T E

Les nouvelles formes de programmation

Allouch Bachir & Trantoul Gilles

juin 2003

Plan

Introduction
Aspect Oriented Programming
Intention Programming
Generative Programming
Composition Filter Programming
Intentsoft
Conclusion

INTRODUCTION

Le développeur d'applications informatiques inscrit sa réflexion dans des modèles théoriques de pensée appelés paradigmes. Il la met en œuvre à l'aide d'outils logiciels qui mobilisent des paradigmes de référence (langage de programmation, langage de modélisation). En tant que concepteur, son activité de création d'applications se situe ainsi dans une approche à la fois philosophique et scientifique.

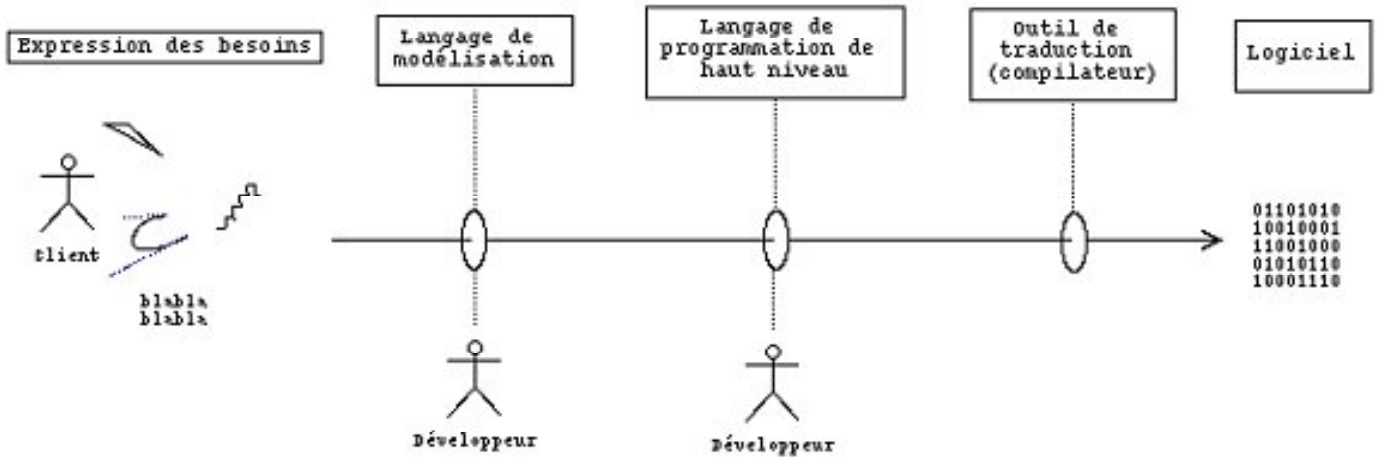
Paradigme :
"Modèle théorique de pensée qui oriente la recherche et la réflexion scientifiques" (Larousse).
"Outil de pensée" (A. Comte-Sponville).
"Un outil de pensée qui oriente la façon d'analyser,
de concevoir et de coder un programme"(G. Caplat).

Un ordinateur ne connaît que le langage binaire (la présence ou non d'un signal). Les pionniers programmaient en langage machine, un exercice d'une grande difficulté car il existe un fossé énorme entre l'expression des besoins en langage naturel et une interprétation possible en langage machine. De plus, il passait plus de temps à configurer et trouver l'instruction machine optimale à utiliser que pour résoudre et coder le problème lui-même.

Afin de gagner en productivité, les informaticiens ont créé et commercialisé des langages asymboliques (langages assembleurs) dans le but de simplifier la programmation de séquences de code récurrentes (opérations arithmétiques, saut, retour, comparaison, ...).

Les besoins informatiques évoluent : ce système rencontra très vite ses limites. Les informaticiens ayant repéré des solutions-types associées à des classes de problèmes connus, décidèrent d'implémenter ces modèles de solutions et de les proposer, clés en main, sous forme d'instructions de programmation de "haut niveau".

L'ensemble de ces instructions constitue un langage de programmation évolué. Parmi les premiers, citons FORTRAN (Formula translation en 1957) pour les applications scientifiques et COBOL (COammon Business Oriented Language en 1960) pour les applications de gestion.



tirée de Economie & gestion : n°16 : Mars 2003

Plus que jamais, il devient nécessaire de structurer le cycle de vie d'un projet de développement en passant d'une approche fondamentalement empirique à une approche résolument méthodique. Des méthodes fleurissent ainsi que des langages de modélisation (ordinogramme, schéma entité/association, graphe, UML -Unified Modeling Language).

La meilleure façon d'aborder un problème complexe consiste à le décomposer. Cependant, il n'existe pas un modèle unique de décomposition d'un tout en ses parties. Les paradigmes de décomposition ont influencé de nombreux langages et méthodes de développement.

Le paradigme de décomposition fonctionnelle (~1970) : Fondé sur une approche cartésienne des problèmes, il consiste à décomposer un programme en sous-programmes (fonctions), eux-mêmes décomposés en sous-programmes, etc. "... en faisant ainsi, vous échangez le confort à court terme pour une rigidité à long terme". (Bertrand Meyer)

Le paradigme de décomposition orientée donnée (~1980) : Dans ce paradigme, les données prévalent sur les fonctions. L'accent est mis sur la dimension statique du système, sur ce qu'est le système, et moins sur ce qu'il fait. Cette approche des systèmes est basée sur l'idée que les structures de données ne changent pas, contrairement aux traitements qui sont amenés à évoluer. "Dis-moi qui tu es, je te dirai ce que tu fais." Dans ce paradigme, les règles de gestion (traitements) sont éloignées des entités (données) qui ne jouent plus qu'un rôle passif.

Paradigme de décomposition orientée objet (~1990) : Victimes de leur succès, les programmes informatiques doivent répondre à l'accroissement continu du niveau de complexité des applications à développer : ils deviennent de plus en plus volumineux et complexes, donc difficiles à maintenir et surtout à faire évoluer au même rythme que les besoins des utilisateurs.

Comment ne pas "réinventer la roue" ? Comment réutiliser les programmes déjà développés plutôt que de les réécrire, bref, comment gérer la complexité ? Aidée par la montée en puissance des processeurs, la programmation objet a permis de répondre à ces questions. Avec des aobjets-briques l'informaticien construit d'autres objets qui, à leur tour, serviront de base pour la construction d'autres objets, …. jusqu'à constituer une application complète : un programme est constitué d'objets collaborant entre eux par envois de messages.

L'ancêtre des langages de programmation objet est Simula (1967). Smalltalk est le plus célèbre des langages objet des années 1970 (Alan Kay, Rank-Xerox - Palo Alto), puis viendront Eiffel (années 1980, Bertrand Meyer), C++ (années 1980, Bjarne Stroustrup, AT&T Bell), Java (années 1990, Sun) et C# (années 2000, Microsoft).

Tout l'art de l'analyse et de la conception objet consiste à identifier les objets pertinents et à faire en sorte qu'ils collaborent, en s'assurant de leur capacité à être réutilisés dans d'autres contextes applicatifs. Pour illustrer, nous pourrions dire que cette approche est basée sur le principe suivant : "Dis-moi ce que tu fais, je te dirai qui tu es".

Du coup, on peut dire que les paradigmes de programmation ont beaucoup évolué depuis l'émergence de l'informatique, en laissant derrière eux leurs représentants, c'est-à-dire les langages de programmation. Plusieurs langages se sont ainsi succédés, chacun ayant sa propre vision de la programmation et répondant à une attente particulière des programmeurs.

L'évolution de la technologie de programmation a amélioré la capacité des developpeur à pouvoir clairement séparer les préoccupations, et donc la capacité d'identifier, encapsuler, et manipuler uniquement les fragements du logiciel qui sont indispensables pour un concept, but ou finalité particulière.

La question la plus pertinente que l'on peut se poser en lisant le titre de ce TE est pourquoi a-t-on besoin d'envisager de nouvelles formes de programmation, alors que la POO (forme la plus modulaire des paradigmes actuelles, et dont le langage applicatif Java est l' un des plus prisé, et dont les notions maîtres seriont : héritage, interface et polymorphisme) donne une bonne flexibilté de réutilisation aux programmeurs.

Le soucis de performance et d'efficacité a poussé les chercheurs à développer de nouveaux paradigmes, dont le but premier est de programmer plus vite, et plus simplement pour répondre aux attentes du marché informatique.

Les gros travaux informatiques, appelés projets, regroupent sous une même optique des dizaines de programmeurs, et il devient nécéssaire de simplifier le travail en parallèle ,la réutilisabilité du code et le code lui-même (passage par des langages spécifiques plus abstrait). Toutes les nouvelles formes de programmation que nous allons voir dans cet exposé s'appuient sur un principe de réutilisabilité maximum, et ont pour ambition de changer la manière de programmer.

La POO a été une évolution importante dans l'approche d'une véritable modularité, mais elle est limitée au niveau de ces possiblités. En effet, les objets ne semblent être qu'une version améliorée des structures du langage C,et l'apport en modularité de sa configuration dynamique ne justifie pas ses faibles performances, mais comme nous allons le voir elle est un support précieux.

Réaliser un logiciel capable de survivre à l'évolution des besoins de ses utilisateurs n'est pas un exercice simple ! Un logiciel efficace et flexible, c'est-à-dire qui accepte les changements sans, ou avec très peu de modifications de code, doit abstraire certaines de ses parties. Cette pratique augmente le nombre de couches et donc la complexité du système, qu'il convient de limiter. Il s'agit donc de concilier flexibilité et simplicité de conception.

L'étude des techniques de développement classiques comme les méthodes itératives font apparaître que le problème de l'informatique est un problème organisationnel lié à un entrelacement des sous-problèmes techniques ou fonctionnels d'une application.

Depuis des années, l'interêt de la communauté scientifique pour ce problème s'est accru de manière considérable et de nombreuses tendances et techniques visant à le résoudre on vu le jour, parmis les principales on peut citer : La programmation orientée-aspect de Xerox PARC, La programmation orientée-sujet de IBM, et la programmation par intention de Microsoft.

Toutes ces approches visent à implémenter le principe de la séparation des préoccupations, et plutôt que d'attaquer le développement d'une application du front, ces approches suggèrent de concevoir des composants logiciels indépendants, et fournissent des moyens pour les assembler.

Le paradigme orientée-objet ne peut compacter et localiser proprement qu'une seule préoccupation. Toutes les autres préoccupations ne peucent pas être encapsuler dans les modules dominants, ce qui aura pour résultat d'avoir du code dispersé à travers les modules avec des appels croisés d'un module à l'autre.

Ce phénomène est appelé "La tyranie de la décomposition dominante", dans laquelle une façon dominante de décomposition impose des structures logicielles qui rendent la séparation des préoccupations difficile voire impossible. Ce problème est difficilement évitable dans plusieurs systèmes conçus avec le paradigme orienté-objet.

Et à l'aube du IIIe millénaire, la nécéssité de telle nouvelles formes de programation se montrent plutôt évidente, avec des applications de plus en plus complexes(des objets distants...), une collaboration entre objets de plus en plus riche, de concepts qui augmentent (remote, synchronized...), des applications que l'ont veut évolutives et adaptables, et des facilités d'intégration des composants existants...

On peut ajouter aussi, la méta-programmation qui, bien qu'étant une solution possible et une technique puissante, elle reste résérvée à des programmeurs avancés. On se retrouve avec des base structurelle qui ne devait pas être altérnée, et des programmes objets pollués.

Les méthodologies de programmation et les langages définissent la façon dont on communique avec les machines; chaque methodologie présente une nouvelle façon de décomposition du problème, elle nous permet plus naturellement de trnasformer les exigences du système en structure de programmation. L'évolution de ces méthodologie nous authorise à créer des systèmes de plus en plus complexes, car finalement, elle nous permet de gérer plutôt efficacement de telles compléxitées...

Nous allons maintenant présenter les nouvelles formes de programmation les plus promettantes.

AOP : La Programmation Orientée-aspect

Présentation de L'AOP

Depuis 1997, la programmation orientée aspect(Aspect-oriented programming) provoque l'engouement de la communauté scientifique s'interessant au génie logiciel. Cette technique offre des nouvelles perspectives à un principe bien connu en génie logiciel : le séparation des préoccupations. Aujourd'hui, et grâce à des outils robustes et éprouvés, l'AOP interesse de plus en plus de services informatiques de grands groupes industriels... Elle pourrait bien devenir la prochaine révolution dans le monde du développement logiciel.

En effet, en étendant la programmation orientée-objet (OOP) avec la notion d'aspect, elle peut rendre modulaire des problèmes transverses tels que la mise en place d'un mécanisme de log ou la gestion des droits. Elle permet d'atteindre une parfaite modularité des programmes informatiques complexes... Ce nouveau mode de programmation est souvent présenté comme une évolution logique de la programmation objet...

"L'AOP, implémentée correctement, ne servira pas uniquement à nettoyer le code, mais aura un grand impact sur tout le processus de développement de logiciel. En fait, la MIT Technology Review [Jan 2001] a mis l'AOP parmis les 10 meilleure technologies émérgentes qui changeront le monde... "

L'AOP théorique

introduction

Les processus de concéption des logiciels et les languages de programmations ont toujours vécu dans une relation de compatibilité mutuelle. Les processus de concéption divise le système en morceaux de plus en plus petits, et les languages de programmation nous fournissent des méchanismes pour que le programmeur puisse construire des abstractions des sous-unités du système afin de les composer de divers manières pour aboutir à un systeme global.

Un processus de concéption et un language de programmation travaille correctement ensemble quand le language fournit des méchanismes d'abstractions et de compositions qui colle proprement avec les genres d'unitées créees lors de la décomposition du système par le processus de conception.

De ce point de vue, plusieurs languages de programmation (dont la Programmation orientée-objet, la procédurale et la fonctionnelle) ont une racine commune dans leur méchanisme de composition et d'abstraction qu'est la procédure généralisée.

Les méthodes de concéptions qui ont évolué autour des languages à base de procédure généralisée (les GP) décomposent le systeme en unités de fonctions ou comportement. C'est la décompostion fonctionnelle (où les unités sont encapsulées dans une procédure, une fonction ou un objet).

Diverses méthodes d'implémentations

Pour implémenter une application réelle, on a le choix entre 3 façons de procéder :

- une implémentation aisement compréhensible mais inéfficace ...
- une implémentation incompréhensible mais efficace ...
- une implémentation AOP aisement compréhensible et éfficace ...

Pour montrer concrètement la différence entre ces 3 méthodes, leurs défauts et leurs qualités, on va se placer dans un contexte de traitement d'image bicolore (passage d'une image via plusieurs filtres), et donc on définit nos contraintes ou éxigences minimales de l'implémentation des algorithmes.

La premiere contrainte a réspecter est naturellement la facilité du développement et de maintenance de notre programme. Car cela nous permet de developper rapidement des programmes sans bugs et d'y faire d'éventuelles modifications.
La deuxieme est une gestion éfficace de la mémoire. Car on manipule souvent des images plus ou moins grandes et il faut, par mesure d'éfficacite, minimiser le nombre de références mémoires et de la taille globale de stokage.

Réspecter la permière contrainte est plutôt facile, les languages de programmations traditionnelles font largement l'affaire. Il suffit d'implementer les filtres sous formes de procédures (au sens large), implémenter les procédures de bases qui manipulent directement les pixels, et ensuite bâtir des procédures de haut niveau en dessus des procédures de bases.

Une permiere implementation "élégante" du OU entre 2 images sera :

```
procedure OU (image a, image b) {
resultat <- creerNouvelImage( )
pour i de 1 a longueur faire
        pour j de 1 a largeur faire
                resultat [ i ] [ j ]  <- a [ i ] [ j ] OR b [ i ] [ j ]
        finpour
finpour
return resultat
}
```

et d'une facon similaire, on peut aussi définir les autres procédures comme :
proc : ENLEVER(a, b) == ET(a, NON(b))
proc : PREMLIGNE(a) == ENLEVER(a, BAS(a))
proc : DERNLIGNE(a) == ENLEVER(a, HAUT(a))
proc : EXTREMITE(a) == OU(PREMLIGNE(a), DERNLIGNE(a))

on a donc des procédures qui traitent directement les pixels dans des itérations, et puis des procédures utilisant ces traitants de bas-niveau, ce qui nous fournit un code facile à lire, comprendre, débogger, étendre, argumenter etc... ce qui respecte donc la première contrainte, mais pas le deuxième !! car à chaque fois qu'on appelle une procédure, elle réutilise les itérations de bases, et recrée une image dont la nécéssite est vraiment temporaire, ce qui aboutit à des créations et appels excessives à des références et allocations mémoires, ce qui nuit a la performance.

La solution commune est plûtot d'avoir une vision globale du programme et de minimiser le nombre de fonctions intermédiaires appelées.

```
procedure EXTREMITE(a) {
result <- creerNouvelImage( )
a_haut <- HAUT( a )
a_bas <- BAS( a )


pour i de 1 a longueur faire
        pour j de 1 a largeur faire
                result [ i ] [ j ] <- OR( AND ( a[ i ][ j ] , NOT( a_haut[ i ][ j ] )),
                                          AND ( a[ i ][ j ] , NOT( a_bas [ i ][ j ] )))
        finpour
finpour
return result
}
```

comparée a l'originale, cette solution est entrelacée !!! Dans un exemple concret de système optique de reconnaissance des caractères OCR, l'implémentation "élégante" et "propre" du programme (comme la première version) n'éxcedait pas les 768 lignes, tandis que l'implémentation plûtot "efficace" et entrelacée passait à 35213 lignes !!!

Le code entrelacé a le malheur d'être extrêmement difficile à maintenir, car un tout petit changement exige un effort mental non négligeable de décryptage du code, avant de le modifier à plusieurs endroit pour le recrypter à la fin.

Dans cet exemple, la fonction de base reste dans la décomposition traditionnelle hiérarchique, tandis que la fusion des boucles est formée en fusionnant les boucles des divers filtres voisines dans le graphe du flux de données des images. Et ce sont les appels mutuels entre ces fonctions et boucles qui seront responsables de l'entrelaçement du code. Car finallement, le seul méchanisme de décomposition fourni par le langage est l'appel des procédures, qui est très adapté pour générer des unités fonctionnelles non-optimisées, mais pas de combiner des appels ou boucles de façons efficaces, d'où l'obligation de le faire soi-même.

Aspect et composant

On appele une unité de base un composant si elle peut être encapsulée proprement dans une procédure généralisée, et un aspect sinon. Et du coup, le but de l'AOP devient de séparer les composants des aspects et d'eux mêmes, en nous fournissant un méchanisme d'abstraction et composition pour en faire un système globale.

Le premier changement dans une implémentation AOP de ce problème sera la nature des filtres qui ne sont plus des procédures, le deuxième au niveau des boucles d'itération qu'on a interêt à leur accordé une structure la plus explicite possible.

```
(define-filter or! (a a)
        (pixelwise (a b) (aa bb) (or aa bb) )
```

où pixelwise est finalement un itérateur, et on peut donc introduire des boucles de haut-niveau qui aura un effet critique sur la détéction, l'analyse et la fusion des boucles.

```
(cond ( (and (eq (loop-shape node) 'pointwise)
             (eq (loop-shape input) 'pointwise)
        (fuse loop input 'pointwise  ... ) ) ) )
```

Et puis, il y'aura le passage du tisseur (weaver) qui aura le graphe de flux de données, et qui essayera de fusionner les boucles tant que possible, donc plusieurs boucles du genre

```
(pointwise (#<edge1 #<edge2) (i1 i2) (or i1 i2) )
```

se fusionneront en :

```
(pointwise (#<edge1> #<edge2> #<edge3>) (i1 i2 i3)
        (or (and (not i1) i2) (and (not i3) i2) ) )
```

une structure qui sera aisément manipulable et modifiable plus tard; avant de générer du code C par exemple incluant la fusion des boucles. A noter que le rôle du tisseur est une intégration, plutôt que de l'inspiration. Du coup, on baisse le code de l'exemple du OCR de 35213 lignes à 1039 lignes + 3520 lignes (dont 1959 pour le tisseur seul).

Une méthode plutôt empirique et vague pour estimer le degré de gain en orientant AOP :

redutcion = (taille du code entrelaçé - taille du programme des composants)
/ (somme des tailles des programmes aspects) .

L'AOP pratique

Motivations de L'AOP

L'une des préoccupations majeurs de la programmation a toujours été la nécessite d'améliorer la clairete, fléxibilite et le réutilisabilité du code écrit ; les paradigmes étaient donc :

- les macros : re-utilisabilié du code automatisée
- la programmation structurée : utilisation d'une boîte noire pour pouvoir prouver la validité d'un code : décomposition modulaire et des structures à point d'entrée/sortie unique.
- la programmation orientée-objet : réutilisation d'une boîte noire d'un mélange de code et de structures de données en ajoutat des types abstraits de données, des classes, des interfaces et des paquetages.
- Mais, qu'est ce qui manque ???

Exemples de notions non-modularisables

Considérons un server web, il contient :

- un ensemble de classes qui fournissent et utilisent le parsing HTML
- un ensemble de classes qui fournissent et utilisent le réseau
- un ensemble de classes qui fournissent le gestionnaire des logs (logging)
- ET le quasi-totalite des classes qui utilisent le logging ...

Et là, on est face à un probleme non-modularisable dans la pensée POO, un probleme qui nous poussent à generer excessivement du code dupliqué, un code qui sera difficile à comprendre (manque de structure explicite oblige : et donc, à grande échelle ne ressemble à rien), à développer ou à changer (car il faudra localiser partout le code correspondant, le comprendre et le modifier, et donc avoir des grandes risques de se tromper...).

Le seul moyen de mettre en oeuvre un module de log en POO est de l'appeler... L'appeler de partout... La dépendance devient donc inévitable....

- L'AOP : réutilisation des aspects qui entrelacent des familles entières de classes : l'aspect est un champ de préoccupation appliquable à des types de données multiples dans une application.

Tous les paradigmes ont élargis les applications de la modularité et des unités-fondamentales du code et sa réutilisabilité ; L'AOP évolue simplement ces concepts au niveau logique supérieur, en manipulant aussi des aspects au lieu de se contenter des objets comme l'unité de la réutilisabilité.

Ou bien, si vous voulez, par exemple, utiliser une librairie de classes crée par quelqu'un d'autre, qui contient une option que vous n'appreciez pas, comment faire pour la désactiver si le code correpondant se trouvaient un peu partout dans les divers classes?

Ou même, que cette librairie ne contenait pas une partie dont vous avez absolument besoin (comme être thread-safe) et que vous ne pouvez pas l'ajouter à moins de changer toutes les autres classes ? et si par malheur, la librairie était ramené à évoluer, il faudrait proceder à de nombreuses modifications, reparties sur la quasi-totalite du code ...

L'AOP nous donne la possibilité de créer des familles de classes qui ont un interet unique (se concentrer sur leur champ de préoccupation), et elle sont proprement combinables avec les autres familles gérantes d'autres préoccupations...

Problèmes constatés

Tout est parti d'un constat simple, quel que soit le mode de développement que nous adoptions, nous ne parvenons jamais à ne pas réécrire de code redondant. Même avec des développeurs excéptionnels, avec toute sorte de programmation (fonctionnelle, procédurale ou orientée objet, etc...), il existera toujours des une situation danslaquelle il faudra écrire des choses répétitives pour gérer certains besoins récurrents. (ex: faire appel à un framework de log de nombreuses methodes.) Cette problematique concerne bien aussi d'autres exemples comme la sauvegarde des données, le contrôle des droits d'acces, le logging etc... La modularisation n'est donc pas parfaite.

Les problemes constatés :

- du code redondant : on retrouve des fragements identiques ou presque un peu partout
- difficulté pour raisonner sur le code : absence de structure explicite, et un code qui, vu de loin, ne ressemble absolument à rien.
- difficulté à changer, réutiliser, maintenir et évoluer le code.
- un certain éloignement du logiciel optimal : qui, comme l'or est toujours maléable et fléxible... "Harold Ossher and Perit Tarr / les logiciels porphogeniques"

ce qui entraînera, bien sûr, une faible tracibilité et productivité car le développeur se concentre plutôt sur la gestion des divers préoccupations entrelaçées que sur le problème essentiel.

Les solutions actuelles

Vu que les préoccupations croisées existent dans tous les systèmes, ce n'est pas étonnant de savoir qu'on a dû engendré quelques techniques pour essayer de modulariser leur implémentation : on cite xdoclet, les classes mix-in, desgin patterns et les solutions spécifiques à un domaine donné. Mais une bonne architecture d'un système prendra en considération des exigences évetuelles ou futures du système, et cela n'est pas évident du tout, à défaut de quoi, il faudra tout recommencer ou changer, ainsi qu'une vision un peut trop futuriste conduira à un programme confu et abusivement concu.

En gros, l'architecte ne sait que très rarement toutes les proéoccupations possibles du système à manipuler, même pour les exigences pré-définies, leurs spécifités nécessaires pour l'implémentation du système peuvent ne pas être tout à fais disponible et adéquates, d'où le dilemme du sous/sur conception auquel fera face l'architecte.

XDoclet, lui, c'est un peu de la AOP "light", au sens ou il te permet effectivement de gérer plus facilement certains aspects de ton code (par le biais de sa syntaxe déclarative). Cependant, à mon sens (Nicolas Delsaux : linuxfr.org), l'interêt futur le plus important de la AOP, c'est de pouvoir changer l'aspect utilise durant le fonctionnement du programme, càd par exemple faire évoluer dynamiquement les règles de sécurite d'un programme.

Alors AOP c'est quoi?

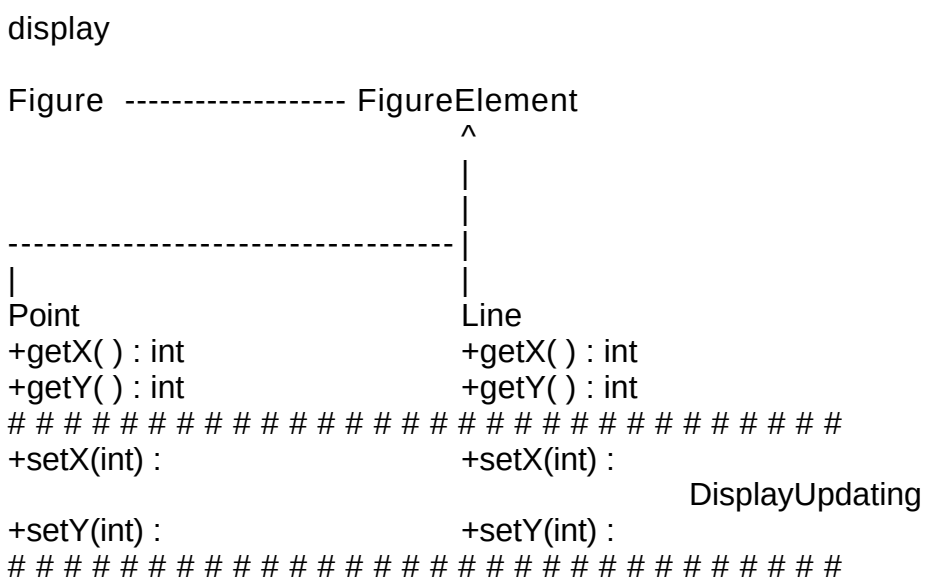
L'AOP est une nouvelle technique de programmation qui permet aux programmeurs de modulariser les préoccupations croisées. L'AOP introduit les *aspects*, qui encapsulent les comportements affféctants plusieurs classes dans des modules réutilisables.

C'est une méthode de réécriture automatique du code pour ajouter ou changer des fonctionnalités. La puissance de l'Aop se situe dans sa capacite d'identifier des modèles dans le code préexistant et de les changer dans plusieurs endroits avec le travail minimal du programmeur, sans changer le code source du programme original.

En tant que programmeur aspect, vous indiquez simplement où (et dans quels cas) vous voulez que les changements se produisent, et indiquez alors quelles mesures vous voulez prendre. Le compilateur d'aspect tisse alors ces changements dans le code compilé. Le code original n'a pas besoin de savoir les fonctionnalités que l'aspect va ajouter, et une simple recompilation sans aspect permet de ravoir les fonctionnalités originales.

Des aspects peuvent être appliqués à un certain nombre de classes, et peuvent donc ajouter des possibilités sans forcer la classe à implmenter ou étendre une autre classe . Naturellement, le module de la programmation orientée d'aspect sont des aspects, qui sont des structures de code tout comme des classes dans la programmation orientée objet.

De la OOP à l'AOP



-un petit exemple de réecriture inutil du code de setX()/setY() et un update() après chacune

L'AOP se base sur le fait que les programmes typiques montrent souvent des comportements qui peuvent pas être integrés dans un module unique du programme d'une façon naturelle, ou même dans divers modules de programmes étroitement liés. Dans la OOP, l'unité naturelle de modularisation est la classe, et une préoccupation croisée est une préoccupation s'éténdant sur plusieurs classes. Des exemples typiques de tels préoccupations sont : L'historique des actions (logging), la gestion d'erreur liée au contexte, l'optimisation de la performance, et les design patterns.

Pour les codes contenants des préoccupations croisées, le diagnostic est simple : un important manque de modularisation. Comme l'implementation de ces comportements croisés sera forcement dispersé, les développeurs auront du mal à la comprendre, l'étendre ou de la changer.

L'AOP est complémentaire à la OOP dans le sens où elle modélise tout ces préoccupations qui ne peuvent l'être par ce dernier dans une seule unité appelée *aspect,* des aspects qui vont être facile à manipuler, changer, insérer, supprimer à l'instant de la compilation, ou même reutilisés.

L'AOP fournit une nouvelle façon de séparer entre la conception des logiciels et leurs implementations. La conception et programmation orientées objets ont été ,indisputablement, un grand succès. Elles ont ,par exemple, améliorés le développement des logiciels en nous permettant d'encapsuler "proprement" des unités de fonctionnalités de plusieurs échelles -des structures simples de données aux outils GUI jusqu'aux serveurs des réseaux.- dans des classes bien définies

Par contre, l'OOP perd son utilité devant des préoccupations systématiques comme la synchronisation, partage des resources, distribution, gestion de la mémoire etc... car l'OOP tend plutôt à entrelacer la classe du système et la structure du module. L'AOP intervient à ce niveau là en implementant ces préoccupations systématiques sous forme d'aspects qui contrôleront les classes qui implementent les fonctionnalités basiques, d'où une image claire de la perspective du contrôle du système.

L'AOP diffère grandement de la OOP dans la manière dont elle gère les recoupements. Avec l'AOP, chaque unité implémentée (aspect) reste inconsciente du fait que les autres unités peuvent l'instancier en tant qu'aspect, et ne sait pas ,par exemple, qu'il existe éventuellement d'autres unités qui archivent ou authentifient ses opérations. Cela représente un très fort changement de concept par rapport à la OOP.

Une implémentation AOP peut employer d'autre méthodologie de programmation en tant que méthodologie de base, ce qui permet de cumuler les bénéfices des deux aproches. On peut donc adopter la OOP comme système de base, et on aura une implémentation AOP jouissante des avantages OOP, comme l'utilisation d'un language procédural comme base pour un language orientée-objet (C et C++ par exemple).

AOP mieux que OOP?

A la question, "qu'est ce que je peux faire en AOP mais pas en POO?" la réponse est catégorique : absolument rien ! Mais bon, la POO ne te permet pas de faire du code que tu sera incapable de générer avec de la programmation structurée, et cette derniere du code que tu sera incapable de générer avec du code spaghetti ou code assembleur .... *Si vous travaillez suffisemment dur !*

Par contre, l'AOP nous élagrit la possibilité de :

- la réutilisabilite : en séparant les abstractions du problème et son implémentation, tous les deux seront réutilisables.
- la modularisation et le compactage des préoccupations : le code est plus facile à lire, comprendre et développer, car le code gérant un certain domaine de préoccupation est regroupé dans une seule unité....
- la fléxibilite : la capacité de créer et "tisser" des aspects complétements nouveaux pour accomplir divers missions et donc avoir divers résultats.
- la possibilité de "composer" des morceaux de code, par exemple une équipe travaille sur un logiciel, et une deuxième équipe travaille sur un autre "aspect" qui sera ensuite composé pour former le logiciel final.

Définition de l'AOP

La définition originale de l'AOP est la suivante : "la programmation par aspect est une technique novatrice permettant de mettre en facteur certaines responsabilités dont la réalisation est à priori dispersée à travers un systeme, fût-il orienté objet."

En gros, l'AOP se décompose en 3 parties majeurs :

- la décomposition aspectuelle : où on décompose les exigences afin d'identifier clairement les préoccupations communes et croisées (ceux des modulesde ceux du système...
- implémentation des préoccupations : où on implémente individuellement chaque préoccupation.
- recomposition aspectuelle : c'est la phase de recomposition, on tisse ensemble les aspects et les fichiers sources métiers... votre code est prêt à être exécuté avec la commande 'java' !!!

Définition des Aspects

Aspect : ~ ad specere (lat.) -> regarder/ apparance particulière à l'oeil ou l'esprit

On peut voir les aspects comme étant un indicateur des décisions à choisir à propos de l'implémentation d'un programme. Dans tous les paradigmes pre-AOP, ça a été les décisions qu'on doit prendre avant de commencer à coder, pour la simple raison qu'est que le code implémentant ces décisions sera emmêlé dans tous les autres codes.

Les aspects sont particulierement utiles dans deux cas:

Le premier est quand vous avez les classes multiples et indépendantes auxquelles vous voulez ajouter une fonction commune . Ca sera impossible de trouver une solution orientée objet à un tel problème, vu la nature indépendante des classes soumises. Au lieu d'essayer soigneusement de trouver une solution en ajoutant des classes supplémentaires(héritage ou interface) , les aspects peuvent entre-croiser les classes qui doivent être changées et les re-cousent dans une mesure qui devrait être prise.

Le deuxieme cas se produit quand vous voulez fournir un certain nombre de dispositifs dans une classe, et permet à chaque dispositif d'être activé ou désactivé au moment de la compilation. En plus, changer la configuration des dispositifs éxige simplement une recompilation après avoir changé les divers aspects à appliquer.

Les gains pratiques de l'AOP

- interactions minimisées : les aspects sont développés individuellement et ne sont pas nécessaires au fonctionnement du programme entier.
- Evolutivité : on peut donc aisement ajouter ou configurer des aspects faites sur le champ sur le programme en cours d'éxecution. Si un aspect est déprecié, on peut le remplacer par une nouvelle version, sans rien changer au programme.
- Planning du développement aisé : l'équipe de développement peut se concentrer pleinement sur le programmation métier (les composants ou objets) ou la configuration des aspects selon un planning prévisionnel simplifié par l'indépendance totale du code métier et des aspects.
- Réutilisation du code : on peut par exemple tisser des aspects (AOP) sur un programme concu en objets (POO), le code purement métier est allégé (les aspects sont codes séparement).

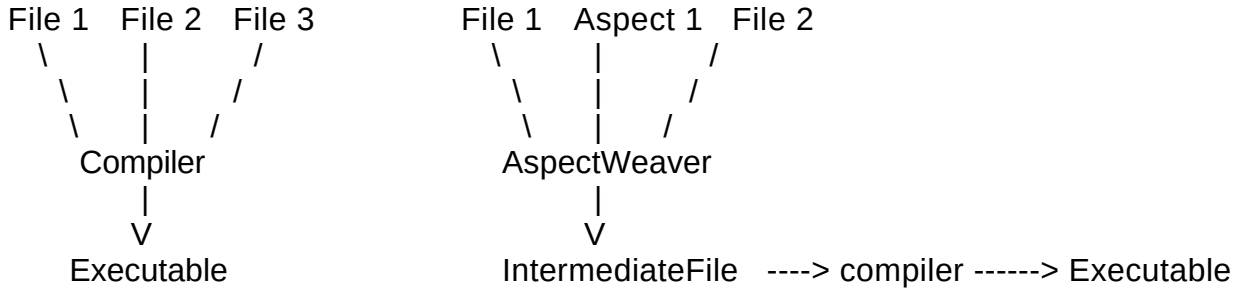
Implémentation de l'AOP

Pour réussir une implementation AOP, on a besoin de 5 éléments de base :

1) un language pour programmer les composents.
2) un ou plusieurs languages pour programmer les aspects.
3) un tisseur d'aspects pour les languages combinés.
4) un programme pour implémenter les composants avec le language 1)
5) un ou polusieurs programmes pour impémenter les aspects avec le(s) language(s) 2)

C'est pourquoi, à la phase de conception d'un système , on doit choisir ce qui va être gérer par le language composant et ce qui va l'être par le language aspect. Le language composant doit s'assurer que des programmes composants adéquates peuvent être codés pour implémenter les caractéristiques basiques du système. De plus, ils ne doivent pas se mêler avec ce dont les programmes aspects ont besoin de contrôler.

D'autre part, les languages aspects doivent proposer une implémentation adéquate des aspects voulus pour contrôler le système. Bien que c'est évident que les languages composants et aspects auront des méchanismes d'abstraction et de composition différentes, elles ont besoin d'avoir des termes communs. Ce sont ces termes communs qui permettent au tisseur aspect de combiner le tout pour avoir l'opération système totale voulue.



AspectJ

AspectJ est un language et un compilateur crées pour fusionner les capacités de l'AOP avec Java, une langue de programmation POO de haut niveau.

Où et comment déclarer un aspect?

Des aspects pour AspectJ sont déclarés dans des fichiers .aj. Comme Java, les aspects peuvent être définis dans une classe. A la différence de Java, l'aspect ne doit pas avoir le même nom que la classe qui le contient ni celle que la classe Java auquel il sera appliqué. Les aspects sont déclares comme les classes:

```
aspect monAspect  {  }
```

Les divers parties d'un aspect

Il y a trois parties principales aux aspects dans AspectJ. Ensemble, ils définissent essentiellement les points où l'aspect doit agir, et le code à executer quand chaque point est atteint. JoinPoint, PointCut et les Advices...

Qu'est ce qu'un point de jonction?

Un point de jonction est un point pendant l'éxecution d'un programme Java ou AspectJ vous permetant d'agir. (là où on veut ajouter du code par exemple (System.out.println(...)).

Les points de jonctions sont déjà prédéfinis par AspectJ. Quand on écrit un aspect, on doit indiquer le point de jonctions surlequel on veut agir en définissant un pointCut.

Où est ce qu'on retrouve les points de jonctions dans AspectJ?

Hormis des points de jonctions spécialement compliqués, on retouve généralement les points de jonctions lors des :

- L'appel des méthodes
- l'éxécution des méthodes
- la création des instances (appels de méthodes ou constructeurs)
- l'éxécution des constructeurs
- les références des champs

Qu'est ce qu'on pointCut ?

C'est un ensemble de points de jonctions correlés (ensemble des endroits de mélange de code concernant la même fonctionnalité). Un pointCut est une référence à un point de jonction particulier ou un groupe de points de jonctions dans un programme où on souhaite agir. On définit un pointCut dans notre aspect de cette manière:

```
pointcut machinCalled( ) : (call (public void Machin.bidule( )));
```

Un pointCut se compose de deux parties, séparées par un deux-points. On met dans la partie gauche le pointcut qui sera défini par un nom, et éventuellement une liste de parametres. Dans la partie droite indique les points de jonctions que le pointcut doit considérer. Dans l'exemple en dessus, on déclare vouloir faire un quelquechose chaque fois la méthode bidule de la classe Machin est appellée. On peut aussi indiquer des méthodes comme parties d'interfaces.

```
call (public void FooInterface.foo( ))
```

L'exemple se rapportera à toutes les classes qui implémente l'interface FooInterface.

Un pointCut peut contenir plusieurs points de jonctions grâce aux opérateurs logiques &&, | |, et !. On peut donc ajouter :

| | (call (public void Foo.bar( ))) à la ligne ci-dessus si nous voulions éffectuer le boulot chaque fois que Machin.bidule ou Foo.bar est appellé.

Les pointCuts peuvent être tres spécifiques ou très générales. (on peut par exemple y inclure l'appel de toutes les méthodes publiques d'une ou de plusieurs classes).

L'Advice

Ce sont les actions à effectuer lors des points de jonctions (joinPoints). L'advice est la mesure que vous voulez prendre quand un pointCut est atteint. Il y a plusieurs types de advices, chacun vous permet de changer le code d'une certaine manière. Deux des types les plus simples, sont Avant et Apres un Advice.

Supposons qu'on veuille imprimer un message à chaque fois que la méthode foo est appellée pour des buts de déboggage, mais nous ne voulons pas gaspiller le temps ou la beauté du code en ajoutant des if(DEBUG) partout. La pointCut définie ci-dessus dans un aspect, et en ajoutant un Advice on peut réaliser la fonctionnalité désirée.

```
before ( ) : fooCalled ( ) {

        System.out.println("Foo est appelee.");

}
```

Les effets de bord

A noter que plusieurs aspects peuvent être stoqués dans un même point de jonction, et du coup, on pourrait avoir des interactions indésirables si un aspect a un effet de bord sur un autre; vu que les aspects ne communiquent pas vraiment entre eux ...

compilation

Un Weaving (tissage) est un processus systématique qui combine les aspects et les objets en utilisant un compilateur, pré-processeur ou un interpréteur. Le tissage peut s'éffectuer au moment de l'éxectuion (run-time) ou le compilation (compile-time).

Et donc, le tout sera compilé avec ajc (en lui passant les fichiers .java et les aspets .aj comme paramètres), le tout sera éxécuté avec la commande java et il nous reste l'avantage de bénéficier toujours du code source .java propre au cas ou on n'a plus l'intention d'afficher les messages de déboggage.

On peut aussi préparer des aspects adéquates pour ajouter des fonctions, étendre des classes etc...

Pour compiler avec AspectJ, on doit éxplicitement spécifier le nom des fichiers sources (les aspects et les classes) à inclure dans la compilation -- ajc ne cherche pas dans la classpath les divers fichiers à importer, comme le fait javac.

Là où les classes dans les OOPs représentent -et fonctionnent- comme des unités indépendantes, les programmes AOP doivent être compilés comme une seule unité plutôt qu'un fichier à la fois.

Toutefois, une limitation importante de la version courante de AspectJ est dûe au fait que son compilateur est incapable de tisser des aspects avec des classes précompilées (byte-code), une limitation qui disparaîtera avec la future version 2.0.

Un des plus importants outils remis avec AspectJ est un naviguateur graphique de structure qui nous révèle en un clin d'oeil la façon dont les aspects interagissent avec les autres composants du système.

Un petit comparatif AspectJ / AJC :

AspectJ	AJC
Extension du language JAVA	Framework AOP, serveur d'application
Nouvelle grammaire pour les aspects	Aspects écrits en JAVA pur
Utilise le code source, et chaque modification nécéssite une nouvelle compilation	Un byte-code permet l'ajout, la suppression et la modification dynamique des aspects
Ne gère pas les districutions	Distribue automatiquement les aspects sur des serveurs distants
Ne permet pas le développement d'aspects	Permet le développement d'aspects et/ou leur configuration à chaud
Atelier UML supportant les aspects	Un serveur + un atelier UML + une **IHM** de configuration. S'intègre à JBuilder, Forte et Emacs.
Pas d'aspects pré-développés	Bibliothèque d'aspects pré-développés configurables
Version 1.0.5	Version 0.8.1
Open Source Mozilla Public Licence	Disponible en Licence LGPL

JAC résoud les défauts d'AspectJ

- Ce n'est pas un nouveau language, et donc, pas de nouveau grammaire ou compilateur.
- Gestion dynamique des aspects, et leur modifications, ce qui parait primordial afin d'éviter de redémarrer le serveur après chaque changement.
- Il propose deux niveaux de services :
  - Un niveau de programmation (même niveau qu'AspectJ), résérvé aux programmeurs aspects, munis d'une certain compétence dans ce domaine.
  - Un niveau de configuration qui permet aux divers développeurs de configurer les aspects existantes afin qu'ils répondent aux exigences de leurs applications. Ce niveau ne demande aucune compétence orientée-aspect.

HyperJ

D'une façon générale, HyperJ vise de ne pas privilégier une déomposition au profit d'une autre, d'où n décompositions possibles au même niveau avec des relations entre les préoccupations. et puis, y'a **CLAW**, un prototype d'un tisseur d'aspect à l'éxecution...

Exemples de codes AOP

- Cet exemple trace le chemin d'éxecution du programme (via toutes les fonctions appelées).

```
aspect SimpleTracing {
        pointcut tracedCall( ) :
                call(void UneClasse.uneMethode(Parametre)) | |
                call(void AutreClasse.* ( . . ));
        before( ) : tracedCall( ) {
                System.out.println("On entre dans "+ thisJoinPoint);
        }
}
```

• Celui là aide à gérer les exceptions et éventuelles erreurs du programme.

```
aspect VerificationPointDeBord {
	pointcut setX(int x) :
	(call (void FigureElement.setXY(int,int)) && args(x, * ))
	| |  (call (void Point.setX(int)) && args(x));

	pointcut setY(int y) :
	(call (void FigureElement.setXY(int,int)) && args( * , y))
	| |  (call (void Point.setY(int)) && args(y));

	before (int x) : setX(x) {
		if( x < MIN_X | | x > MAX_X )
			throw new IllegalArgumentException("x : valeur invalide");
	}

	before (int y) : setY(y) {
		if( y < MIN_Y | | y > MAX_Y )
			throw new IllegalArgumentException("y : valeur invalide");
	}
}
```

• Une implémentation de moniteurs orientée-aspect.

```
aspect SignalerChangementMoniteur {
	private static boolean dirty = false;

	public static boolean testAndClear( ) {
		boolean result = dirty;
		dirty = false;
		return result;
	}

	pointcut move ( ) :
		call(void FigureElement.setXY(int, int)) | |
		call(void Line.setP1(Point)) | |
		call(void Line.setP2(Point)) | |
		call(void Point.setX(int)) | |
		call(void Point.setY(int)) ;

	after( ) returning: move ( ) {
		dirty = true;
	}
}
```

- Un exemple de redéfinition éventuelle de certaines fonctions durant l'éxécution

```
void around() : call( public void Hello.say( ) ) {
        if( Math.random( ) > .5 ) {
                proceed( );  // on ne fait rien
        }
        else {
                System.out.println( "Le sort ne vous sourit pas. ");
        }
}
```

- Un exemple d'ajout de méthodes et variables dans le code source via un aspect

```
public aspect TimeStamp {

  private long ValueObject.timestamp;

  public long ValueObject.getTimestamp(){
     return timestamp;
  }

  public void ValueObject.timestamp(){
     //"this" réfère à la classe ValueObject et non l'aspect Timestamp
     this.timestamp = System.currentTimeMillis();
  }
}
```

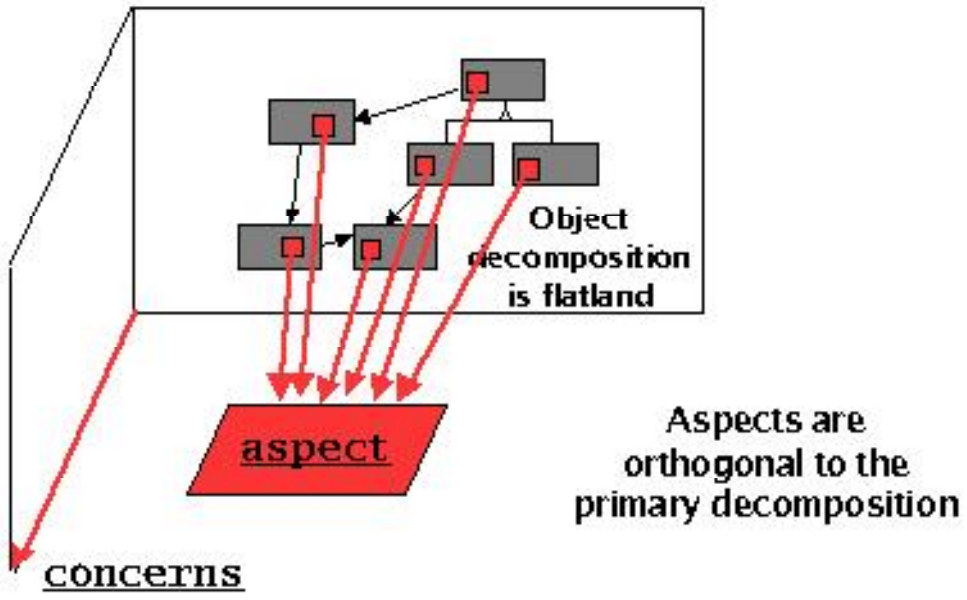
- exemple d'optimisation d'un programme : (passer Fibonacci en mode dynamique)

```
public class Fibonacci {
        public static int fib(int n) {
                if( n < 2 )
                    return 1;
                else
                    return fib(n-1) + fib(n-2);
        }

        public static void main(String[ ] args) {
                System.out.println("Fib(10) vaut "+ fib(10));
        }
}
```

```
public aspect Memorization {
	private HashMap cache = new HashMap( );

	private pointcut fibs(int in) :
		execution(int Fibonacci.fib(int)) && args(in);

	// On calcule uniquement si le résultat n'est pas déjà dans cache
	int around(int in) : fibs(in) {
		Integer result = (Integer) cache.get(new Integer(in));
		if( result == null ) {
			int f = proceed(in); // Non-trouvé, on calcule...
			cache.put(new Integer(in), new Integer(f));
			return f;
		}
		else
			return result.intValue( ); // Trouvé, OK !
	}
}
```

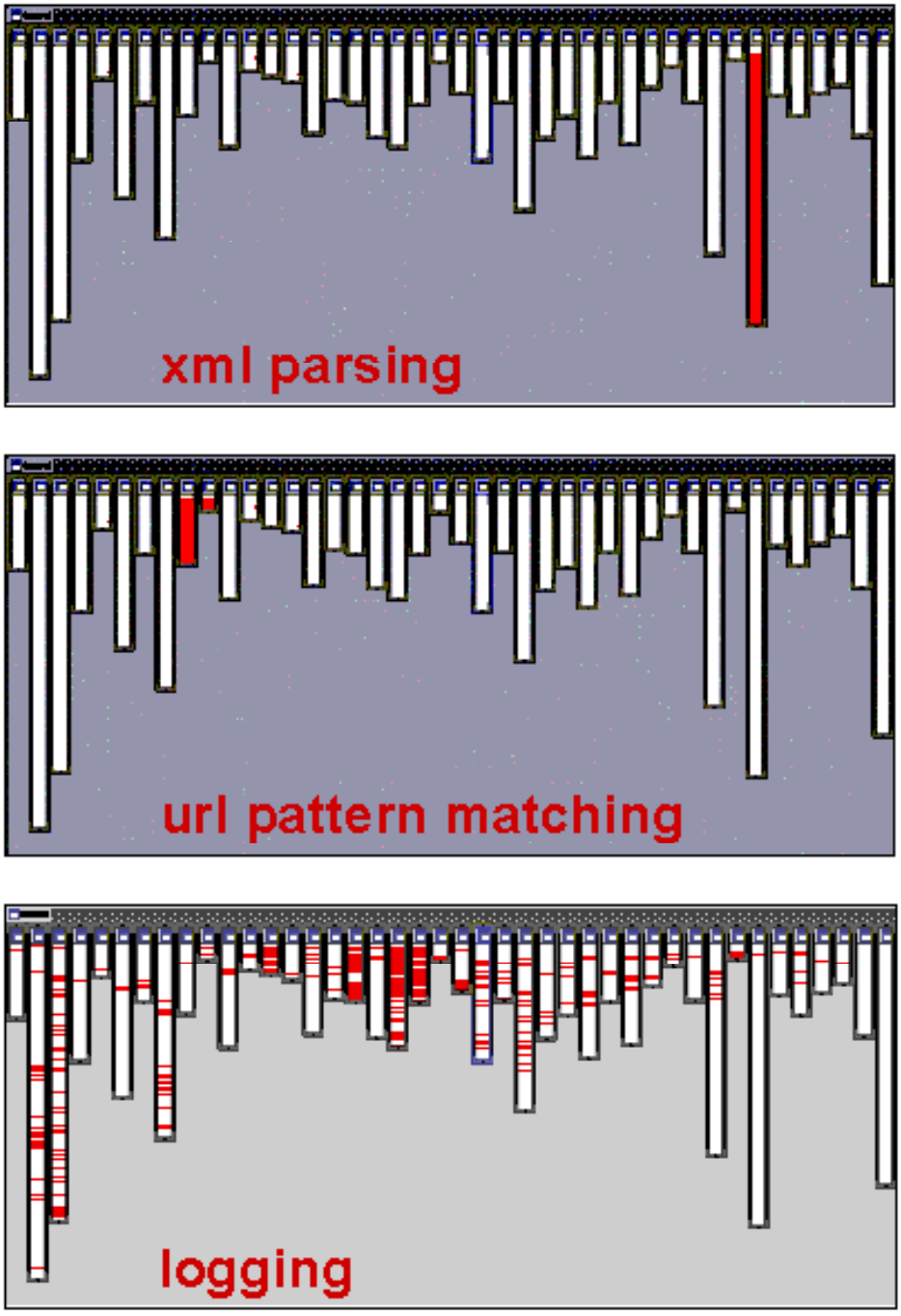
Images

- La décomposition objet est bi-dimensionnelle, les aspects y sont orthogonaux.



tirée de Untangle your code with aspect-oriented programming ; Frank Sauer, TRC

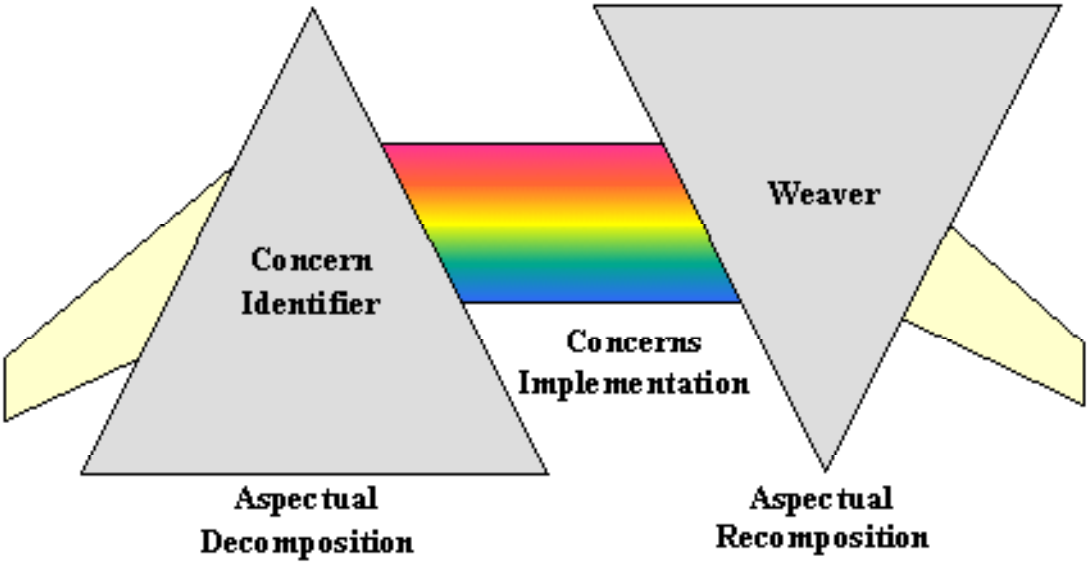
- l'exemple du logging, le non-modularisable : L'exemple suivant, connu comme le loup blanc dans le monde AOP Java, illustre bien le problème : dans le code du moteur Servlets/JSP Tomcat, certains besoins techniques sont très localisés (lecture des fichiers configs, et donc parsing XML), d'autres sont répartis sur peu de classes (traitement des requêtes HTTP, et donc expressions régulières), mais il en est qui se trouvent dans quasiment tout le code (appel au framewoek de log, pour diagnostiquer et journaliser les problèmes survenus en fonction de leur gravité, ainsi que pour rendre compte des requêtes traités par le serveur).



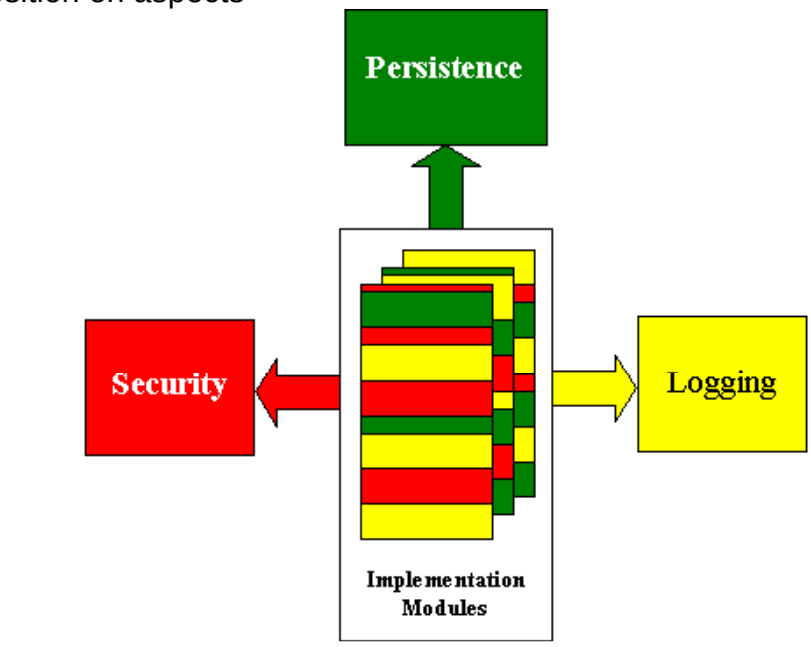
tirée de Aspect-Oriented Programming with AspectJ : Erik Hilsdale, AspectJ.org

les 3 images qui suivent sont tirées de I Want my AOP ! : JavaWorld : Ramnivas Laddad

- L'image de l'expension du problème (et donc décomposition en aspects et composents) puis la réduction (et donc le tissage des deux)



- la décomposition en aspects



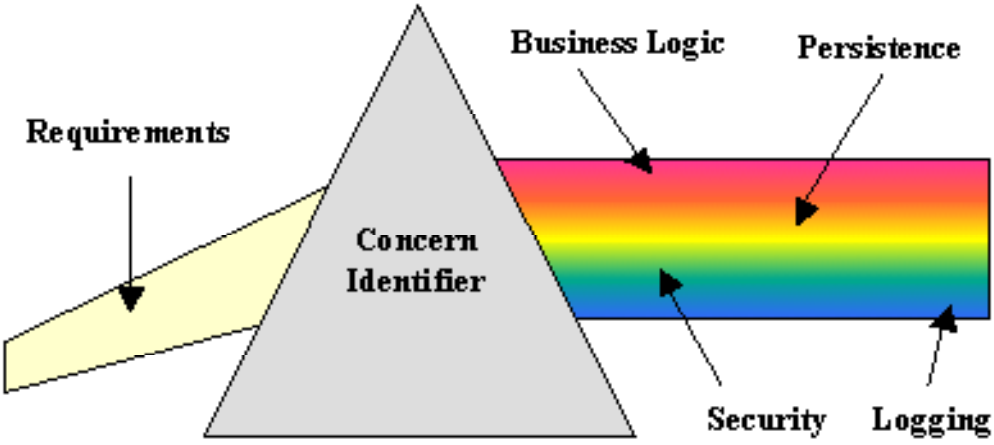
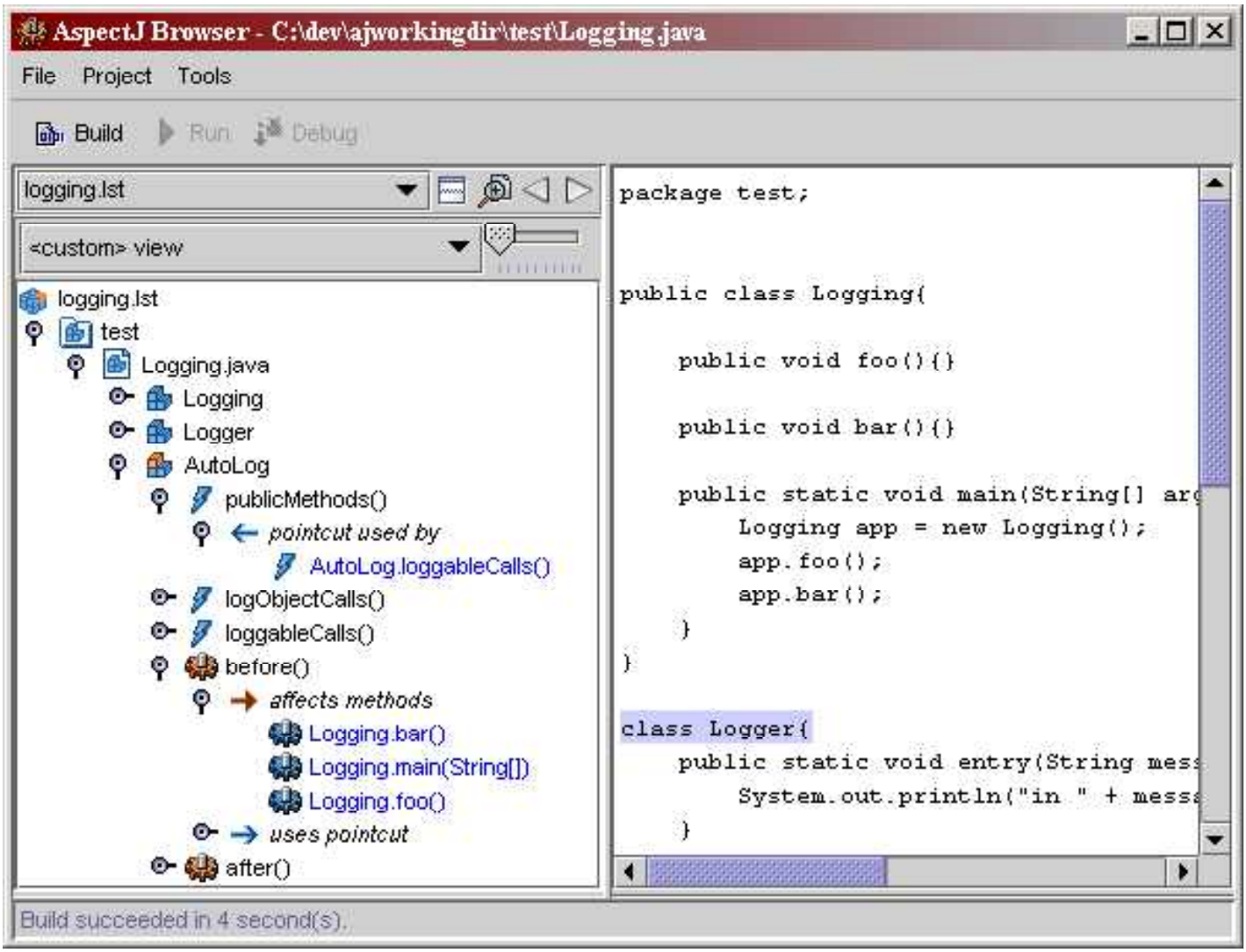


image tirée de Improve modularity with aspect-oriented programming Nicholas Lesleki

- L'image de l'outil visuel d'AspectJ



Conclusion

L'application automatique des aspects au nouveau code est l'une des atouts principaux de l'AOP : l'auteur d'un nouveau code ne doit pas s'inquiéter des composants croisés afin d'y participer. Grady Booch considère l'AOP comme étant l'une des 3 mouvements qui marquent -ensemble- le début d'un changement fondamental dans la façon d'écrire ou de concevoir les logiciels. **voir "Through the Looking Glass" / Ressource section**

Ciczales considère dans sa présentation en 97 que l'AOP, considérée comme un approche explicite de programmation est relativement une idée jeune, qu'il reste plein de travail à faire pour exploiter toutes les possibilités de l'AOP et mieux comprendre sa relation avec les autres idées déjà existantes pour la rendre accessible au plus grand nombre d'utilisateurs.

Pouvoir quantifier l'importance et le gain de l'AOP on aura besoin d'éffectuer fréquemment des divers tests appliqués aux problèmes réels, et mettre le point sur l'utilité de l'AOP comme étant une amélioration ou étention des techniques présentes. Finalement, l'AOP s'est prouvée être plus facile à développer et maintenir tout en étant afficace face à des thèmes plus complexes.

L'AOP aborde un espace de problème qui n'a jamais été traité par les languages orientés objets ou les languages procédurales. AspectJ fournit rapidement des solutions élégantes et réutilisables à des problèmes qui semblaient être les limitations fonsamentales de la programmation.

Evaluation de l'AOP

Bien que l'AOP paraît promettante dans le domaine des séparations des occupations, est-ce qu'elle le sera aussi bien qu'elle en a l'air? deux tests de comparaison pratique ont été effectués sur l'organisation de développement logiciel entre deux implémentations avec et sans AOP par [MWBRLK2001] , le premier visant à déterminer si AspectJ améliorera la capacité des développeur à localiser et corriger les bugs dans un programme multi-threadé, et le second visait la facilité de changement dans un système réparti. Chaque expérience avait 3 essais réalisés par 2 groupes, une avec AspectJ, et l'autre avec un langage de controle. La conclusion était optimiste :

- il est plus facile pour les participants de comprendre (et debogger) un aspect ou une classe si l'effet de l'aspect est sur des parties claires du système.
- AOP a changé la façon avec laquelle les participants abordent les tâches. Le groupe AOP a cherché premièrement une solution qui peut être modularisée dans un aspect, et ça pourrait prendre largement plus de temps si la solution ne pouvait pas être encapsulé dans un aspect.
- Il est plus facile de bâtir et debugger des programmes basés sur l'AOP si l'interface est étroit (en fait, un aspect a un effet bien défini sur un point précis du code), et que la réference de l'aspect vers le code de base est unidirectionnel.
- Il est plus facile de comprendre et gérer un aspect s'il forme une sorte de colle entre 2 structures orientées-objets.
- L'AOP paraît prométtant, mais d'autres projets seront nécessaires pour déterminer comment, quand, où et dans quels projets on peut tirer plein d'avantage de l'AOP.

Un autre groupe [PC2001] a basé ses tests sur un développement d'un système de contrôle de température (TCS) . 4 aspects majeurs et leus relations dans le TCS ont été identifié: la modélisation mathématique, représentation du monde réel, gestion d'emploi du temps et la synchronisation. Les deux premiers doivent rester séparées pour éviter du code entrelacé, quant aux 2 autres, ils s'interagissent entre eux. Dans ce test, un groupe a implementé le tout via un approche orienté objet, et l'autre ustilisant AspectJ. L'appréciation générale :

- La séparation fournie par l'AOP est bien efficace dans le cas des interfaces étroites (même résultat que celui de [MWBTLK2001].
- L'effort majeur était de localiser l'interface entre les préoccupations essentielles et le code de base. Par contre, il n'y avait pas plus de difficulté dans l'implémentation. Ce qui propose que les chercheurs ont besoin d'apprendre davantage comment voir un système sous forme d'aspects.
- L'AOP n'a pas de défauts significatifs en terme de performance.
- La bonne séparation des préoccupations doit être renforcé par des moyens architecturals. Les méchanismes fournis par l'AOP ne peuvent pas remplacer une bonne conception.
- Il est plus facile d'écrire et changer certain types de préoccupations; mais d'autres doivent être liés aux constructeurs des aspects.

AOP : les pours et les contres :

Premièrement, avec peu d'aspects, l'AOP arrive à gérer le **tracing**, deboggage et **instrumentation support** pour un système complexe. En plus, l'AOP peut réduire et limiter une implémentation de quelques **design patterns** avec un petit nombre d'aspects. Deuxièmement, avec un seul aspect, l'AOP peut chopper les protocoles de gestion d'erreur et les algorithmes de partage de resource faisant appel à plusieurs classes, tout ça grâce à leur faculté d'entrelacer, par principe, la modularité des classes.

D'autre part, l'AOP a ses faiblesses. Par exemple, la structure du programme peut devenit très complexe, ce qui exigera plus de descriptions pour la structure, par rapport aux techniques conventionnelles. De plus, l'AOP vise à diviser les aspects d'un programme en divers blocs, pour les re-combiner afin d'obtenir le programme actuel. Ce type de stratégie traite la synchronisation comme étant un simple aspect monolithic, ce qui nous emmènerair à une certaine inflexibilité. Finalement, ce genre de problème peut être éviter en adoptant une implémentation qui appliquera les concepts d'aspects à un système orienté objet ...

L'AOP est une bonne altérnative pour implémenter les systèmes d'exploitation. Elle est concue sur une technique de programmation orientée-objet, mais elle dépasse la barrière des entrelaçements naturelles des systèmes orientés-objet en fournissant une couche d'abstraction au dessus de celle des fonctionnalités, ce qui offre l'implémentation des systèmes d'exploitation plus de fléxibilité par rapport aux techniques traditionnelles. De plus, l'AOP jouit d'un grand potentiel à gérer les tâches systématiques d'une façon bien définie.

Finalement

L'AOP introduit un nouveau style de décomposition et c'est une façon prométteuse de séparation des préoccupations qui ne le sont pas aisement dans des programmes OOP. Elle est bien efficace quand il s'agit de préoccupations unidirectionnelles aux effets bien définis, comme le deboggage et la gestion des traces, mais l'AOP est toujours en phase de recherche. Actuellement, 3 domaines de recherche AOP sont importants :

- La facilité d'utilisation, les techniques qui permettent de voir un système en terme d'aspects doivent être bien documentées, et la création de principes de conception (comme UML pour AOP, et l'AOP design patterns, ce qui aura pour effet de réduire le coût d'adaptation des ingénieurs logiciels vers ce nouveau paradigme.
- Plus d'études et d'éxperiences sur l'utilité de l'AOP ce qui pourrair encourager les géants industriels.
- Les outils de développement pour découvrir les aspects, restructurer les codes existants.

Ce sont ces 3 facteurs qui décideront si finalement les industriels adopteront ou pas l'AOP...

Intentional programming

But : réussir à modulariser au maximum la programmation et accentuer le fait que les langages peuvent etre adaptés selon les souhaits des programmeurs (appelés intentions).Le but est donc d'obtenir un langage modulaire.

Leger historique : La vision de l' « intentional programming » est due à Charles Simony, directeur de la section application de microsoft et auteur de Word .Son but : que les programmeurs puissent exprimer leurs intentions explicitement ds le code.
Dès le début des années 90, ils ont développé un système de « programmation simpifiée (compilateur rendant les erreurs sous forme de graphe,…).

Idée :
Utiliser un langage hôte mais réduit à sa forme minimale (cad pas de besoin de for car le while suffit amplement, …).Ce langage sera alors étendu par un riche ensemble de méthodes fortement réutilisables et surtout indépendantes les unes des autres.
Ces « intentions » sont des composants réutilisables, l'optimisation se fait alors à l'intérieur de classes ou de méthodes abstraites et l'interraction doit être spécifiée comme une interface des composants.

Comment ?
En fait, ils ont fortement utiliser les grammaires attribuées ,introduite dès la fin des années 60 par Don Knuth et vue dans le cours de licence et se basant sur l'idée d'exprimer la sémantique des langages avec le meme niveau de rigueur que leur syntaxe. La principale innovation sur la notion de grammaire attribuée est la notion de « forwarding ».Cette notion est utilisée de la même façon que l'héritage mais aussi pour choisir à la compilation entre différentes implémentations de la même syntaxe. En particulier, « forwarding » peut etre utiliser pour implémenter le surchargement (overloading), et aussi pour implémenter les optimisations.

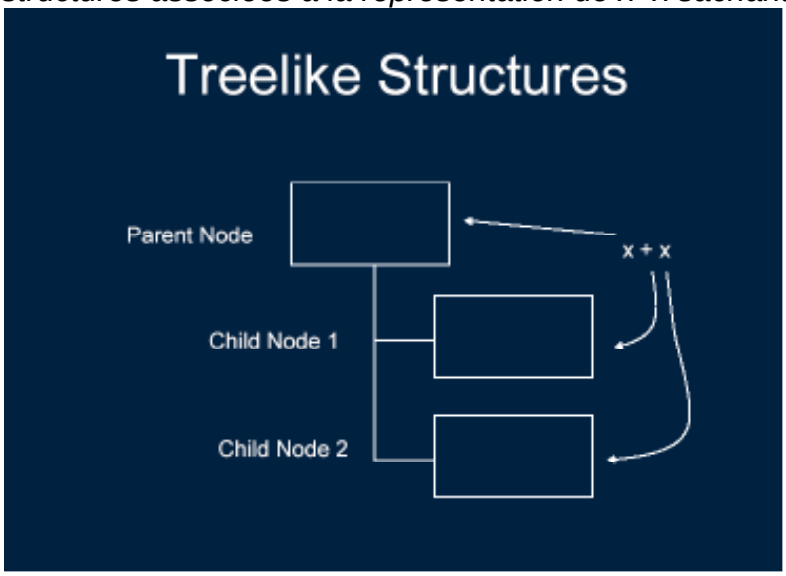
Concepts généraux

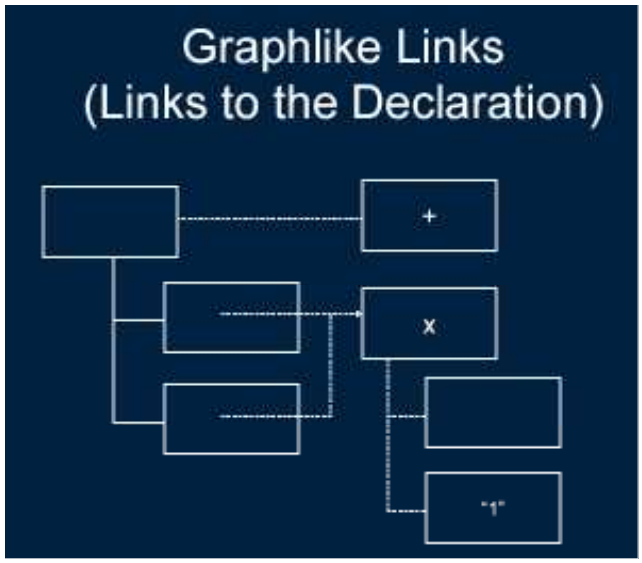
La source est représenté comme un graphe dont le comportement est donné lors de la programmation.
Le comportement du source est déterminé par des méthodes opérant sur le graphe du source.
Utilisation d'un langage hôte comme C.
Les abstractions du langage sont appelées "intentions".
Elimination de l'analyse syntaxique. En effet, elle limite l'extensibilité du langage de programmation.Les règles de syntaxes artificielles sont évitées.
Le graphe source est construit comme un élément qui "tape" le code source.
Il est possible de naviguer dans le graphe pendant la programmation.

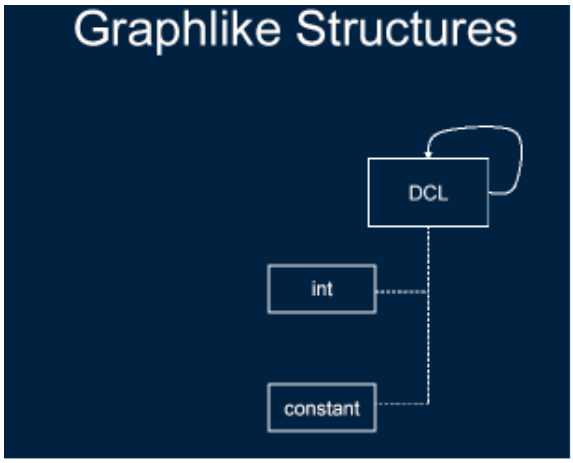
Les programmes IP sont codés dans une structure arbre (tree-like data structure) où chaque noeud possède un graphe (graph-like structure) contenant la définition de l'intention lié à ce noeud.

Exemple pour la compréhension des structures extraits de "I P" par Naveed Arshad

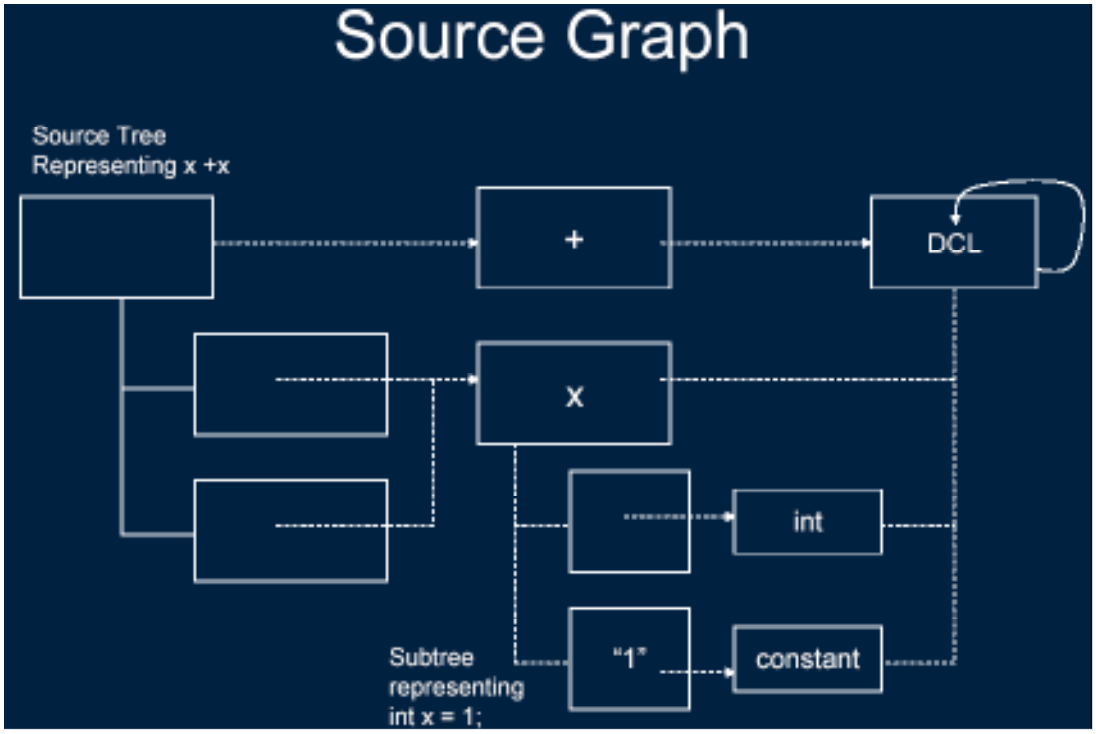
structures associées à la représentation de x+x sachant la declaration (DCL) int x=1;







Graphe du source obtenu pour cette représentation



Les sémantiques des intentions sont décrites par les transformations sur l'arbre, lesquelles convertissent les instances contenues dans les primitives d'intentions, et à partir desquelles le code interprêté peut-être généré par des moyens standards.
Le look des intentions est définie par le programmeur, ce qui facilite l'interraction avec le code, et permet donc de programmer selon son propre style sans effet sur le code interprêté.
Les intentions peuvent être définis pour toutes les méthodes de tous les langages d'héritage. De plus, des codes provenant de ces langages peuvent être importés sans perte d'information ou d'efficacité.

Descriptions des differentes méthodes de la prog. par intentions.

Les méthodes d'interpretation (Rendering methods) permettent d' afficher le graphe source sur l'écran mais aussi sa représentation en mode textuel ou en mode graphique (2D)

Les méthodes d'écriture (Type-in methods) sont appelée quand l'arbre source est manipulé et créé des séquences d'attachement lors de l'affichage du source à l'écran

Les méthodes de réduction sont utilisés dans le processus de transformation de l'arbre source à l'intérieur des structures de bas niveaux et la réimplémentation des constructions de haut niveau avec des structures de bas niveau.

exemple :

Les méthodes de débogage permettent un débogage non-linéaire, elle donne aux intentions la capacité de transformer le code non-linéaire avec une telle méthode.

Les méthodes d'édition et de refactorisation permettent de doter le système de programmmation d'outils performant pour l'édition et la restructuration du code.

Les méthodes de contrôle de version permettent de gérer le développement de simultané de plusieurs programmmeurs sur le projet en gérant la resolution de conflits lors de la modification du même code. Cette condition est obligatoire à la conception d'un véritable système de programmation.

Spécificités de cette forme de programmation

- les programmeurs peuvent ajouter des librairies d'extension
- de nombreuses librairies sont disponibles
- les librairies d'extension peuvent étendre n'importe quelle partie de l'environnement incluant le compilateur, le debuggeur,...
- Refactoring:

Dans les langages traditionnels, le refactoring est difficile car le design de l'information n'est pas disponible dans la source (il faut connaitre le langage utilisé!) Cette pratique consiste à remanier le code existant dans un souci permanent de simplicité et de qualité. Elle permet notamment, par des transformations microscopiques et successives, de faire apparaître une architecture dans du code qui en était dépourvu. Sa définition est « toute modification d'un programme permettant d'en améliorer la structure interne sans en modifier le comportement externe ». Pendant ces périodes, le développeur permet de prendre un peu de recul sur le code et de revoir d'un coté l'architecture du programme et de l'autre, la simplicité du code. Par exemple, il s'agit de décomposer les algorithmes en entités très courtes à placer dans des méthodes distinctes, de donner des noms clairs aux variables et méthodes et surtout de positionner correctement dans l'espace, les blocs formés par les boucles et les conditions. Or dans l' IP le source est représenté comme un arbre syntaxique abstrait (AST) et ainsi, les représentations de haut niveau peuvent être accessibles.

- Meta-programming:

Les mêmes abstractions qui sont disponibles au niveau de base peuvent être utilisées pour fournir les librairies d'extension.De plus, les mêmes intentions peuvent être utilisées pour écrire du code source et des librairies.On peut aussi fournir des librairies qui génèrent un warning à la compilation.

- Code hérité:

Il est possible d' importer des codes venant d'autres langages, pour cela il est nécéssaire d'avoir un analyseur syntaxique et les intentions pour ce langage spécifique.Il est ensuite facile d'améliorer le code importé.

- Outils d' édition et d' exploration:

L' éditeur a une structure de graphe, qui se construit au fur et à mesure que l'on tape le code, et l'on peut observer le code sous sa forme textuelle ou graphique.

- Debugger :

Le debuggeur est efficace car il permet au développeur de passer "à travers" l'execution du programme à des niveaux différents.Il peut aussi débugger le système IP lui-même.

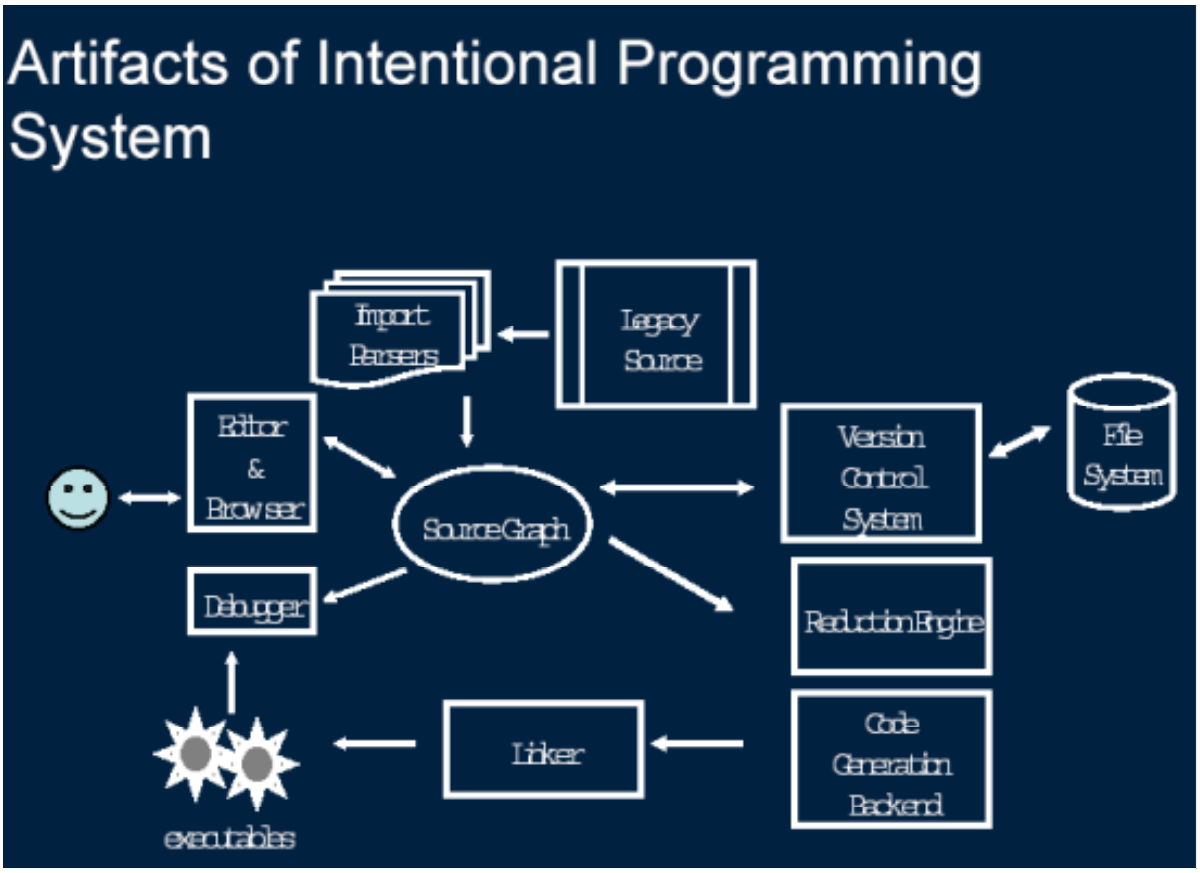
- Version control system :

Cet outil pemet de fusionner diverses versions d'un même code.Il est capable de fusionner corretement les modifications faites sur une fonction par deux utilisateurs différents.

- Moteur de reduction:

Le moteur de reduction fournit la charpente de la transformation du code, il joue le rôle d'un compilateur sans en être un.La principale différence est qu'il n'y a pas d'analyseur syntaxique dans le moteur de reduction. Il résulte d' un ensemble limité de primitives d'abstraction appelées "code réduit" ou "R-code".

Schéma final



Generative Programming

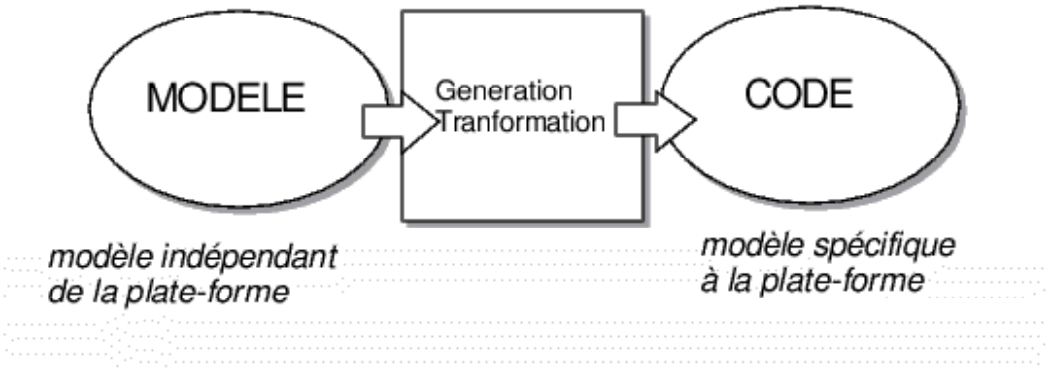
Remarque : s'aide des principes de l'aop...

idée : simplifier la programmation en gérérant du code, à partir de modèles.
La GP est une technique qui permet d'éviter la réécriture de plusieurs parties de code, en effet, elle permet de réutiliser aisément un code existant. Elle consiste à résoudre un problème de la façon la plus général possible sans perte d'efficacité, c'est-à-dire que le code écrit résoudra une famille de solutions et non, une solution paticulière.

exemple : un tableau d'entier est une solution particulière
un vecteur est une solution gén.

but :
Le but de la GP est de remplacer les manuels de recherche et un assemblement de composants, avec une génération automatique des composants nécéssaires selon la demande.

Comment?



Principe de la GP:

La séparation des préoccupations est un principe fondamental des techniques de programmation par aspects. Ce terme, inventé par Dijkstra, réfère à l'importance de s'occuper de la question du temps. Pour éviter à un code de s'occuper de plusieurs problèmes simultanément, la GP sépare chaque problème à l'intérieur d'un ensemble distinct de code. Ces "sous-codes" sont ensuite combinés pour générer le code nécéssaire.

La séparation de l'espace du problème de l'espace des solutions.
L'espace du problème consiste dans l'abstraction du domaine spécifique avec lequel les programmeurs d'application voudraient intéragir, alors que l'espace des solutions contient les composants d'implémentation.Ces deux espaces ont des structures différentes et c'est pourquoi leur liaison se fera selon une configuration verticale (c'est-à-dire entre deux différents niveaux d'abstraction).

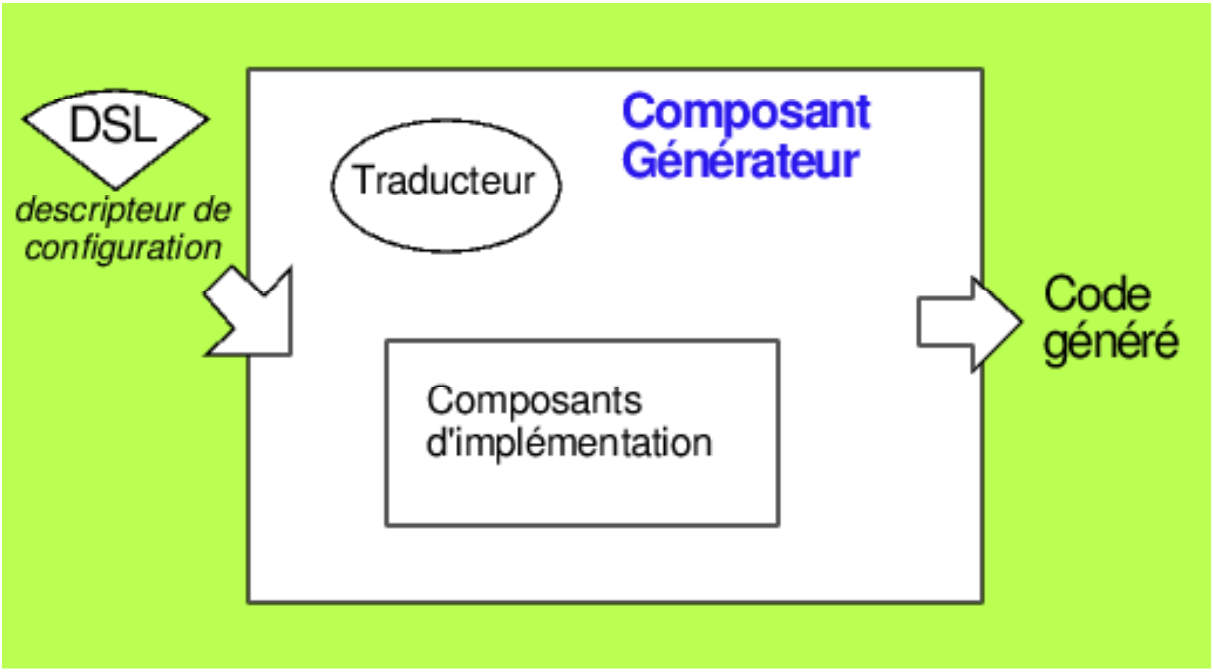
La paramétrisation des différences. Comme dans la programmation générique, la paramétrisation nous permet de représenter des familles de composants (composants avec plusieurs principes communs)

Analyse et modélisation des dépendances et interractions. Toutes les combinaisons de paramètres ne sont pas valides, et la valeur de certains paramètres peuvent impliquer la valeur de plusieurs autres. Ces dépendances sont représentés comme une configuration horizontale, puisqu' elles apparaissent entre les paramètres dans le même niveau d'abstraction.

Elimination supérieure et optimisation des performances.En générant des composants de manière statique (c'est-à-dire à la compilation), cela permet d'éliminer le code inutilisé et les vérifications à l'éxécution. Des améliorations importantes peuvent ainsi être produites.

Les composants de génération

Les composants générateur accepte un descripteur de la configuration (DSL) qui va être interprété et qui va permettre d'assembler les composants concrets désignés dans ce descripteur.



Qu'est-ce qu'un DSL?

Un DSL est un langage abstrait relativement ciblé et réduit (spécifique au domaine d'application) mais qui donne aux développeurs une capacité d'expressivité beaucoup plus forte qu' un langage d'implémentation tel qu'on en connait.

Que se passe-t-il au niveau du compilateur ?

Le compilateur de la Generative programming se distingue par ces deux phases de traitement.

La première phase, appelée phase de raffinements, permet la decomposition du code, le choix de la meilleure représentation et du meilleur algorithme.Cette phase permet ainsi d'obtenir un code concret et spécialisé.

La seconde phase est une phase d' optimisations du code, elle utilise les principes suivants.

- inlining

  L'expansion en ligne est une version élaborée de la macro-expansion du prétraitement. Elle remplace l'appel d'une fonction par le corps de la fonction où les arguments sont fixés. Plusieurs appels différents correspondront à une nouvelle traduction du corps de la fonction. On évite ainsi d'exécuter la séquence d'appel de fonction, mais la taille du code peut s'en trouver accrue.

- mise en mémoire cache
- fusion des boucles (exemple de l' addition de matrices)
- déroulement des boucles lorsque le nombre d'itérations de celle-ci est faible.
- mouvement de code

  déplace les codes invariants hors des boucles.

  exemple

```
while( i < 10){
        resultat = fibonacci(200);
        /* code dépendant */
}
```

  sera transformé en

```
resultat = fibonacci(200);
while( i < 10){
        /* code dépendant */
}
```

- élimination des sous-expressions communes
- élimination du code inutile
- évaluation partielle

  Evaluer partiellement une fonction dont plusieurs de ces paramètres sont des constantes dans le contexte spécial.

Exemple célèbre de la voiture

Prenons l'exemple d'une voiture basique à laquelle nous voulons ajouter un autoradio et une climatisation. De plus, nous avons besoin d'encadrement pour l'installation de ces équipements.
Nous avons aussi besoin d'une batterie, dont la capacité devra être suffisante pour alimenter l'auto-radio et la climatisation. Voici ci-dessous, un modèle basique pour cette représentation..
Exemple tiré du document de Ulrich Breymann, Kerzysztof Czannecki et Ulrich Eisenecher (réf. cza97).
Composants

Les données de configuration de la voiture sont encapsulées dans la structure Config. template<class Config> class Car { public: typedef typename Config::radio radio; typedef typename Config::radioFrame radioframe; typedef typename Config::airconditioner airconditioner; typedef typename Config::airconditionerFrame airconditionerFrame; typedef typename Config::battery battery; private: radio r; radioframe rf; airconditioner ac; airconditionerFrame acf; battery b;	template<class AirCond, class Radio> class CarConfiguration { public: typedef Radio radio; typedef typename Radio::frame radioFrame; typedef AirCond airconditioner; typedef typename AirCond::frame airconditionerFrame; typedef selectBattery<AirCond, Radio>::battery battery; };

Le tableau suivant liste les équipements disponibles..

part	available variant
air conditioner	AC100 (needed power = 100) AC150 (needed power = 150) AC180 (needed power = 180)
radio	Radio10 (needed power = 10) Radio50 (needed power = 50)
frame for air conditioner	small AC frame big AC frame
frame for radio	small radio frame big radio frame
battery	battery with power 150 battery with power 200

La taille de la charpente de l'auto-radio et sa consommation dépendent de la taille de celui-ci.La même chose pour la climatisation. Nous supposons que les gros équipements électriques nécéssitent de grosses charpentes et beaucoup de puissance. Les classes suivantes modèlisent ces parties et leurs dépendances.

```
// PARTS
// frames
class SmallRadioFrame {};
class BigRadioFrame {};
class SmallAirconditionerFrame
{};
class BigAirconditionerFrame {};


// radios:
class Radio10
{    public:
       enum {POWER=10};
       typedef SmallRadioFrame
frame;
};


class Radio50
{    public:
       enum {POWER=50};
       typedef BigRadioFrame
frame;
```

```
// airconditioners:
class AC100
{    public:
       enum {POWER=100};
       typedef SmallAirconditionerFrame
frame;
};


class AC150
{    public:
       enum {POWER=150};
       typedef BigAirconditionerFrame
frame;
};


class AC180
{    public:
       enum {POWER=180};
       typedef BigAirconditionerFrame
frame;
};
```

La puissance de la batterie doit être assez large pour permettre le fonctionnement des deux équipements décrites précédemment.

```
// template for battery search:
template<int power>
class Battery
{    public:
       typedef typename
Battery<power+10>::battery battery;
};


// available batteries:
template<>
class Battery<150>
{    public:
       Battery() { cout << "Battery with
power 150" << endl;}
       typedef Battery<150> battery;
};
```

```
template<>
class Battery<200>
{    public:
       Battery() { cout << "Battery
with power 200" << endl;}
       typedef Battery<200> battery;
};


// limit search in selectBattery below:
// no type battery available
template<>
class Battery<0> {};
```

Il est maintenant nécéssaire de pouvoir selectionner la batterie la plus appropriée aux équipements présents.La recherche se fait recursivement, par pas de 10 unités de puissance.

```
// configuration rule
template<class AirCond, class Radio>
class selectBattery
{    public:
       typedef typename
          Battery    <(AirCond::POWER+Radio::POWER // minimum
                       // required power
                       <= 200 ? // biggest available battery
                      AirCond::POWER+Radio::POWER // ok
                       : 0) // search failed
                    >::battery battery;
};
```

Il est alors facile de construire automatiquement la voiture équipée dans notre programme.La batterie est déterminé par le compilateur et une erreur de configuration amènera à une erreur de compilation.

```
Car<CarConfiguration<Radio10, AC100> > theFirstCar;
Car<CarConfiguration<Radio10, AC150> > theSecondCar;
Car<CarConfiguration<Radio50, AC100> > theThirdCar;
Car<CarConfiguration<Radio50, AC150> > theFourthCar;
// next line yields compiler error, because no
// battery with power  230 is available
Car<CarConfiguration<Radio50, AC180> > PowerCar; //
error
```

Pour programmer des structures plus complexes, il devient nécéssaire d'utiliser des structures plus puissantes comme si,tantque,...

Composition Filter

Introduction

Le Groupe TRESE de l'Université de Twente, département d'ingénierie logiciel, a développé le modèle de CF, capable de resoudre les nombreuses déficiences de la POO. Cette nouvelle forme de programmation se base pourtant sur la POO et, a pour but de l'étendre. Pour cela il a été nécéssaire d'introduire des filtres d'entrée et de sortie gérant l'envoie et la réception de messages au "meta level".
La programmation par CF a pour but de fournir des techniques pour supporter la conception de gros projets logiciels. L'un des aspects les plus importants est la tentative d'améliorer la complexité de beaucoup de systèmes. Les filtres sont utilisés pour exprimer des préoccupations complexes et entrecoupées.

Historique

L'histoire de la Composition Filter Programming démarre rééllement en 1991 avec la première vision des filtres, mais il faut souligner que l'equipe de l'Université de Twente travaille la Programmation par Objets et ses évolutions depuis 1987
Le modèle de CF est une évolution des 1ères versions du langage Sina
Au lieu d'étendre le langage avec de nouvelles constructions provenant de langages différents, le framework de CF a été introduit, lequel intègre toutes les constructions dérivées de ces différents lanagages et leurs interfaces à l'intérieur d'un modèle simple et unifié.

Spécificités des filtres

Les propriétés d'extension modulaire et orthogonale distinguent la CF des autres techniques orientées aspect.
L' extension modulaire signifie que les filtres peuvent être attachés à des objets exprimés dans différents langages sans modification de la définition des objets, elle rend les filtres indépendants de l'implémentation.
L' extension orthogonale signifie que la sémantique d' un filtre est indépendante de celle des autres filtres, elle rend les filtres composables.
La CF permet d'introduire de nouveaux filtres si nécéssaire (Open-ended).
Chaque filtre représente un aspect (préoccupation), il est prédéfinie et possède une sémantique bien définie.
Un filtre décrit ce qu'il va faire et non comment il est implémenté.

Principes

Les versions récentes de la CF incluent le principe de superimposition. Ce principe décrit l'endroit dans le code où le comportement de l'aspect doit être ajouté. Ce modèle de CF supporte les comportements entre-coupés en travers d'un ensemble de méthodes supportées par une ou plusieurs classes.

Le modèle de CF adopte les spécifications déclaratives, cela siginifie que l'on ne veut pas savoir comment est codée la méthode mais uniquement ce qu'elle fait.
Le modèle de CF est une extension modulaire de la POO, mais pour supporter les spécifications de la CF, il faut utiliser une interface qui enveloppe les composants d'implémentation.

Représentation simplifiée du modèle de CF (tirée du document "Composing Multiple concerns using composition filters" du groupe TRESE).

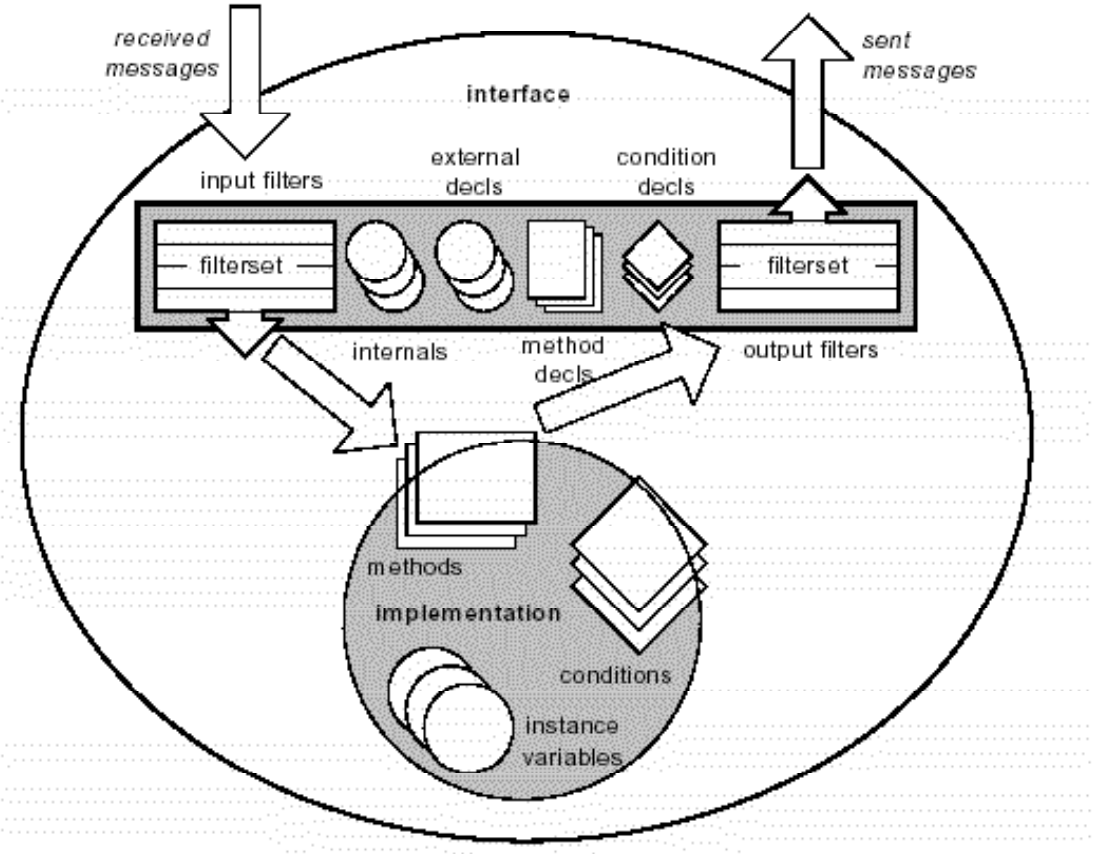


Figure 5. Simplified representation of the CF model.

Les filtres décrives les comportements de l'objet. Chaque filtre spécifit une inspection et une manipulation particulière des messages. Les filtres d'entrée et de sortie peuvent manipuler les messages reçus, respectivement envoyés à l'objet. Ils sont déclarés dans filtersets. Les filtres peuvent faire référenceà des objets internes ou externes. Les objets internes sont instanciés et encapsulés à l'intérieur de l'objet CF alors que les objets externes sont des références à des objets extérieurs à l'objet CF.
Les filtres définissent le comportement de l'objet comme une composition des comportements de ces parties d'implémentation. L'interface est une extension modulaire et indépendante du langage, de la partie implémentation.
Les méthodes de condition de la partie implémentation permettent d'obtenir des informations sur l'état de l'objet.

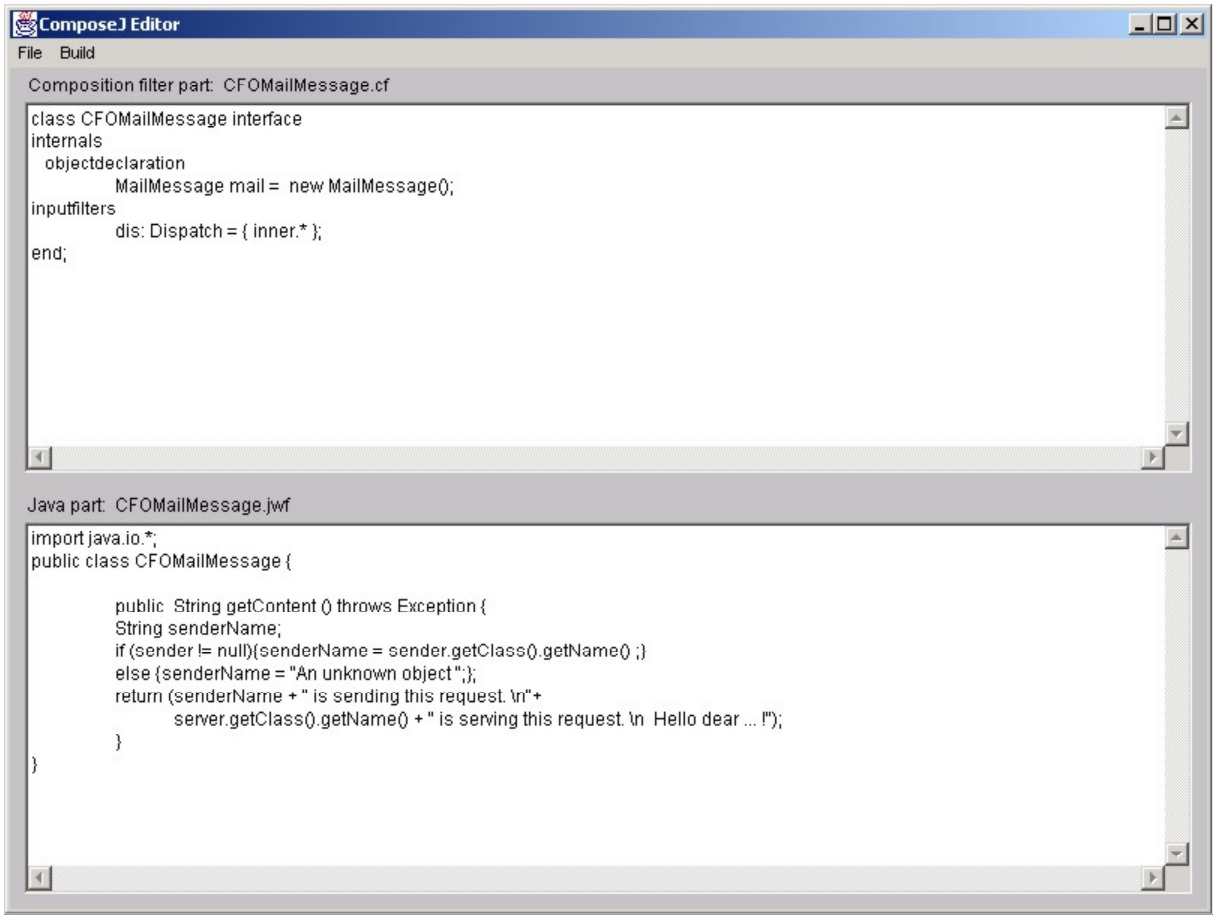
voir exemples pages suivantes

Exemple de logging:
L'aspect Logging se compose de NotifyLogger, lequel est superimposé sur tous les aspects à part Logging. Logging est implémenté en enyoyant tous les messages reçus comme des objets à l'objet global logger. L'aspect Logging creer un booleen interne logOn pour toutes les instances, lequel est utilisé pour rendre actif ou non le "logging" des messages.
extrait de "Composing Multiple Concerns Using CF" (voir bibliographie).

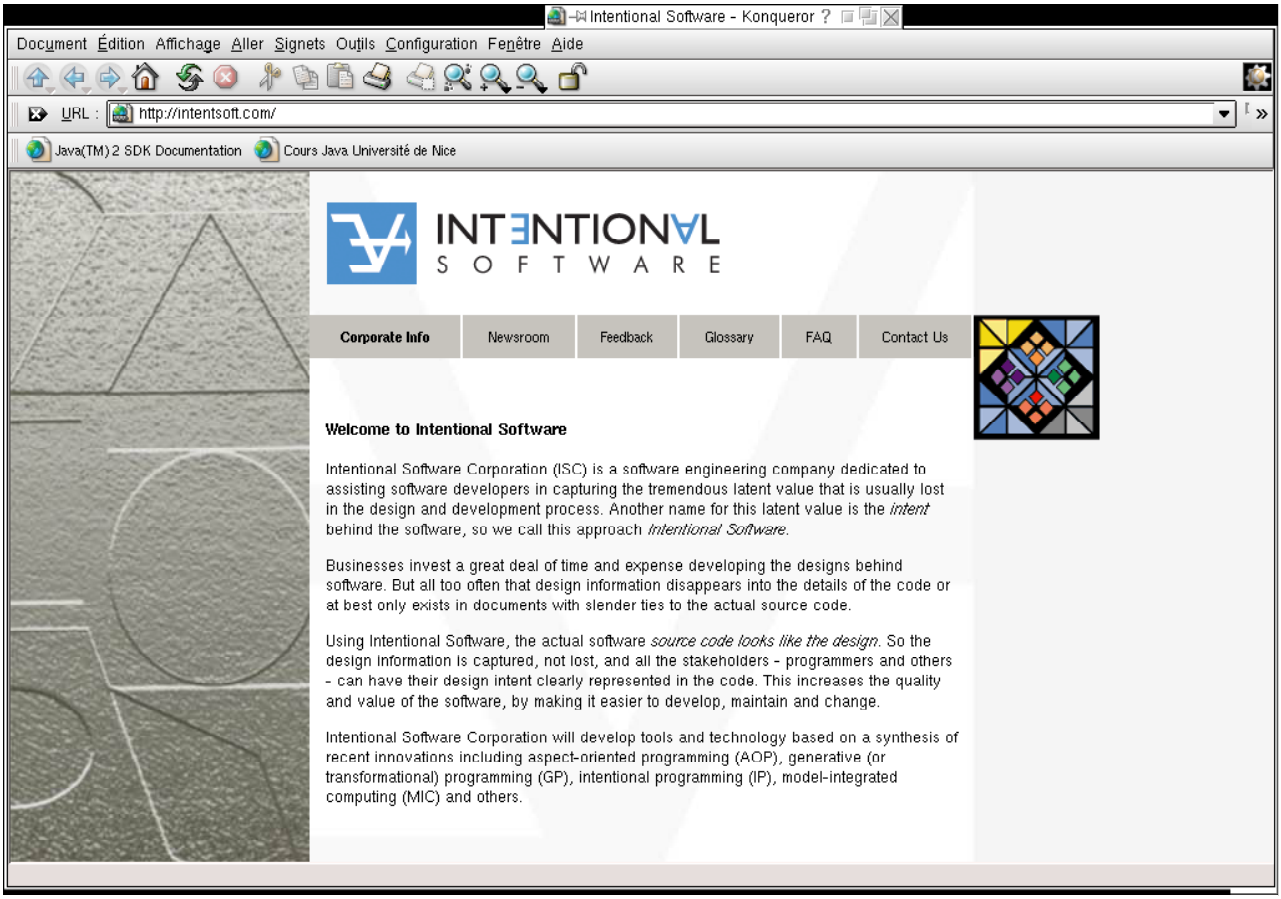
```
concern Logging begin // introduces centralized logger
   filterinterface notifyLogger begin // this part declares the crosscutting code
      externals
         logger : Logging; // *declare* a shared instance of this concern
      internals
         logOn : boolean; // created when the filterinterface is imposed
      methods
         loggingOn();  // turn logging for this object on
         loggingOff();  // turn logging for this object off
         log(Message);  // declared here for typing purposes only
      conditions
         LoggingEnabled;
      inputfilters
         logMessages : Meta = { LoggingEnabled=>[*]logger.log };
         dispLogMethods : Dispatch = { loggingOn, loggingOff };
   end filterinterface notifyLogger;

   filterinterface logger begin //defines the interface of the logger object itself
      methods
         log(Message);
         // various methods for information retrieval from the log
      inputfilters
         disp : Dispatch = { inner.* };  // accept all methods implemented by myself
   end filterinterface logger;

   superimposition begin
      selectors
         allConcerns = { *!=Logging };  //everything except instances of Logging
      conditions
         allConcerns <- LoggingEnabled;
      filterinterfaces
         allConcerns <- notifyLogger;
         self <- logger;
   end superimposition;

   implementation in Java;
      class LoggerClass {
         boolean LoggingEnabled() { return logOn };
         void loggingOn()  { logOn:=true; };
         void loggingOff()  { logOn:=false; };
         void log(Message mess) { … }; // get information from message and store
      }
   end implementation;
end concern Logging;
```

ComposeJ

ComposeJ est l'implémentation du principe de CF en Java
Le langage de ce logiciel est défini, en évaluant différents aspects de Java et sa syntaxe est montré à travers divers exemples.Pour réduire le fossé entre la vision conceptuelle et l'implémentation, il a fallu définir un modèle de translation décrivant la translation abstraite du modèle CF vers Java.
Le système ComposeJ est une extension du compilateur standard de Java.



INTENTSOFT



Bellevue, WA- 17 Septembre 2002 DR. Charles Simonyi, l'ingénieur distingué à Microsoft et le professeur Gregor Kiczales de l'université de Colombie britannique ont aujourd'hui annoncé la formation d'Intentional Software Corporation (ISC), une nouvelle compagnie qui développera la technologie pour améliorer considérablement la productivité de développement de logiciel en capturant l'intention de calcul clairement dans le code. En outre, un accord de licence avec Microsoft a été signé qui donne à Microsoft un droit d "être le premier dans les négociation" pour des les développements dans les premières années critiques.

Simonyi a travaillé chez Microsoft pendant plus de 20 années où il présidé divers projets de développement de logiciel comprenant Word et beaucoup d'autres applications de Microsoft. Kiczales a mené le développement au centre de recherches de Palo Alto de la programmation aspect-orientée et de l'AspectJ, une nouvelle technologie importante de développement de logiciel.

Kiczales a cité qu'il y'a "une synergie énorme entre la programmation intentionnelle et la technologie orientée-aspect. En intégrant les deux, Intentional Software Corporation pourrait donner à des réalisateurs excités des outils très utiles. Des outils que les utilisateurs d'AspectJ demandent déjà.

Gates et Simonyi ont indiqué que la nouvelle société permettrait à ce dernier d'assembler une équipe unique de talent académique et commercial, et fournissent l'incitation et la discipline du marché nécessaire pour poursuivre la vision incertaine, mais potentiellement révolutionnaire de la programmation intentionnelle.

L' accord s'assure que Simonyi continuera à être une partie importante de la famille Microsoft et fournit des permis à la propriété intellectuelle appropriée de Microsoft de sorte que Simonyi et son équipe puissent continuer à développer cette technologie.

Intentional Software Corporation est consacré à la création d'outils et de technologies de la programmation qui permettent à des programmeurs de capturer l'intention de conception explicitement dans le code. Ces outils amélioreront également la valeur du logiciel en augmentant sa qualité et en la rendant moins chère de maintenir. Les outils développés par Intentional Software Corporation seront basés en partie sur les innovations récentes dans la programmation aspect-orientée, la programmation générative ou transformationnelle, et la programmation intentionnelle.

Les outils intentionnels de logiciel seront de valeur dans presque toutes sortes de développement de logiciel. Quelques exemples où l'utilisation du logiciel intentionnel simplifiera considérablement les améliorations de produits seront la création des websites complexes, des interfaces utilisateur pour les produits communs d'électronique grand public tels que les téléphones et les imprimantes, des systèmes et des interfaces utilisateur incluses pour des applications des véhicules à moteur et de l'avionique, et de beaucoup plus.

Conclusion personelle

La faible avancée du développement de ces techniques ne nous permet à l'heure actuelle de faire une réelle comparaison entre ces formes qui se révèlent fortement complémentaires (voir le projet développé par intentsoft).

Toutes ces nouvelles formes de programmation ont pour but de faciliter la vie du programmeur en ajoutant une interface pour " l'éloigner" du code et de ces difficultés. La façon de programmer est donc différente, plus simple et plus rapide mais ne permettant pas une gestion pointue de la programmation (gestion mémoire, performances,...). La réutilisabilité du code est maximale et le travail en parallèle ou en cascade largement facilité.

On peut faire une rapide analogie avec la différence entre les systèmes Windows et Unix : le premier permet au plus grand nombre d'accéder à l'utilisation d'un OS alors que le second privilègie une utilisation controlée, efficace et avancé.

On pourrait aussi être médisant et penser que ces innovations, financées par l'industrie informatique (Microsoft, IBM et Xerox !), visent uniquement à diminuer les coûts de conception en réduisant les temps de programmation (il ne reste qu' un pas à franchir pour voir la qualification et le salaire des développeurs diminuer lui aussi).

A l'heure actuelle, la concrétisation de ces idées n'est qu' une forme évoluée de la programmation par objets,en effet, le langage Java est utilisé comme base de travail et les concepts implémentés ne s'assimilent qu'à des outils d'aide à la programmation. Alors que le coté théorique nous offre une approche révolutionnaire de la programmation.

Bibliographie

Divers

"Les paradigmes de programmation" revue "Economie et Gestion" mars 03
"Intentsoft.com : history, faq and news"

Aspect Oriented Programming

"An introduction to AOP and AspectJ" par Roger Mac Farlane, Université de Mac Gill.
"AOP with AspectJ" par Rick Hilsdale, AspectJ.org.
"Historical survey" cours csc330, Université of Victoria.
"Future of programming" par Bob Mathis, Université de l' Ohio.
"Untangle your code with AOP" par Frank Sauer, TRC.
"Intro to AOP", occam (zope community)
"AOP" par Jonh Lam, Naleco Research inc
"AOP" par Gregor Kizcales (eccop97)
"The impact of AOP on future application design" par Andrei popovici
"Réflexivité en Java" : Cours DEA : Essi.fr
"Programmation orientée aspect" : Olivier Mangez : BorCon
"La programmation orientée-aspect avec .NET et J2EE" : Thomas Gil : DotNetGuru
"An introduction to Aspect-oriented programming" : Ken Wing Kuen Lee : COMP 610E
"Programmation orientée-aspect" : Frédéric Duclos
"Improve modularity with AOP" : Nicholas Lesleki
"An introduction to AOP with AspectJ" : www.cs.wustl.edu/~naleiden
"Java Aspect Components : Présentation" : AOPSYS
"Aspect-oriented programming and application" : CS5204 : Karen Yang : ViriginaTech
"I Want my AOP !" : JavaWorld : Ramnivas Laddad

Intentional programming

"IP British Computer Society" par le Dr Oege de Moor, Université d'Oxford.
"Interactive Source Code", "Transformations and Visualization of Abstraction using IP system", "Extensibility and Visualization of source code documents" par Lutz Roeder.
"I P" par Naveed Arshad

Generative Programming

cza97 Ulrich Breymann, Kerzysztof Czannecki et Ulrich Eisenecher.
"Generative Programming using Adaptative and AOP" : Karl Lieberherr, Univ. de Boston.
"Generative programming" par Marat Boshernitsan Univ. de Berkeley
"Generative programming in C++"
"Real-time Java" par Angelo Corsaro

Composition Filter Programming

"Composing Multiple Concerns Using CF" par Lodewijk Bergmanns et Mehmet Aksit.
http://trese.cs.utwente.nl page officielle du groupe de recherche TRESE sur la CFP.