LA MAÎTRISE DES SYSTÈMES D'INFORMATION

ADELI

La LETTRE n° 33

Octobre 1998

Pourquoi un AGL n'est pas une usine

À quoi ressemblerait l'AGL idéal ? À quoi ressemblera l'AGL de l'avenir ?
Dans son récent ouvrage, ***Outils de construction du logiciel****, paru aux éditions Hermès, Yves Constantinidis s'attaque à quelques mythes bien ancrés. Celui de l'AGL considéré comme une « usine à logiciels », est l'un des plus répandus. Dans cet extrait, nous allons voir pourquoi les AGL ne sont pas, et ne seront probablement jamais, des usines à logiciels (software factories) comme on le dit souvent.*

Le mythe de l'usine à logiciels

On compare souvent l'AGL idéal à une usine. D'après cette vision, le logiciel est produit par une équipe d'ingénieurs spécialisés, un peu comme une automobile est construite par une équipe d'ouvriers spécialisés.

- Avec l'aide de l'AGL, le logiciel est d'abord spécifié, dans le respect de la définition des besoins préalablement stockée dans le référentiel de l'AGL.
- Une deuxième équipe de concepteurs prend alors en charge les spécifications détaillées et le découpage du logiciel en modules.
- Les spécifications détaillées de ces modules sont alors prises en charge par des programmeurs.
- Puis chaque module est codé par un spécialiste d'un domaine technique ou d'un domaine métier : interface graphique, accès à la base de données, algorithmes plus ou moins sophistiqués, modules spécialisés.
- Après la phase de programmation, les modules sont assemblés par une autre équipe, comme les pièces d'une automobile sur une chaîne de montage.
- La totalité des fonctionnalités est testée par une équipe d'informaticiens-testeurs, suivant des jeux de tests préalablement conçus et stockés dans le référentiel de l'atelier.
- Le logiciel ainsi construit peut alors entrer en production et passe sous la responsabilité de l'équipe de maintenance.
- Chaque modification, suite à une demande d'amélioration ou à la détection d'une erreur, est répertoriée et enregistrée dans le référentiel de l'outil, garantissant la traçabilité de chaque intervention.

Pour les cas, heureusement de plus en plus rares, de logiciels qui n'auraient pas été conçus, développés et maintenus selon les mêmes principes et avec les mêmes outils, et pour lesquels toutes les étapes de production n'auraient pas été tracées et stockées dans un référentiel, l'AGL idéal dispose d'un ou de plusieurs outils de « rétroingénierie » ou « rétroconception », permettant de reconstituer *a posteriori* les divers documents manquants, en remontant en sens inverse les étapes de spécifications détaillées, de spécifications globales, ou de définition des besoins.

Conçu et réalisé avec un tel atelier, un logiciel a la même finition qu'une automobile sortant d'une usine d'automobiles, et il est aussi sûr d'utilisation qu'un pont construit selon les règles du génie civil.

La réalité

Cette vision de l'AGL idéal, quelque peu caricaturée ici, est décrite par nombre de « méthodologues » et propagée par la presse spécialisée, soutenue dans ses efforts par un nombre impressionnant de fournisseurs d'outils de développement.

Si de tels AGL-usines n'ont encore jamais vu le jour, ce n'est pas faute de disposer de moyens pour concevoir, développer et commercialiser des outils et des référentiels suffisamment puissants. Ce n'est pas non plus par manque de méthodologies adéquates.

L'AGL-usine est tout simplement un mythe, et ceci pour trois raisons : (1) L'activité de développement de logiciel n'est pas une activité de production. (2) Le développement de logiciel, au sens large du terme, ne suit pas un processus linéaire. (3) Dans le monde du développement de logiciel, une trop grande spécialisation nuit à la productivité.

Revenons sur ces trois affirmations :

- *l'activité de développement n'est pas une activité de production.*
  Contrairement au matériel, le logiciel n'a pas à être fabriqué, au sens industriel du terme. Il passe presque directement de la conception à l'emballage. Les tests, la validation, le contrôle qualité, ne concernent quasiment pas la fabrication (qui consiste en réalité à dupliquer des disquettes, à graver un CD-Rom ou à télécharger des fichiers) mais le développement. Les tests, comme chacun le sait, se font au niveau du développement. Il est donc abusif de parler d'usine là où il s'agit en fait de bureaux d'études ;

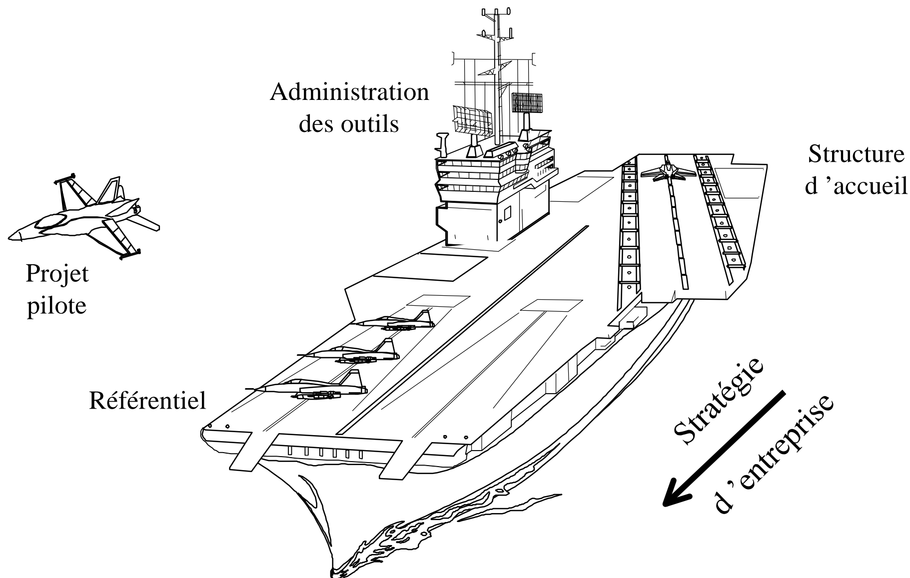
- *le développement du logiciel ne suit pas un processus linéaire.*
  Le modèle de cycle de développement en cascade (waterfall model) n'est qu'une approximation de la réalité, et il a été remis en cause à de nombreuses reprises. Le cycle en spirale est lui-même un modèle qui, bien que se rapprochant plus de la réalité, peut être sujet à controverse. En réalité, un grand nombre d'activités de développement sont conduites, non seulement en parallèle, mais de façon totalement désynchronisée. Et la généralisation du développement par objets, où des groupes de modules (clusters) sont construits de manière quasi indépendante par de petites équipes, et souvent par un développeur isolé, ne va faire qu'accélérer cette tendance. Le modèle de l'usine, et même du bureau d'études traditionnel, est inapplicable à la construction du logiciel ;

- *dans le monde du développement de logiciels, une trop grande spécialisation nuit à la productivité.*
  Le métier de programmeur qui se contente de coder en Cobol des spécifications hyperdétaillées écrites en français a presque disparu. Ceci n'est pas seulement dû à une inflation des termes, par ailleurs très réelle. C'est le métier lui-même qui appartient au passé. La division « verticale » du travail augmentait la productivité des usines de montage du temps de la Ford T. Dans une équipe d'informaticiens, le seuil à partir duquel cette division du travail devient contre-productive est vite atteint.

En effet, la difficulté de construire un système cohérent ne réside pas dans la conception des briques élémentaires, mais dans l'architecture de l'ensemble et dans la constitution d'interfaces stables entre les différents modules. Ces trois arguments suffisent à prouver que l'usine ne peut, ni ne doit, servir de modèle aux ateliers de génie logiciel.

L’avenir : le porte-avions

Si l’AGL idéal n’est pas une usine, à quoi peut-on alors le comparer ? Peut-on trouver une analogie avec un secteur d’activité existant ? C’est l’aéronautique qui nous apportera une analogie : l’AGL de l’avenir ressemblera de moins en moins à une usine et de plus en plus à un porte-avions.



Il est nécessaire, pour un chef de projet, de disposer d’une large part d’initiative, de suffisamment d’autonomie, tout en se conformant à l’orientation générale qui lui est imposée par l’entreprise. En d’autres termes, il doit être seul maître à bord de son projet, après avoir reçu une mission clairement définie qu’il s’est engagé d’accomplir.

Le porte-avions est un référentiel muni de points d’accès permettant l’échange de modèles avec les autres projets. Chaque projet reçoit un plan de vol précis et travaille sur un objectif précis avec une équipe de taille réduite. Une fois l’objectif traité, l’avion (le projet) revient à sa base (le référentiel), sachant néanmoins que cette base peut elle-même avoir évolué.

Chaque projet apporte des informations pertinentes au référentiel global, c’est-à-dire uniquement des informations pouvant présenter un intérêt pour les autres projets ou pour l’ensemble de l’entreprise. Ces informations peuvent être retravaillées au niveau du référentiel global avant d’être mises à la disposition des autres projets. En cours de « vol », le chef de projet peut recevoir des directives nouvelles tout en conservant une totale autonomie opérationnelle.

L’AGL idéal est donc un ensemble hétérogène d’outils, aussi différents les uns des autres que peuvent l’être un hélicoptère de combat d’un avion de chasse, mais parfaitement communicants. Ils disposent chacun d’une large autonomie, tout en pouvant opérer sur un objectif global bien défini. ▲

Yves Constandinidis

Extrait de l’ouvrage *Outils de construction du logiciel*, Hermès, 1998.
© Éditions Hermès, 1998

Les composants horizontaux

Carte routière de la réutilisation du code logiciel

Cet article traite de l'interaction entre le composant vertical qui encapsule les règles métier et le composant horizontal qui contient les spécifications techniques. Le développement des composants verticaux ou métiers s'amplifie de jour en jour ; bientôt ceux-ci seront disponibles sur catalogue. Néanmoins point de salut pour le composant vertical sans de solides infrastructures horizontales.

La démarche de développement à base de composants impacte de façon très positive la réutilisation du logiciel et la productivité générale des équipes. Mais attention, la communauté informatique a formulé de telles déclarations au sujet des technologies orientées objet !… Pourtant, les observateurs de l'industrie rapportent que seulement 15% des développements orientés objet conduisent à la réutilisation.

Avec le temps, la difficulté du développement devient proportionnelle à l'exigence et à la complexité des systèmes d'information. La complexité a généralement augmenté plus vite que la qualité des méthodologies et des outils. Elle s'accroît exponentiellement avec les systèmes transactionnels pour le web. Les applications Web doivent pouvoir monter en charge massivement, elles doivent être sécurisées et livrables en “années Internet”, c'est-à-dire dans un délai de 3 à 6 mois. Pour répondre à ces besoins, nous sommes forcés d'évoluer d'une architecture à 2-tiers vers une architecture à 3-tiers ou à multi-tiers. Pour remplacer le client et le serveur, nous avons désormais le navigateur, le serveur web, le serveur de traitements, les serveurs de bases de données et le middleware applicatif reliant l'ensemble. Nos composants doivent donc prendre en compte cette complexité si nous voulons réellement profiter de cette technologie prometteuse.

Le composant est en fait un pas en avant sur l'objet. Il n'est pas, néanmoins, aussi bien défini que l'objet. Il existe une base théorique bien établie pour ce qui concerne la méthodologie orientée objet. Même si certains développeurs ne la comprennent pas, ne l'utilisent pas correctement ou ne sont pas d'accord, il existe un corps de matériaux de référence qui définit précisément l'objet et en réglemente l'utilisation. Le composant ne dispose pas d'un tel pedigree.

Lorsque nous parlons de composant, nous ne faisons pas référence à un quelconque contrôle graphique mais aux fonctionnalités métiers à l'intérieur du composant. Un composant est fait de règles métier, de fonctionnalités d'application, de données, de ressources qui sont encapsulées pour permettre la réutilisation dans de multiples applications. En outre, les composants doivent être portables et inter-opérables au travers des applications.

Composants verticaux : besoins fonctionnels

Les développeurs qui travaillent avec des composants ont tendance à penser que les composants sont verticaux. Prenons, par exemple, la transaction de dépôt bancaire ou la transaction de remboursement de frais de santé, qui effectue l'ensemble des tâches, de l'interface utilisateur à la mise à jour de la base de données. Ceci peut être mis en œuvre en tant qu'entité indépendante incluant le code pour toutes les couches dans un environnement multi-tiers (l'interface utilisateur, l'interface du serveur web, l'accès au middleware et l'accès à la base de données). En d'autres termes, nous pouvons définir un composant vertical comme un objet qui met en œuvre un traitement fonctionnel. La phase d'analyse d'un projet

nous dit que nous en avons besoin pour une tâche fonctionnelle spécifique. Un composant vertical peut être défini en termes d'entrées, de règles métier et de résultats. C'est ce qu'un ingénieur appellerait une boîte noire - un mécanisme qui accomplit une fonction spécifique sans devoir regarder sous le capot. Tout ce qui nous intéresse, c'est ce qu'il fait et pas comment il le fait.

Lorsqu'ils sont conçus correctement, les composants verticaux peuvent être insérés dans n'importe quel système et accomplir les traitements requis. Mieux encore, si les règles métier changent, le composant vertical peut être modifié sans que les autres parties du système soient affectées.

Lorsque nous construisons un système, celui-ci est composé d'un certain nombre de composants verticaux. Notez que lorsque nous spécifions un composant vertical, nous ne nous intéressons pas à l'accès aux données, à l'interface utilisateur ou au déploiement de l'application sur des serveurs multiples. Ce sont des détails qui sont cachés dans le composant vertical.

De nombreux départements informatiques développent ainsi, car ils répondent à des exigences fonctionnelles (qui définissent généralement l'ensemble des aspects d'une transaction) plutôt qu'à des spécifications techniques. Les spécifications techniques incluent typiquement la décomposition fonctionnelle qui permet à une application d'être développée horizontalement. Et les spécifications techniques sont absolument essentielles lorsque le travail est réalisé en environnement orienté objet pur (ce qui peut expliquer l'échec d'un nombre élevé de projets OO). Mais le temps requis pour la préparation d'une spécification technique est un luxe que bon nombre de départements informatiques ne peuvent se permettre.

Composants horizontaux : spécifications techniques

Si nous regardons la chose sous un angle différent, lorsque les départements informatiques prennent le temps de préparer d'abord les spécifications techniques, le travail est fréquemment décomposé horizontalement.

Par exemple, lors du développement d'un système bancaire multi-tiers, un programmeur de base de données va écrire un service qui valide un numéro de compte et un autre qui crédite un dépôt dans la base de données. Un programmeur d'interface graphique utilisateur va créer un écran qui recueille le numéro de compte et le montant de l'utilisateur et l'envoyer (en utilisant un middleware) à un service qui contient les règles métier pour les dépôts. Ce service va alors appeler les deux services de la base de données, obtenir les résultats, et les renvoyer à l'interface utilisateur. S'il s'agit d'une application web, un programmeur web va traduire les données IHM en HTML (ou Java) et les envoyer au navigateur de l'utilisateur. Ainsi, pour une transaction, nous avons potentiellement 4 (ou davantage) programmeurs impliqués et 5 interfaces à négocier. Nous avons des développeurs qui conçoivent les services de la base de données, des développeurs qui conçoivent l'interface homme/ machine, des développeurs qui gèrent le serveur web, et des développeurs qui gèrent le middleware. En d'autres termes, la structure de notre équipe de développement est horizontale plutôt que verticale ; le résultat final est donc horizontal plutôt que vertical.

La difficulté qui résulte du scenario de développement horizontal est que le centre d'attention du problème fonctionnel à résoudre se déplace vers des détails techniques, et souvent les besoins fonctionnels en pâtissent. Les programmeurs ont tendance à se concentrer sur leur partie plutôt que sur l'objectif général à atteindre. Ils se soucient, par exemple, de questions concernant le middleware applicatif, comme les sockets, les procédures d'appel distantes (RPC), les gestionnaires de requêtes objet (ORB), les requêtes de messages asynchrones et la technologie à file d'attente. Tout ceci détourne l'attention du programmeur de son vrai travail qui est de se concentrer sur les problèmes fonctionnels.

Les sept API

Inutile de dire que le concept du composant vertical est plus qu'intéressant ; pourtant sa mise en œuvre n'est pas limpide. C'est vrai que toutes les règles métier sont encapsulées dans un objet, mais cet objet possède au moins 7 interfaces de programmation (API), ainsi que le montre la figure 1.

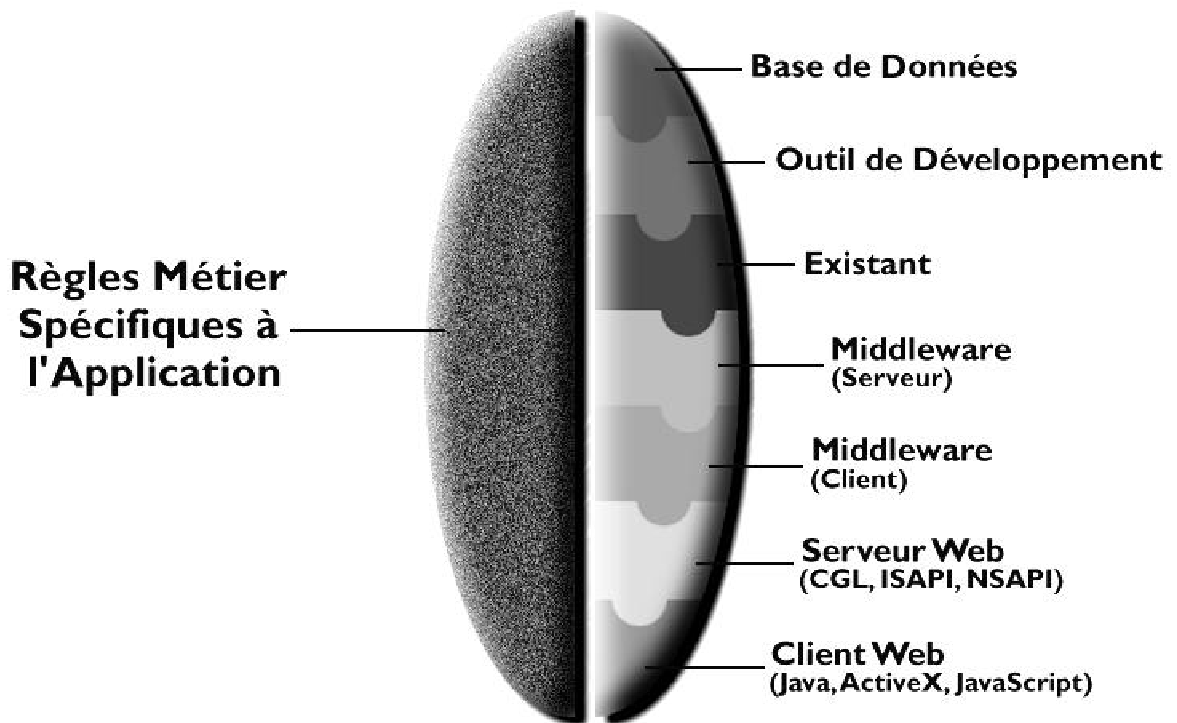


Figure 1

- D'abord le code de l'interface utilisateur qui utilise une, ou plusieurs, API Web.
- Ensuite, l'interface entre le serveur Web et l'application utilisant CGI, ISAPI, ...
- L'API middleware côté client.
- L'API middleware côté serveur.
- S'il faut accéder à un mainframe, alors existe une API supplémentaire.
- L'API au SGBDR.
- Et finalement, l'API à l'outil de développement utilisé.

Lors de l'utilisation de composants verticaux multiples, chaque composant vertical utilise les 7 mêmes API, et repose sur du code accédant à chacune de ces API. Si une API doit être changée, elle doit être modifiée dans l'ensemble des composants verticaux.

En clair, il s'avère donc nécessaire de trouver une solution combinant le meilleur des deux approches, horizontale et verticale.

Synergie

Si nous regardons une fois encore les composants verticaux, nous nous apercevons que chacun exécute à peu près les mêmes appels aux API. Forts de cette connaissance, nous pouvons alors définir le composant horizontal comme API traversant l'ensemble des composants verticaux comme indiqué dans la figure 2. Nous avons alors à faire à des traitements qui se trouvent à l'intérieur des composants horizontaux et à des besoins fonctionnels qui sont dans les composants verticaux.

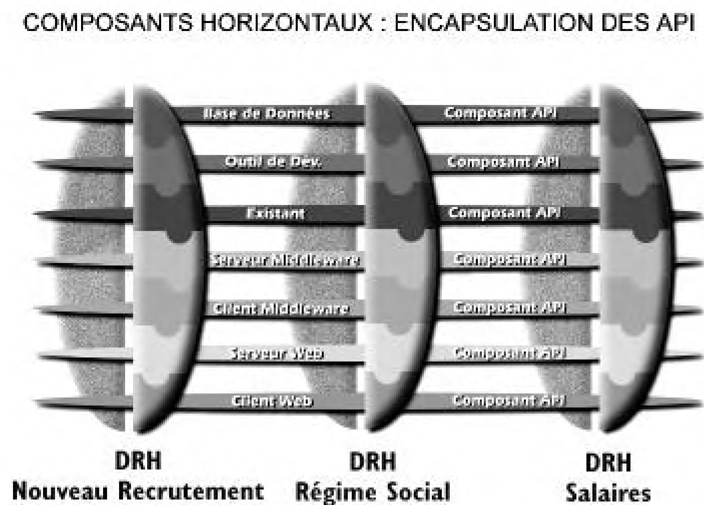


Figure 2

Nous pouvons maintenant extraire les composants horizontaux dans leur propre espace comme l'indique la figure 3. Nous fournissons un ensemble cohérent de fonctions d'emballage pour les composants horizontaux, et réduisons tous les composants horizontaux à une seule API. Ceci supprime, par exemple, les détails relatifs à la communication interprocess des composants verticaux et permet aux développeurs de se concentrer sur la logique fonctionnelle plutôt que sur les communications. Nous avons essentiellement réduit le traitement à une API. Naturellement, quelqu'un va devoir construire cette API. C'est une tâche pour vos développeurs les plus pointus, parce que l'efficacité et l'utilisation de votre application dépend de la robustesse de l'API.

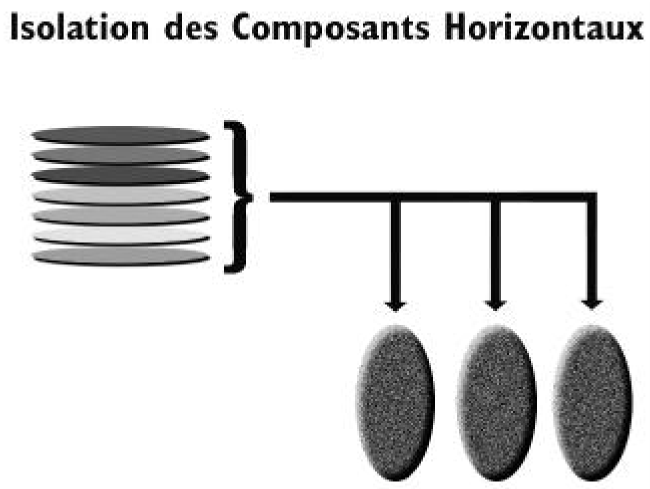


Figure 3

Caractéristiques des composants horizontaux

Les composants horizontaux doivent être intelligents ; ils doivent pouvoir obtenir de l'information sur le composant vertical qui les utilise et être capables de répondre au besoin du composant vertical. Prenez, par exemple, l'API de la base de données du composant horizontal. Si un développeur de Client/Serveur à 2-tiers désirait accéder à la base de données, la procédure normale serait d'écrire du SQL. Dans un environnement multi-tiers, cependant, le code de l'interface à la base de données est sur le serveur, pas sur le client où résident d'habitude la plupart des outils client/Serveur. Certes, nous pouvons encore concevoir le composant horizontal de la base de données pour qu'il accepte une requête SQL comme paramètre, mais nous perdons un peu de l'avantage de notre outil. Ainsi, les outils à 2-tiers ne sont-ils donc pas une bonne solution dans un monde multi-tiers. Au lieu de ceci, nous devrions être capables de faire en sorte que les composants horizontaux coopèrent et échangent des données entre eux.

L'utilisation de composants horizontaux permet au développeur de créer un écran avec des champs. L'écran contient aussi des informations quant à la relation entre les champs et les tables dont ils représentent les colonnes, et toute information clé relative au schéma de la base de données (essentiellement un arbre d'interrogation). Ces métadonnées peuvent être obtenues par interrogation à la base de données et sauvegardées dans un magasin de métadonnées ou référentiel. Lorsque l'utilisateur requiert un événement (par exemple, le rapatriement de données pour visualisation), le composant horizontal de l'interface utilisateur protège, dans un premier temps, tous les champs, puis passe la liste des champs, l'arbre des interrogations, et l'événement utilisateur ("sélectionner" dans cet exemple) au composant horizontal de la base de données.

Le composant horizontal de la base de données construit alors dynamiquement la requête SQL nécessaire et renvoie les résultats au composant horizontal de l'interface utilisateur. Si l'utilisateur avait choisi un autre événement de la base de données ("insérer" par exemple), les actions du code de l'application seraient identiques : l'appel d'un accès à une base de données. Le composant horizontal de l'interface utilisateur obtiendrait alors des valeurs de tous les champs et les passerait avec la liste des champs, l'arbre d'interrogation et l'événement utilisateur ("insérer" dans cet exemple) au composant horizontal de la base de données. Nous avons donc ici un morceau de code commun (le composant horizontal de la base de données) qui est utilisé lors de chaque instance d'accès à la base de données. Le même principe s'applique aux autres composants horizontaux ; nous avons essentiellement fabriqué un code de traitement réutilisable.

Il est important de pouvoir prendre en compte les situations uniques ou inhabituelles, ou les facteurs qui n'étaient peut-être pas apparents lors de la création des composants horizontaux. Pour être vraiment robustes, les composants horizontaux doivent être extensibles à la fois globalement et pour des transactions spécifiques. Ainsi, pour une transaction spécifique, vous voudrez peut être modifier le composant généré en SQL mais garder intact le modèle général.

Plus que l'interface à la base de données

Nous pouvons appliquer la même abstraction aux règles métier et à l'interface utilisateur. C'est-à-dire, stocker les règles métier dans un référentiel et les contrôles graphiques dans une librairie de classes (ou mieux encore, dans le même référentiel).

Ainsi que le montre la figure 4, les composants verticaux héritent leur traitement des composants horizontaux, leur comportement de la librairie de classes et leurs règles métier du référentiel. Le référentiel saisit les métadonnées de l'application, y compris le schéma de la base de données, les règles métier, la personnalisation des contrôles graphiques, les graphiques - tout ce qui est nécessaire pour construire des composants verticaux et fournir des informations aux composants horizontaux. Les entrées dans le référentiel sont un exemple d'héritage multiple - un concept de la technologie orientée objet rarement appliqué dans la pratique. Un objet du référentiel hérite son comportement de la librairie

des contrôles graphiques de la plate-forme sur laquelle fonctionne l'application, ses métadonnées du schéma de la base de données et ses règles métier du développeur.

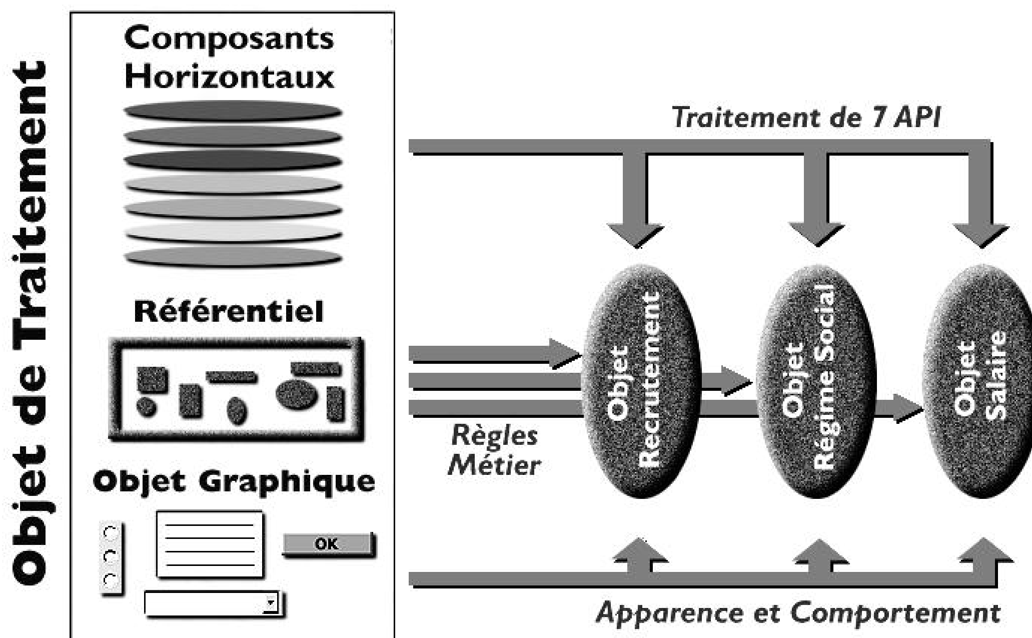


Figure 4

Une autre abstraction peut être effectuée sur les composants horizontaux. Si un objet de traitement est défini comme un ensemble de composants horizontaux, les composants verticaux (ou applications) ont à faire alors à un seul objet pour tout le traitement. Les composants horizontaux, à l'intérieur de cet objet de traitement, communiquent entre eux, en plus de communiquer avec les composants verticaux. Le fait de gérer de façon centrale l'évolution des composants horizontaux se répercute sur les composants verticaux. S'il est conçu correctement, l'objet de traitement peut être réutilisé par l'ensemble des projets et départements de l'entreprise.

Cette approche apporte d'autres avantages. Puisque les objets dans le référentiel héritent leur comportement de la plate-forme, la même application peut être exécutée sur différentes stations de travail, et adopter l'interface utilisateur appropriée pour cette plate-forme. Ainsi, la même application peut-elle être utilisée sur le Web, Windows, Macintosh, Motif ou sur des terminaux passifs. Les composants verticaux sont inchangés sur les plates-formes.

De même, puisque le référentiel peut encapsuler l'interface à la base de données, les applications sont alors indépendantes du moteur de la base de données choisi, et les composants verticaux sont inchangés si le moteur de la base de données change ou si le même composant vertical doit accéder à des bases de données hétérogènes et multiples.

Nous pouvons également élargir l'environnement sans devoir modifier les composants verticaux. Supposons, par exemple, que nous devons ajouter une couche de sécurité supplémentaire comme le chiffrement de données. Nous pouvons créer un nouveau composant horizontal et l'ajouter à l'objet de traitement. Nous devrons, peut être, modifier les composants horizontaux adjacents, mais l'ajout est transparent pour les composants verticaux utilisant l'objet de traitement. Lorsque les composants verticaux sont affectés (par exemple, quand des niveaux multiples de sécurité sont ajoutés, ceci nécessitant que les privilèges de l'utilisateur soient vérifiés), la modification peut fréquemment être faite par l'ajout d'une propriété dans le référentiel qui peut alors être distribuée aux composants verticaux.

Nous pouvons nous demander quelles surcharges de traitement pourraient être provoquées. Il n'est pas facile de répondre. Dans la plupart des applications, les vraies raisons d'une performance médiocre sont dues à une technologie qui est sous-utilisée ou mal utilisée ou à une mauvaise conception plutôt qu'au code. Mais des composants horizontaux mal conçus peuvent réellement causer des problèmes de performance. C'est pourquoi il est capital d'avoir recours aux développeurs les plus pointus lors de la construction de l'infrastructure des composants horizontaux.

Middleware transparent

Nous n'avons pas encore évoqué la question relative aux communications entre les couches. Ce middleware applicatif est critique dans les applications multi-tiers, mais est difficile à apprendre et à utiliser. Pourtant, le service qui traite les règles métier est tout ce qui est nécessaire si les composants horizontaux du middleware et de la base de données sont assez intelligents. Nous avons déjà montré comment faire cela pour le composant de la base de données. Les couches du middleware requièrent aussi de l'intelligence supplémentaire pour être transparentes. Ici, le tour consiste à réaliser un référentiel décrivant les composants verticaux et qui sera utilisé par les composants horizontaux afin de les instancier correctement. L'accès à ce référentiel pourra être effectué aussi bien par la machine du client que par celle du serveur. Le référentiel servira à décrire l'interface des composants verticaux. Ainsi, tout ce qu'un client doit faire dans un composant vertical est d'appeler l'objet de traitement requérant le service. Ce fichier demandant l'interface au middleware peut résider sur un serveur (ou des serveurs multiples pour redondance) ; il est accessible à la fois par les clients et les serveurs. Le composant horizontal du middleware client accède au fichier de l'interface, identifie le service et les arguments dont il a besoin, il fait appel à ce service et extrait les données nécessaires du client. Le composant horizontal du middleware du serveur accède au même fichier d'interface et détermine les caractéristiques du client qui y fait appel. Il sait alors quelles sont les données qu'il attend et celles qu'il doit renvoyer.

Intégration de l'ensemble

L'état de l'art actuel en matière de développement multi-tiers n'atteint pas les gains escomptés. Nous pouvons débattre des raisons ; en tous cas il est impérieux de faire quelque chose si nous voulons développer de réels systèmes d'entreprise respectant les contraintes de budget et de temps.

Nous avons démontré qu'une façon d'obtenir des améliorations significatives de productivité de développement de logiciel est basée sur une approche hybride reposant sur l'utilisation de composants horizontaux et verticaux permettant aux composants verticaux d'hériter de composants multiples. La clé est de bâtir des composants horizontaux intelligents qui répercutent les structures des données dans les composants verticaux des applications. ▲

Larry Finch , traduit et adapté par Fabienne Dravers

Fiabilité des programmes informatiques

« La Science », de janvier 1993, a publié un article de Bev Littlewood et Lorenzo Strigini intitulé « La fiabilité des programmes informatiques ».

L'article est fort intéressant et contient de bonnes réflexions et incitations à la prudence au sujet des tests. On ne teste jamais assez, sauf quand cela devient trop cher.

Mais une phrase m'inquiète : « En théorie, on peut créer un logiciel parfait ».

Et bien, c'est faux. D'abord parce qu'il faut dans ce cas définir le mot parfait.

D'autre part, un certain Kurt Gödel, mathématicien viennois, publie en 1931 et 1934 des articles sur les systèmes formels. Dans ces articles de logique mathématique il démontre qu'un système formel, est soit incomplet, soit incohérent. La cohérence[1] étant la faculté de ne pas renfermer de contradictions potentielles, on s'aperçoit que la perfection n'est vraiment pas de ce monde. Vous me direz que ceci s'adresse uniquement à des systèmes formels, c'est à dire bâti sur une logique utilisant des opérateurs mathématiquement définis. Et bien, a fortiori si l'on construit des programmes avec des opérateurs non strictement définis, le résultat final ne peut être que pire que pour les systèmes formels. Ainsi, dire que les programmes parfaits sont possibles est aberrant. Il faut le crier bien fort dans une civilisation courant après toutes sortes de perfection : non, c'est illusoire, en cherchant la perfection on souhaite éliminer l'imparfait, éliminer les insectes dits nuisibles avec des tonnes d'insecticides, aromatiser les aliments, pratiquer l'eugénisme génétique, stériliser les trisomiques, génocider les minorités indésirables, donc se compromettre.

Kurt Gödel ne s'y est pas trompé, sentant l'Anschluss de Hitler arriver sur l'Autriche, il émigra aux États-Unis. ▲

Michel Demonfaucon

Bibliographie :

G.Chazal. *Éléments de logique formelle*. Hermès.1996.
A.Delessert. *Introduction à la logique*. Presses polytechniques romandes. 1988.
R.Cori, D.Lascar, *Logique mathématique*. Tome 2. Masson. 1994.

---

[1] *Les logiciens préfèrent le mot « consistance ».*

Architecture distribuée et génie logiciel français

Impacts sociaux-professionnels

Un nouveau défi se profile à terme pour le logiciel français, secteur de l'informatique française déjà mis à mal par trente années d'incurie.

Le logiciel français tant à l'honneur dans les années 90 du fait de quelques grandes SSII aujourd'hui mal en point, se retrouve avec en face de lui une nouvelle déferlante, celle de la vague de l'informatique distribuée, qui pointe peu à peu le bout de son nez par-dessus les grandes vagues froides et venteuses de l'Atlantique Nord.

Bien que tenue en peu d'estime, cette "ID" est relancée par un vecteur puissant : l'interconnexion des réseaux de télécommunications informatiques existant. Cette astuce extraordinaire, mise en place par les chercheurs américains de la guerre froide, consiste non pas à construire quoi que ce soit, mais littéralement « squatter » les réseaux existants, administrations, entreprises, centres de recherche, qui le veulent bien (et en tire un profit : l'information de pointe).

Pour peu d'investissements, un brin d'imagination et de débrouillardise, les chercheurs associés se sont construit un réseau maillé indestructible du fait de ses nombreuses routes variées pour aller d'un point à un autre. En France Transpac, par exemple, fait la même chose, mais sur un réseau appartenant à France Telecom, entreprise publique. Pour Internet, tous les réseaux, de quelque nature que ce soit, sont utilisables ; l'esprit coopératif y trouve son compte.

Cette volonté de passer sur tous les chemins possibles nécessita de mettre au point des passerelles de connexion hétérogènes tel que le fameux TCP/IP. Mais TCP/IP est presque une brute de simplicité à côté de plates-formes de développement harmonisé, quelle que soit la machine matérielle ou le système d'exploitation. On voit donc arriver des environnements d'informatique distribuée qui découpent vos applications en rondelles, comme le saucisson sec, avec des tranches de finesse variable suivant votre capacité. L'application, la donnée sont devenus sécables et nomades. Elles ne sont plus localisables statiquement mais se répartissent physiquement et dynamiquement d'après la charge des différents systèmes impliqués. Bien sûr, au cas où vous tiendriez à des choses très sérieuses - comme des comptabilités - la sécurité, l'intégrité dite référentielle sont garanties, et ceci non plus pour des petites quantités de données mais jusqu'à 60 milliards d'enregistrements indexées par application. Il est "évident" que le stockage à plat est quasi illimité. Plusieurs dizaines de milliards de milliards[1] d'enregistrements peuvent être stockés dans un système de plusieurs centaines de sites distribués à travers la planète.

Si, de plus, vous souhaitez des temps de réponse acceptables, les moniteurs transactionnels distribués (pour les anciens : TDS ou CICS, mais éclatés sur de multiples systèmes hétérogènes !) vous apporteront la fiabilité et la rapidité en temps nécessaire ; ils vous permettront aussi si vous le souhaitez, de débiter un compte dans une base de l'éditeur XYZ et de créditer sa contrepartie dans une base de l'éditeur ABC.

Qu'apporte la France dans ce paysage ? Bull offre un début de réponse avec UNiCics et autre ISM; mais la France ne brille pas particulièrement dans ce domaine. Pourtant il apparaît que l'architecture distribuée est aussi naturelle que la tarte aux pommes (les américains adorent ce dicton, pour moi,

---

[1] 2^{64} moins 10 enregistrements nécessaire à la conservation de descriptifs de la base.

j'aime encore mieux la tarte aux pommes avec de la vanille et un peu de cannelle). Donc pouvoir acheter tout nouveau "bouzin" clignotant de ses mille feux (un effet de l'imaginaire informatique) et le coller au boulot dans un réseau, peu importe lequel, me paraît l'issue naturelle de ce long et usant affrontement entre utilisateurs et constructeurs.

De l'autre côté de l'échelle se retrouve un univers microcosmique où l'ensemble des attributs grands systèmes SGBD relationnels est inclus. C'est la bureautique moderne structurée par Microsoft dont nous dirons un mot dans un prochain article.

Le fait de pouvoir disposer des outils communément admis comme utiles et agréables tout en s'équipant de matériels dont le prix unitaire ne cesse de chuter, est considéré de plus en plus comme du consommable. C'est du budget de fonctionnement et non d'investissement, et cela fait que cette logique "un employé=un ordinateur" détruit peu à peu la sphère d'influence "grands systèmes" et crée une désaffection de leur utilisation. Il en découlera une nécessité de restituer le métier de l'informaticien et même de redéfinir les métiers de l'informatique, dont la structuration hiérarchique et socio-professionnelle est profondément modifiée, une fois de plus, par les nouvelles technologies. Les nouveaux informaticiens n'ont pas cherché à se démarquer des anciens en cherchant un nouveau nom de métier. Microticien est-il respectable ? Sûrement pas, il valait mieux conserver le terme générique bien que celui-ci devrait être transformé pour les anciens (dont je suis) en macroticien.

On assiste en fait, ou l'on participe, à une déconstruction analogue à celle de la déconstruction de la mécanographie dans les années 1950 à 1960. Le phénomène est de même ampleur mais reste masqué par la déconstruction générale de l'économie des années 1990 et la profession informatique recherche encore un esprit de solidarité jamais totalement atteint et battu en brèche par l'individualisme des générations montantes.

Le chacun pour soi est le seul moyen de ne pas sombrer en groupe. La généralisation de l'ordinateur "personnel" relié en réseau mais pouvant fonctionner de manière autonome est-elle le reflet des comportements sociaux ? Serons-nous toujours prêt à demander collectivement des avantages supplémentaires mais rarement disponibles individuellement pour des actions de générosité, de partage et d'entraide ? ▲

Michel Demonfaucon

Guillaume Portails

Son accès, Macrodur et la fin du code

La célèbre société, installée dans l'État porteur du nom du premier Président des États-Unis, s'est senti des ailes lorsque le dernier petit bijou de la suite bureautique, dont l'inspiration première vient d'une invention du Fondateur lui-même pour faciliter ses travaux d'étudiant en comptabilité finances, a décidé de franchir un grand pas et de faire sombrer définitivement sa pierre d'achoppement : le système de gestion de fichiers dbaseIV du défunt Ashton-Tate.

Déjà bien amoindri par les problèmes juridiques intentés aux auteurs de ce produit qui n'aurait été que l'heureux résultat d'un plagiat, dbaseIV est définitivement absorbé par un accessoire anodin de la suite, qui se présente graphiquement par une icône représentant une clé.

On peut d'ailleurs continuer à fonctionner en mode mixte, avec les vieilles applications dbaseIV tout en intégrant les données dbaseIV dans le nouveau système. Je fais cela chaque jour plusieurs fois de suite.
Cela ne concerne pas directement ADELI, dira-t-on, et d'ailleurs nous ne faisons pas de publicité. Mais il y a là, rampant subrepticement, un fait nouveau, longtemps pressenti, jamais cru par les informaticiens chevronnés… la fin du code.

Ayant compris que DB2 n'était en somme qu'un gigantesque tableur (cette comparaison est stupide bien sûr, mais très efficace pour faire comprendre l'idée de base) - et qu'est ce qu'un fichier sinon un tableur très appauvri ? - le grand éditeur décide de porter un grand coup dans le monde des SGBD, tout en avançant à pas de loup dans ce monde dominé par les grands spécialistes qui font, eux, disent-ils, des produits sérieux et pas des gadgets.

À partir du moment ou l'on sait gérer astucieusement des tableurs en les faisant obéir aux règles de traitement des ensembles (union, intersection, etc.) on a gagné le droit de faire enfin ce que tout utilisateur rêve de faire sans jamais avoir osé le demander : créer son application en faisant des clics !

On arrive ainsi, assez facilement, à créer totalement graphiquement une petite application de gestion de production comportant 114 tables et 250 requêtes SQL. Ne vous inquiétez pas, les requêtes SQL sont générées automatiquement à partir de graphiques style MCD en tirant des liens entre types de colonnes et en déclarant leur cardinalité. Si la cardinalité est encore un obstacle conceptuel, d'"office" (excusez-moi) on vous colle une cardinalité implicite de 1 à 1. Mais le système a encore besoin d'un guide efficace, car si vous ne tirez pas sur le lien pour le faire apparaître graphiquement entre deux tables, une requête sur les deux tables génère un produit cartésien des deux ensembles d'éléments, ce qui est rapidement fatiguant en temps de traitement et espace disque.

Pour lier le tout, 100 états de gestion avec quelques dizaines de macros événementielles.
L'événement c'est quand on clique sur quelque chose de préférence style bouton, qui se manifeste sur l'écran.
Alors le clic est interprété comme une vraie décision et on lance un traitement équivalent à quelques 100 000 lignes de notre vieux Cobol si c'est nécessaire.
Aucun code là-dedans et en tout cas rien de très monnayable par des sociétés de service ou des indépendants. En effet, le code source en quelque sorte protégé par la sécurité de la base est inclus dans le système à jamais.

Mais les désespoirs de nos confrères ont été entendus et le Fondateur Guillaume Portails veut bien dans la version 1997, permettre de placer un "runtime" ou autrement dit (que dit l'AILF ?) un système de code d'exécution[1].
C'est très bien on pourra mettre trois fois rien dans le code et le vendre très cher car personne ne le verra alors que tout est fait par graphique boutons et clic de souris.
Pourvu que cela dure !

Le système se veut également orienté client-serveur et fonctionne sur NT. L'option multi-utilisateurs avec partage de base et verrouillages est prévue.
Si jamais cela coince, SQL Server, une version réduite de Sybase, est disponible pour faire le répartiteur des données centralisées.
Et comme un malheur n'arrive jamais seul, Intel ne baisse pas les bras et propose des systèmes multiprocesseurs pour NT mettant l'ordinateur d'entreprise à la portée du premier bureauticien du coin de rue !

Bon, on verra bien, mais en tout cas la documentation fonctionnelle de l'application n'est pas générée automatiquement. Il y a encore du travail pour quelques bons rédacteurs techniques en informatique.

Ah, j'oubliais, le nombre d'octets gérables par une base est de 1 giga-octets, mais vous avez droit à un nombre illimité de bases reliées entre elles.

Surpuissants sont les produits du Pacifique Nord ! On avait déjà remarqué que les Indiens du coin érigeaient de très grands totems face à l'océan. Les esprits des totems sont-il réincarnés dans ce logiciel infernal ? ▲

Michel Demonfaucon

[1] De plus en cherchant bien, mais il n'y en a plus beaucoup, il existe un générateur de run-time pour les versions pré-historiques telles que 2.0.

Le mondial des AGL…

Dégager, à coup sûr, le meilleur AGL d'une offre mondiale

Pour rompre avec la routine des traditionnels XXXXscopes, ADELI envisage d'organiser la comparaison des différents types d'ateliers de génie logiciel, sous une forme plus moderne et plus spectaculaire.

Une nouvelle méthode de confrontation des AGL

Nous indiquons, ci-dessous, les étapes de cette nouvelle méthode.

Sélectionner 32 AGL concurrents

Les 32 AGL en compétition seront répartis en 4 zones géographiques :

- 8 AGL américains ;
- 8 AGL français ;
- 8 AGL issus de pays de l'Union européenne (hormis la France) ;
- 8 AGL autres (suisses, canadiens, indiens, roumains, russes, chinois, japonais, etc.).

Si le nombre d'AGL est trop important ou mal équilibré, organiser des éliminatoires locales, selon un règlement propre à chaque zone.

Constituer des groupes

Mettre les noms des 8 AGL d'une même zone dans un chapeau. Pour constituer un groupe, tirer un nom dans chaque chapeau ; ce qui donne 8 groupes homogènes réunissant chacun : un AGL américain, un AGL français, un AGL européen, un AGL autre.

Écarter doucement 2 AGL dans chaque groupe

Dans chaque groupe, opposer chaque AGL à chacun des 3 autres du groupe. Ce qui fera 48 combats singuliers.
Attribuer des points selon le résultat de chaque rencontre ; par exemple, 3 points à l'AGL vainqueur, 0 à son adversaire malheureux, 1 point à chaque AGL en cas d'égalité. Tout autre barème est valable.
Seuls, les deux premiers du groupe sont qualifiés pour la suite de la compétition.

Organiser des affrontements décisifs

Planifier des rencontres bilatérales à partir des 16 meilleurs AGL, par élimination directe selon un tableau préétabli qui brasse les rescapés des différents groupes.
Au premier tour, ne resteront que les 8 vainqueurs, au suivant 4, puis 2 et enfin le meilleur.

Couronner le champion

La primauté de l'AGL vainqueur sera célébrée par l'attribution solennelle d'un trophée (une statuette de scribe). Ce trophée sera remis en jeu tous les 4 ans. L'AGL qui gagnerait le trophée trois fois de suite le garderait définitivement.

La règle du jeu

Sans entrer dans les détails, nous donnerons, ci-dessous, quelques éléments du règlement de ce concours.
Chaque rencontre oppose deux équipes aux couleurs de leur AGL.

Chaque équipe doit mettre en difficulté le produit de l'équipe adverse par des attaques menées par combinaison de commandes des fonctions :

- montrer qu'une fonctionnalité promise dans la documentation n'est pas réalisable ;
- planter le produit adverse en moins de 20 commandes ;
- etc.

La première équipe qui réussit marque un but.

L'équipe qui demande au produit adverse de réaliser une fonction que son propre produit est incapable d'accomplir marque un but contre son camp.

Seront punies, d'un carton rouge, les fautes suivantes :

- modification de son AGL pendant la partie ;
- utilisation d'une simulation préenregistrée ;
- tacle de la souris ;
- dopage de la configuration supportant son AGL ;
- contamination par virus du produit adverse ;
- etc.

Cette méthode évite le piège de la transitivité. L'AGL A peut battre l'AGL B, l'AGL B peut battre l'AGL C, ce qui n'interdit pas à l'AGL C de battre l'AGL A.

Il n'est pas impossible qu'un AGL battu dans un groupe, rencontre à nouveau le même adversaire en finale et le batte.

Ainsi, cette méthode réintroduit la glorieuse incertitude du sport, en entretenant l'indétermination sur le résultat de chaque rencontre.

Le financement

Cette compétition s'avère longue et coûteuse. Aussi, faut-il songer à en assurer sérieusement le financement.
Les rencontres seront publiques et les places seront payantes.
Les grands éditeurs seront invités à apporter leur contribution financière en échange de l'apparition de leur logo dans les salles de confrontation.
Nous autoriserons les partisans des différents AGL à se regrouper en clubs et à venir participer à la rencontre et à supporter leur AGL à grand renfort de trompettes et de banderoles. La vente d'objets promotionnels dérivés constituera un appoint de recette.
Les rencontres principales seront télévisées. Les produits, destinés aux informaticiens de moins de cinquante ans, pourront passer des messages publicitaires pendant les pauses.

Les perspectives

ADELI propose de créer une filiale intitulée Fédération Internationale de Face-à-face des AGL, chargée de gérer l'organisation de cette compétition.
Nous sommes certains qu'aucun journal, aucune radio, aucune télévision ne prendra le risque de se priver des retombées publicitaires de cet événement de portée mondiale. ▲

Alain Coulon