir plusieurs *Auteurs*, les impacts seront nombreux, non seulement au niveau du stockage, mais également dans l'implémentation de la couche de présentation, alors qu'il suffirait logiquement de changer un label en une liste. Malgré toutes les précautions prises pour séparer les briques, l'indépendance des couches est rompue, le couplage est fort car les liens sont régis par du code et sont donc impératifs. **C'est le principal défaut de l'Approche Objet.**

## Aspectize propose une nouvelle approche

Fondée sur une approche descriptive, elle réussit à séparer totalement le métier et la technique. Là où les liens sont traditionnellement impératifs, Aspectize propose une **implémentation descriptive**, qui permet rapidité, souplesse et totale indépendance entre les couches.

Respectant rigoureusement les paradigmes d'une architecture n-tiers, elle reprend logiquement les différentes couches d'une Application :



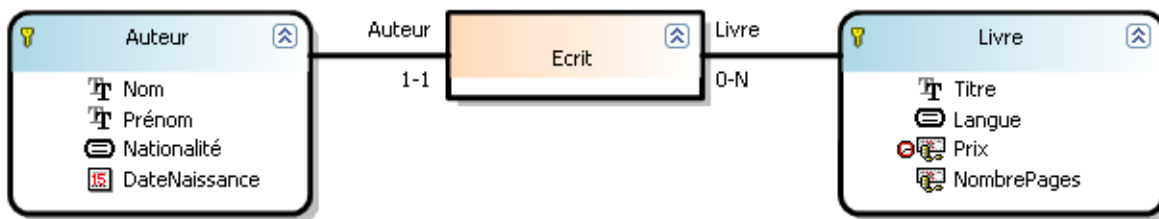
Regardons d'un peu plus près comment les différentes briques sont constituées et articulées les unes par

rapport aux autres. Plusieurs éléments sont fondateurs dans la mise en œuvre de l'approche.

## Le schéma relationnel, contexte métier central de l'application

Le **Schéma** est le lieu de définition du **modèle logique métier**. En représentant un modèle Entité/Relation, il permet de centraliser la définition de toutes les données manipulées par l'application.

La relation est au cœur du Schéma : si nous reprenons notre exemple précédent, nous avons vu que l'approche classique du modèle objet tendait à lier l'objet *Livre* et l'objet *Auteur*, en implémentant cette relation sur les objets eux-mêmes, avec des références croisées. L'approche d'Aspectize est de ne jamais lier deux entités directement, mais de considérer la relation comme étant un lien logique à part entière entre les différentes entités.



Nous avons une entité *Livre*, une entité *Auteur* et une relation *Ecrit* entre les 2 qui renseigne le fait qu'un *Auteur* *Ecrit* un *Livre*. Cette différence est fondamentale, puisqu'elle permet aux entités d'être réellement et constamment indépendantes entre elles et de **considérer les relations comme des éléments du modèle à part entière**. La structure de l'entité *Auteur* n'est pas impactée par l'entité *Livre*. Le Schéma peut évoluer beaucoup plus facilement : on peut imaginer qu'un *Auteur* écrive plusieurs *Livres*, et donc qu'un *Auteur* sera en relation avec plusieurs *Livres*. On peut imaginer une évolution avec une relation supplémentaire entre *Auteur* et *Livre*, telle que *Recommande*. La compatibilité du modèle est assurée car cette évolution n'impacte aucune modification des Entités déjà définies. Cette nouvelle relation sera chargée, ou peut-être pas, dans les différents contextes d'usage de l'entité Métier. L'indépendance des Entités permet de prendre cette décision de façon contextuelle ce qui contribue à la souplesse du modèle.



Précisons que ce modèle représente le contexte métier de l'Application, et non pas un modèle physique de stockage. Certes, on pourra stocker les méta-données physiques dans ce modèle, s'il s'agit d'une base relationnelle, mais le **Schéma va bien au-delà et permet de stocker beaucoup plus**

d'informations complémentaires, utilisées dans l'Application :

- La **validation** des données, qui permet de définir les règles à appeler dans le cas d'une validation de données liée à un Contrôle ou à l'appel d'une Commande.
- Les **événements logiques** liés à la création, modification ou suppression d'entités, qui permettent de configurer les appels de Commandes sur ces événements.
- Le caractère **Temporel** d'une propriété, qui permet d'ajouter une dépendance avec le temps, de façon déclarative.
- Des **expressions calculées** qui permettent de définir des agrégations et des propriétés calculées de façon déclarative.

L'implémentation de ce modèle, n'est pas un modèle objet, mais est réalisée avec le DataSet standard de .Net, conteneur idéal qui permet de stocker tout type de données relationnelles. Bien qu'il soit loin de faire l'unanimité au sein de la communauté des développeurs - sa manipulation est assez verbeuse, il n'y a pas d'intellisense, cela reste un objet technique qui nécessite une bonne connaissance pour être utilisé correctement - le DataSet a l'avantage d'être un conteneur générique de données bindable, serialisable, triable, filtrable, clonable, avec une gestion de l'état des données et une logique événementielle extrêmement puissante. Son usage dans l'infrastructure Aspectize est totalement transparent, et les principaux aspects négatifs ont été gommés ; **l'IntelliSense est disponible**, et la **navigation dans les données est facile et intuitive**. **La souplesse et le dynamisme du DataSet en font un conteneur de données métier idéal**, bien plus efficace qu'un modèle objet spécifique.

Cela permet de garder une approche générique pour beaucoup de fonctionnalités, et notamment le DataBinding.

## DataBinding Relationnel et validation de données

Le DataBinding est une bonne pratique qui permet de s'affranchir d'écrire du code pour faire interagir, de façon bi-directionnelle, les données et les contrôles de l'IHM. Cette opération est réalisée de façon déclarative, et **beaucoup de code fastidieux n'est plus à écrire**.

Pourtant, la technique de DataBinding proposée par Microsoft a quelques limites :

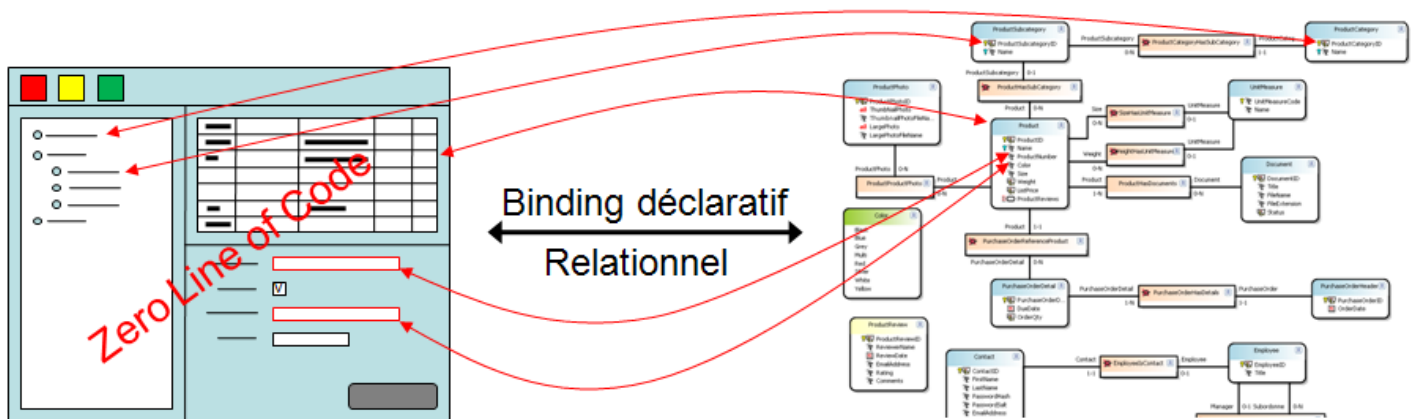
- elle fonctionne bien pour une grille ou un contrôle simple, mais elle ne fonctionne pas pour de nombreux autres contrôles, comme par exemple l'arbre.
- elle nécessite d'écrire du code technique, que finalement peu de développeurs maîtrisent.
- elle concerne essentiellement une Table simple, et ne permet pas de naviguer facilement dans les données relationnelles.

Avec Aspectize, le **DataBinding est considérablement enrichi** :

- il est étendu à tous les contrôles standards du Framework .Net 2.0, y compris les arbres et les TabControl, ainsi que les bibliothèques tiers de Contrôles.
- il est étendu au **Web** et aux applications **Ajax**, ce qui est une innovation unique (aucune implémentation de DataBinding en JavaScript n'existe par ailleurs).

- il prend en compte l'aspect **relationnel du modèle**, et permet de tirer parti des relations entre entités, pour réaliser une navigation relationnelle sans écrire de code.
- il permet d'associer **automatiquement une validation de données**, et permet d'exploiter une règle métier sans écrire du code.
- il est complètement déclaratif, se gère via une interface graphique et intuitive, sans écrire la moindre ligne de code.

Le **DataBinding Relationnel** d'Aspectize permet ainsi de gérer de façon déclarative, l'intégralité des données dans l'IHM. En quelques clics de souris, et sans la moindre ligne de code, il est possible d'associer toute donnée du Schéma à un contrôle de l'Application.



Il suffit de choisir graphiquement l'Entité à afficher dans un contrôle, en tenant compte du **chemin de navigation relationnelle des données**. Ainsi, la structure d'un arbre va logiquement suivre la structure relationnelle des données, qui correspond le plus souvent à la logique d'affichage des données. Que l'interface soit Windows ou Web, la logique est exactement la même ; un contrôle affiche des données relationnelles, par une association logique entre ses propriétés et les Entités du Schéma.

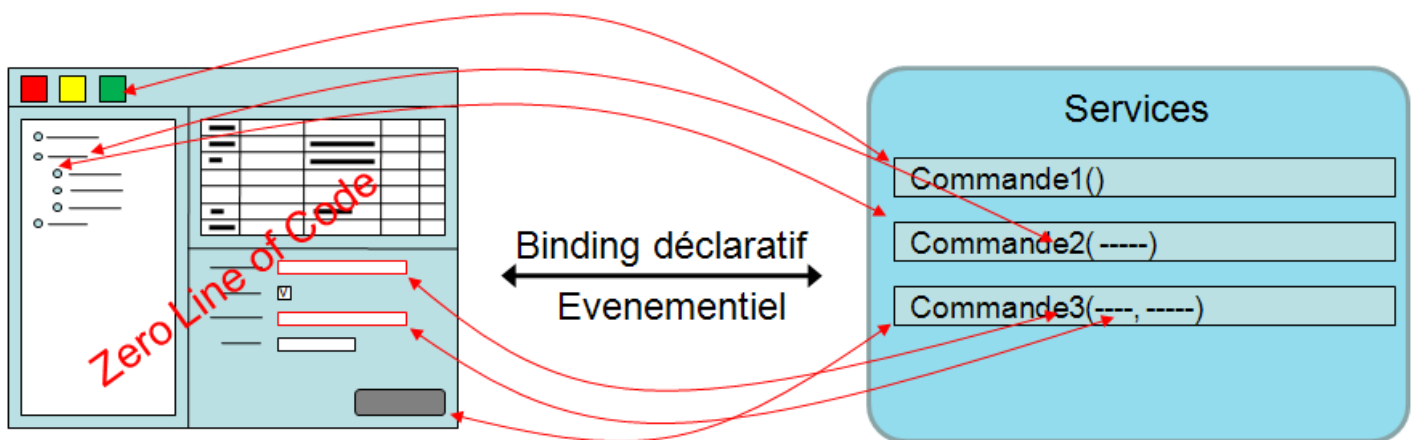
La **validation des données**, qui reste une bonne raison pour écrire du code métier spécifique, est complètement intégrée au DataBinding. Chaque définition d'un DataBinding entre un contrôle et une donnée peut faire l'objet d'un contrôle de validation. Que la validation soit simple (non-nullité, limite supérieure ou inférieure, expression régulière), ou qu'elle fasse appel à une règle métier, écrite en .Net standard et de façon complètement **indépendante de l'IHM**, elle est **complètement déclarative et ne nécessite aucun code**. Il n'y a aucun code d'appel à écrire, aucun événement de contrôle à gérer, le câblage avec l'IHM a déjà été fait une fois pour toutes, ce qui constitue un gain de temps considérable et un gain de qualité évident, car quantités de sources de bugs sont ainsi évitées.

## Architecture SOA et Command Binding

Le serveur d'Applications Aspectize est nativement et naturellement une plate-forme SOA. Le code métier écrit est automatiquement accessible comme un Service de l'Application, qui est l'élément fédérateur qui répond à une problématique métier. Il n'y a rien de particulier à écrire comme code ou à configurer pour créer un nouveau Service. **Il suffit d'écrire le code métier en .Net standard**, et le service sera rendu disponible par le serveur d'Applications. Et cela fonctionne, parce que le contexte métier n'est pas implémenté avec un modèle objet rigide.

La grande innovation du Serveur d'Applications Aspectize réside dans l'**interception des appels** : tous les appels à un service seront interceptés par le Serveur d'Applications. Le service métier bénéficie ainsi de tous les services techniques transverses implémentés et disponibles : **conversion de données, gestion d'erreurs, sécurité, trace, log, bouchons, Load Balancing, Fail Over**, sont automatiquement disponibles. Il n'est plus nécessaire d'écrire de code pour implémenter ces fonctionnalités, il suffit de configurer le service pour qu'il en bénéficie. Activer la sécurité d'une Application se fait en cochant une case, et de façon complètement indépendante du code métier. Un autre exemple emblématique est que **toutes les exceptions sont automatiquement attrapées et loguées** : il n'est plus possible d'avoir une exception non attrapée par le système. La robustesse des Applications est naturellement accrue sans efforts particuliers.

Configurer les intercepteurs est beaucoup plus simple et rapide que d'écrire du code et cela peut se faire tardivement, sans remettre en cause le code existant. Une Application non distribuée peut ainsi le devenir, même si cela n'a pas été prévu au départ. La mise en place d'une trace ou d'un bouchon peut se faire sans toucher à l'implémentation. L'Architecture applicative se décide au fur et à mesure des développements métiers, et n'a plus aucun impact sur la démarche projet. Ce n'est plus un pré-requis indispensable avant de commencer les développements. On peut ainsi **développer des briques indépendantes et les assembler au fur et à mesure** des besoins métiers. L'agilité est enfin possible.



De la même manière que le DataBinding Relationnel permet l'association déclarative d'un contrôle et de données, le CommandBinding permet l'association déclarative d'un événement IHM et d'un appel de service. Cette association qui donne lieu traditionnellement à l'écriture de code, et qui lie fortement les contrôles de l'IHM aux composants métiers, se fait maintenant de façon déclarative. Au lieu de faire un appel explicite et impératif dans l'événement OnClick d'un bouton, l'association entre le bouton et la Commande est fait tardivement et par simple configuration. Cela permet de disposer d'une IHM totalement indépendante des autres composants, ce qui n'était pas possible auparavant. La qualité des applications s'en trouve grandement améliorée, et la réutilisation des différents composants devient possible. Cette configuration est réalisée avec **Binding Studio, qui permet de sélectionner de façon graphique et intuitive les éléments à associer**. En quelques minutes, l'IHM est prête pour interagir avec les services métier du Serveur. Le Serveur d'Applications s'occupe de toutes les conversions de données à l'aller et au retour.

Avec l'infrastructure Aspectize, il n'y a plus de raisons pour écrire du code dans les IHM. On atteint le "Zero Line of Code" : c'est beaucoup de temps gagné et beaucoup de sources de bugs qui sont ainsi évitées.

## Vers un Système d'Information piloté par le métier

Baisse des risques, gain de temps, souplesse accrue du système, les applications gagnent en qualité. Il est beaucoup plus facile de construire des applications robustes. La plupart des **problèmes techniques sont éliminés, et les développements sont concentrés sur le métier**, dès le premier jour. Plus besoin d'investir massivement dans une conception d'architecture, qui aura beaucoup de mal à évoluer avec le temps pour concilier montée en charge et évolution des besoins.

Les traitements de la couche logique métier sont écrits en .Net tout à fait standard. Aucune intrusivité, pas d'usage d'API, le code standard permet d'exprimer le métier, avec des services indépendants, dans l'esprit des architectures SOA. Plus facile à écrire et donc à lire, le code métier est débarrassé de toute pollution technique : le lien avec le fonctionnel est beaucoup plus fort. La détection des erreurs est bien plus rapide, et les gains en maintenance sont immédiats. Le système peut enfin être **piloté par le métier**.

Les aspects techniques n'étant plus gérés par du code lié au développement métier, vous avez la garantie que cette gestion sera faite de la même façon dans toutes vos applications. Cela unifie l'ensemble de vos développements dans une implémentation conforme aux règles de l'art. Le temps où l'uniformité était liée à la responsabilité de chaque développeur est terminé.

Là où les Framework traditionnels vous invitent à apprendre leurs API et à écrire du code, l'infrastructure Aspectize permet de disposer d'une **architecture clé en main**, par simple configuration, sans écrire une seule ligne de code. C'est une innovation majeure, en particulier pour les **Applications Web full-Ajax**, pour lesquelles il n'existe aucun environnement pour les réaliser sans écrire de code. La courbe d'apprentissage est rapide, il n'y a **pas d'API à apprendre**, vous travaillez avec votre environnement favori qui est Visual Studio et votre langage standard .Net. Il n'y a aucune fermeture : vous pouvez ajouter du code spécifique à tout endroit, l'infrastructure est totalement interopérable avec tout type d'environnement applicatif, ou composant tiers.

Les applications sont nativement capables de **monter en charge**, elles sont **sécurisées, déployables** en quelques opérations simples, **dès le premier jour**. L'approche permet de concilier l'agilité des Approches RAD avec la robustesse d'une infrastructure logicielle professionnelle.

Si les coûts de développement sont très nettement diminués, les **coûts de maintenance** sont encore plus fortement impactés. D'une part, il est beaucoup plus facile de faire intervenir des ressources différentes et nouvelles sur les projets, le coût de maintenance de la connaissance d'un projet est quasiment nul. D'autre part, l'impact d'un ajout ou d'une modification sur l'existant est très faible, voir inexistant. Le gain généré est permanent **tout au long du cycle de vie de l'application**, depuis son premier jour jusqu'à sa fin de vie.

Et demain, lorsqu'il s'agira de porter le SI sur le Cloud, cela pourra se faire sans le moindre changement dans l'implémentation. La plate-forme est portée sur Azure, la migration d'une Application d'un monde OnPremise à Azure est quasiment gratuit.

## Les Garanties de l'approche Aspectize

L'approche Aspectize répond à toutes les problématiques d'Application Business, pour gérer des données relationnelles. Quel que soit le domaine, l'approche Aspectize **garantit un résultat visible et opérationnel dès le premier jour du projet**.

Si la baisse des charges de développement peut légitimement être divisée par 3 (estimation réalisée pour une application Web standard de gestion, type CRM), la **baisse des coûts de maintenance** est encore plus

significative. Avec un **impact proche de zero**, vu qu'il n'y a pratiquement pas de code existant à impacter, l'ajout de fonctionnalité ou la modification d'éléments existants se fait sans aucun effort supplémentaire, lié à l'existant. Là où les approches traditionnelles rendent extrêmement couteux les modifications, du fait de la non-régression de l'existant, qui sera nécessairement impacté parce qu'il est implémenté par du code impératif.

L'**intégration** de la solution dans la suite **Visual Studio** permet l'acquisition et la **maitrise de la solution** extrêmement **rapidement**. Peu d'API à apprendre, peu d'éléments techniques à connaître, le niveau d'**expertise technique** requis pour construire un système d'information professionnel est logiquement **tiré vers le bas**. Il devient plus facile d'échanger les ressources, la montée en connaissance d'une application et la reprise d'existant se fait rapidement ; l'implémentation nécessite peu de documentation, les éléments constituant l'application (Schéma relationnel, DataBinding et CommandBinding) sont **auto-documentés**.

Enfin, et cela constitue peut être l'élément le plus novateur, l'approche Aspectize permet une **maîtrise totale du budget** d'une application. Quel que soit le domaine, le fait de disposer d'une application fonctionnelle chaque jour, permet de stopper un projet à tout moment sans perdre l'investissement initial. Cela ouvre des perspectives de nouveaux business model sur le métier du développement.

A propos d'Aspectize

Aspectize est un éditeur de logiciel qui développe et commercialise une solution innovante pour développer des applications métiers distribuées en environnement Microsoft .NET.

Fondée sur une approche descriptive, elle permet de construire logiquement un Système d'Information Distribuées. Basée sur une séparation complète entre le fonctionnel et le technique, elle offre une architecture SOA souple, et réussit parfaitement la séparation Présentation/Traitements/Données.

Aspectize réduit fortement le volume de code de vos applications et les délais de mise en œuvre.

Aspectize permet de faire des choix techniques plus tardivement et limite considérablement les risques liés au développement de vos projets.

Aspectize invite à adopter une démarche agile en permettant la construction incrémentale des applications, et renforce la qualité du développement.