Le rôle du langage de programmation dans la réalisation de code de qualité, les cas de Delphi et C++

Dr. Driss Kettani
Alakhawayn University, Ifrane,
Morocco.
D.Kettani@alakhawayn.ma
D.r Zakaria Maamar
Zayed University, Dubai,
United Arab Emirats.
Zakaria.Maamar@zu.ac.ae

Introduction

La quête de l'assurance de la qualité doit être présente tout au long du processus de l'analyse, de la programmation et de la mise en œuvre d'un logiciel[Humphry 95]. Il est donc essentiel de disposer d'une définition applicable à tous les stades du cycle de vie du logiciel et utilisable aussi bien par l'ingénieur architecte du système que par le programmeur qui code les spécifications fonctionnelles. Malheureusement, il n'est pas facile de faire le lien entre les attributs de qualité de haut niveau (appelés attributs externes, par exemple la maintenabilité) et les attributs de qualités de bas niveau (appelés attributs internes, par exemple la complexité cyclomatique). Le programmeur se trouve donc, dans la majorité des cas, non impliqué dans le processus de qualité logicielle et produit par le fait même un code non conforme aux standards retenus.

Nous pensons que tout processus de qualité logiciel doit nécessairement inclure le travail du programmeur et nous croyons qu'il faut satisfaire trois conditions sine quoi none pour atteindre les objectifs escomptés :

1. Adopter un processus de qualité reconnu par une instance international tel que l'ISO (International Standardization Organization) ou le SEI (Software Engineering Institue);
2. trouver un model intermédiaire pour relier les attributs externes aux attributs internes rendant ainsi les attributs de qualité accessibles au programmeur;

3. choisir un langage de programmation qui respecte (et fait respecter) les attributs internes de qualité.

Ces trois conditions sont complémentaires et non séparables.
Dans la section 2 de ce papier, nous définirons la qualité logicielle, les standards internationaux et nous mettrons l'emphase sur le décalage qui existe entre les attributs de haut niveau et les attributs de bas niveau. Dans les sections 3 et 4, nous abordons successivement les caractéristiques de haut niveau et de bas niveau d'un logiciel en mettant l'accent sur les liens qui existent entre ces deux niveaux. Par la suite, en section 5, nous montrerons comment le langage de programmation joue un rôle majeur dans l'aboutissement à la qualité logicielle. Nous présentons à cet effet une étude comparative entre deux langages de programmation populaires, à savoir Delphi et C En conclusion à ce papier, nous présenterons certaines idées et perspectives pour explorer des horizons de recherches prometteuses.

2. La qualité logicielle

Il est difficile de trouver une définition formelle qui s'applique universellement sur le concept de la qualité du logiciel. En effet, dans un sens restreint, certains auteurs pensent qu'un logiciel est de qualité s'il rempli toutes ses spécifications fonctionnelles [Crosby 79], [Sommerville 98] . Le problème avec cette définition est que ces spécifications ne sont pas toujours faciles à obtenir, elles sont généralement incomplètes et parfois même en opposition avec les attributs de qualité adoptés par l'organisation (besoins du client oblige!).
Une autre définition plus large et plus appropriée à notre point de vue est celle qui consiste à lier le concept de qualité au cycle de vie d'un logiciel. Ainsi, selon plusieurs auteurs [Humphry 95], [Fenton 91], un logiciel est de qualité si le processus global de son développement durant toutes les étapes de son cycle de vie a suivi une norme de qualité logicielle adoptée par l'organisation. En effet, il existe plusieurs normes/processus de qualité logicielle proposés(ées) par des organismes de notoriété internationale tels que ISO (ISO-9000, ISO-9001 et ISO-9126), IEEE (IEEE-730, IEEE-830 and IEEE-1012) et SEI (the SEI maturity model). Généralement, toutes ces normes décrivent, en termes globaux et abstraits les caractéristiques de qualité que l'on voudrait voir dans un produit logiciel final. Par exemple, la norme *ISO-9126* (caractéristiques d'évaluation d'un logiciel) qui est bien acceptée en domaine des systèmes d'information stipule qu'un logiciel doit être:

- **Fonctionnel**: résout la tâche pour laquelle il a été conçu;

- **Robuste** : traite convenablement et efficacement les erreurs prévues et imprévues;
- **Utilisable**: dispose d'une interface conviviale qui permet et facilite son utilisation;
- **Efficace**: utilise les meilleures ressources pour effectuer les traitements computationnels;
- **Maintenable**: compréhensible, analysable et facile à mettre à jour;
- **Portable**: peut fonctionner sur d'autres plates-formes moyennant des modifications mineures.

La norme ISO 9126, comme on peut le constater, ne présente aucun intérêt réel aux yeux d'un programmeur puisqu'elle ne donne aucune indication permettant de lier ses attributs de haut niveau avec l'aspect technique, c'est à dire la programmation.
Nous pensons que quelque soit le standard choisi, il faudra toujours gérer cette constante fatale qui est le décalage entre les principes généraux de haut niveau d'une norme et son implantation technique par le biais d'un langage de programmation. Ce décalage ne peut être dépassé qu'on remplissant les trois conditions que nous avons présentées en introduction et que nous rappelons ici : (1). Adopter un processus de qualité reconnu, (2). trouver un model intermédiaire pour relier les attributs externes aux attributs internes rendant ainsi les attributs de qualité accessibles au programmeur, et (3). choisir un langage de programmation qui respecte (et fait respecter) les attributs internes de qualité.
Concernant la condition (1), il va sans dire que le choix d'un standard reconnu de qualité logiciel doit se faire en tenant compte des objectifs et des contraintes de chaque organisation. Nous avons déjà cité plus-tôt quelques standards et normes reconnus dans ce domaine. Par contre, nous ne nous intéresserons pas spécifiquement à ce sujet dans ce papier et nous référons le lecteur à [IEEE 94] pour une étude comparative entre les normes de qualité qui peut servir de point de départ.
Les conditions (2) et (3) seront abordées dans les prochaines sections qui suivent.

3. Rendre la qualité logicielle accessible au programmeur

Nous pouvons rendre la qualité accessible au programmeur en créant explicitement un lien entre les attributs de haut niveau (tels que

robustesse, réutilisabilté, efficacité, etc.) et les concepts techniques (tels que structure de données, variables, structure de contrôle, etc.) qu'utilisent le programmeur. Il faut établir pour chacun de ces attributs de haut niveau des procédés techniques qui permettent de l'atteindre et de le maintenir. En effet, les caractéristiques de qualité de bas niveau d'un logiciel affectent directement sa qualité de haut niveau [Fenton 91]. Si on arrive à associer à chaque attribut de qualité des procédés techniques et à demander, par la suite, au programmeur de respecter ces procédés, nous pourrions mieux impliquer ce dernier dans le processus de qualité**. Il deviendra, en fait, lui aussi responsable de la qualité du code qu'il produit.** En effet, si les attributs de qualité sont satisfaits dans chacun des procédés techniques et des composants structurels qui y sont impliqués, il y a de fortes chances que le code global les satisfasse aussi et, graduellement, nous assisterons à une amélioration des produits logiciels de l'organisation. Nous désignons ici par mot 'composant' toute entité élémentaire de représentation (variables, objets, etc.) et de traitement (boucles, itérations, etc.) qui compose un programme. Le programmeur est beaucoup plus à l'aise avec ces concepts qu'avec les caractéristiques générales de qualité telles que les notions de maintenabilité, de robustesse, etc..

Pour créer ce lien explicite entre les attributs de haut niveau et les procédés techniques, nous proposons d'utiliser le modèle de Dromey [Dromey 94] qui a été conçu dans cette perspective et qui se distingue par sa préoccupation de rendre accessible au programmeur les concepts théoriques de qualité. En utilisant le modèle de Dromey, il devient possible d'appliquer une norme de qualité telle que ISO-9126 et de l'adapter aux besoins pratiques d'un programmeur.

Le modèle de Dromey consiste à décomposer le programme en composants techniques élémentaires et de trouver des attributs de qualité qui s'appliquent à chacun de ces composants. Un premier niveau de décomposition nous donne trois composants: les Structures de Représentation (**SdR**), les Structures de Traitement (**SdT**) et les Commentaires (**Cmt**). En effectuant une autre décomposition sur ces mêmes composants, on obtient d'autres composants de niveau élémentaire (ou de bas niveau):

SdT ::= Assignation ∪ Séquence ∪ Sélection ∪ Répétition ∪ Expression ∪ Module;

SdR ::= Littéral ∪ Constante ∪ Type ∪ Enregistrement ∪Tableau ∪ Objet ∪ Variable;

Chaque caractéristique de qualité de haut niveau dépend de plusieurs attributs de bas niveau de ces composants élémentaires. Ces attributs de bas niveau sont bien connus des programmeurs et sont relativement faciles à comprendre. Nous mettrons l'emphase

dans les paragraphes qui suivent sur les SdT, les SdR et les modules afin d'illustrer nos propos.

3.1. Les structures de représentation

Les SdR regroupent toutes les données manipulées par le programme quels que soit leur nature et leur type. Une SdR d'un programme doit être:

- **Assignée**: possède une valeur particulière qui lui est attribuée avant son utilisation;
- **Initialisée**: possède une valeur particulière qui lui est attribuée avant son entrée dans une boucle;
- **Précise**: dispose d'un espace physique suffisant en concordance avec son type conceptuel;
- **Consistante**: a uns sens précis et agit à l'intérieur d'un espace circonscrit;
- **Directe**: dispose d'une représentation symbolique qui reflète sa signification sémantique;
- **Indépendante du rang**: ses bornes ne sont pas délimitées par des constantes;
- **Documentée et auto descriptive**: est documentée lors de sa déclaration et est commentée lors de son utilisation;

3.2. Les structures de traitement

Une SdT est une opération ou une instruction (ou une combinaison des deux), qui vise à réaliser une action particulière en manipulant des structures de représentation. Chaque structure de traitement doit être:

- **Progressive**: incrémente systématiquement sa variable de contrôle (boucle et/ou récursivité),
- **Consistante**: dispose d'un sens unique dans un domaine de définition précis,
- **Structurée**: possède une seule entrée et une seule sortie sans recours aux courts-circuits;
- **Résolue**: les traitement qu'elle effectue correspondent bien aux types de données qu'elle manipule;
- **Effective**: n'effectue pas de traitements inutiles (par exemple, une expression de la forme X+1-1);
- **Non redondante**: dispose d'une structure computationnelle cohérente;
- **Directe**: dispose d'une représentation symbolique qui reflète sa signification sémantique;
- **Documentée**: est documentée et commentée tout au long de son utilisation;

3.3. Les modules

Un module est un ensemble d'instructions qui réalise une sous tâche particulière de la tâche globale. Un module est lui-même un sous-programme autonome et doit, de ce fait, respecter tous les attributs de qualité requis pour un programme quelconque. De plus, chaque module doit être:

- **Paramétré**: toutes ses données d'entrée sont fournies en paramètres;
- **Encapsulé**: n'utilise aucune variable globale;
- **Cohésif**: les relations computationnelles entre ses différentes SdR sont claires et pertinentes;
- **Générique**: effectue une tâche précise qui ne dépend pas d'un type particulier de SdR;
- **Abstrait**: sa tâche est intellectuellement compréhensible à l'extérieur du contexte computationnel.

Malheureusement, tous les langages de programmation ne contribuent pas nécessairement à l'atteinte de ces attributs de qualité. Pour exiger d'un programmeur de respecter les attributs de qualité, il faut d'abord mettre à sa disposition les outils de programmation appropriés. *Ces outils doivent non seulement supporter les attributs de qualité, mais aussi les encourager et les favoriser*. Par exemple, on ne peut pas demander à un programmeur d'adopter une approche modulaire s'il utilise une version originale de BASIC où la notion de module n'existe pas. De la même façon, on ne peut lui exiger de bannir les effets de bord s'il travaille avec le langage C++ qui encourage implicitement et structurellement les effets de bord.

4. Langage de programmation et qualité logicielle: les cas de Delphi et de C++

Nous tenterons de démontrer dans cette section que le choix du langage de programmation joue un grand rôle dans l'atteinte des objectifs de la qualité logicielle. En effet, il est primordial que d'utiliser un langage qui respecte les normes de qualités et qui les fait respecter en cas d'abus de la part des programmeurs.
Nous présentons ici une étude comparative où nous avons mis à l'épreuve de la qualité logicielle des langages populaires et répandus :

- **Delphi** [Cantu 97] parce que le langage de programmation sur lequel il est basé (Pascal) [Koffman 97] se veut un langage d'apprentissage de l'algorithmique et de la programmation. Il a été en fait conçu dans l'esprit d'un programmation claire,

structurée et compréhensible, le but ultime étant de produire un code «intellectuellement gérable» à tout moment.

- **C++** [Murray 93] pour des raisons évidentes découlant de la popularité de ce langage dans l'industrie du logiciel.

Plusieurs concepts fondamentaux en Informatique ont été explicités, exploités et mis au service de ces deux langages de programmation. Ces concepts couvrent principalement la théorie des structures de données (représentation et sémantique), les mécanismes de passage de paramètres, l'abstraction, l'encapsulation et la modularité. Nous pensons que le langage Pascal (et par conséquent l'outil Delphi) possède intrinsèquement des caractéristiques qui favorisent la réalisation d'un code de qualité. De l'autre côté, nous croyons que le C++, bien que très puissant et populaire, permet de par sa flexibilité et ses (passes croches!!!), de passer à côté de certaines règles fondamentales du génie logiciel et de la qualité logicielle.
La prochaine section montre comment Delphi et C++ supportent et encouragent/découragent les attributs de qualité du modèle de Dromey et comment on y exploite les ingrédients pour réaliser un code de qualité.

4.1. Structures de représentation: constantes et variables

Delphi	C++

Const
Limite = 1000;
Sentinelle = non;
Var
r : Real;
i, j : Integer;
Star : Char;
Valide : Boolean;
Begin
...
End;
	{
	Double r = 5.;
	Int i = 0, j = i+1;
	i = j = 2;
	Const Double Pi = 3.14;
	Char Star = '*';
	...
	}

Tableau 1: les variables et les constantes en C++ et en Delphi

En Delphi, toute variable ou constante doit être déclarée au début du module avant la section des traitements (voir tableau 1). Tout oubli volontaire ou involontaire dans la déclaration des variables utilisées entraîne une erreur de compilation. Cela a plusieurs impacts positifs sur la qualité du code produit. D'abord parce qu'il oblige le programmeur à mieux réfléchir aux structures de données utilisées et donc à mieux les concevoir. Ceci favorise le respect de l'attribut **Précision** mentionné dans le modèle de

Dromey. Par la suite, la déclaration d'une variable, suppose que le programmeur connaît déjà son utilité dans les traitements. L'usage d'une variable est donc circonscrit et limité aux finalités précises pour lesquelles elle était conçue. Ceci coïncide directement avec l'attribut **Consistance** mentionné dans le modèle de Dromey. Finalement, la déclaration d'une variable incite le programmeur à **l'initialiser,** à lui **assigner** une valeur et à **l'utiliser** effectivement dans les traitements.

En C++, la déclaration des variables et des constantes est laissée aux bons soins du programmeur (voir tableau 1). Le compilateur émet occasionnellement des Warnings quand les variables utilisées ne sont pas initialisées et/ou assignées. Généralement, quand le programme est compilé et exécuté, le programmeur a tendance à ignorer ces Warnings et à se concentrer sur l'optimisation du code. Au fur et à mesure que les erreurs apparaissent, il essaye de les corriger en consultant les Warnings. Plusieurs de ces erreurs peuvent être évitées si le programmeur avait mieux conçu ses structures de données et si l'outil de programmation ne lui permettait pas d'adopter de «mauvaises» pratiques. C'est d'ailleurs de là que vient le terme «permissif» qu'on attribue généralement au C++ du fait qu'il permet beaucoup d'opérations «dangereuses» aux niveaux sémantique et opérationnel. En conclusion, le manque de déclarations de variables et de constantes crée ou, du moins, contribue à créer plusieurs problèmes relatifs à la **Précision** des variables utilisées et à leur **Consistance** puisqu'elles n'ont pas une sémantique unique. Il va sans dire que le C++ permet les déclarations des variables et des constantes mais ceci n'est pas suffisant à notre point de vue. *Un outil de programmation ne doit pas simplement supporter passivement des attributs de qualité mais il doit aussi les favoriser, les encourager et empêcher toute mauvaise pratique qui peut les affecter.*

4.2. Structures de représentation: les tableaux

Delphi	C++
Type Jours = (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche) Var Tableau: Array [1..32, 1..25] of Integer; Agenda : Array [Jours] of Integer;	Int Tableau[32][25];

Tableau 2: les tableaux en C++ et en Delphi

La structure de tableau est l'une des structures les plus utilisées en programmation. Elle permet l'automatisation des traitements sur des collections homogènes de variables.
En Delphi, lors de la déclaration d'un tableau, il faut indiquer explicitement la borne inférieure et la borne supérieure du tableau (voir tableau 2). Ceci implique qu'on peut définir des tableaux **indépendamment du rang** de l'indice en paramétrisant la borne supérieure et la borne inférieure. En ce qui concerne l'indice du tableau, Delphi exige qu'il soit ordinal mais ne le limite pas au type entier. L'indice peut être défini par le programmeur comme étant de type énuméré ou intervalle de données selon son besoin actuel.
Par exemple, s'il veut programmer un Agenda hebdomadaire pour automatiser l'emploi du temps, il peut utiliser la déclaration illustrée dans le tableau 2. Ainsi, plutôt que d'adresser le troisième élément de l'agenda par « Agenda[3] » , il peut l'adresser par « Agenda[Jeudi] » . Cela permet d'être **Directe** dans les traitements d'un tableau. En fin, mentionnons que Delphi vérifie systématiquement pendant la compilation et pendant l'exécution , les bornes inférieure et supérieure des tableaux et signalent tout adressage hors limites. Cela permet d'éviter des mauvaises surprises et contribue à une bonne **Calculabilité de l'indice du tableau**.
En C++, l'indice est toujours de type entier et ne peut pas être défini par l'usager. De plus, la borne inférieure doit être toujours égale à 0. Ceci ne permet évidement pas d'atteindre l'attribut de **d'Indépendance du rang** et empêche d'être **Direct** dans les traitements d'un tableau. Par ailleurs, C++ n'effectue aucune vérification des bornes du tableau ni en exécution ni en compilation.
Les manipulations des tableaux en C++ violent donc plusieurs attributs de qualité (**Précision**, **Directe**, **Indépendance du rang**, **calculabilité de l'indice**) et *représentent un exemple typique d'un outil qui ne permet pas de supporter les attributs de qualité.*

4.3. Structures de représentation: transtypage

Delphi	C++

Var
  x : Real;
  n : Integer;
  ch: Char;
...
n := Round(x) ;
n := Trunc(x) ;
n := Ord(ch) ;
ch := Chr(n) ;
ch := Succ(ch) ;
ch := Pred(ch) ;
	Int n, p, q;
	Double x;
	Char ch;
	
	...
	x = Double(p) / Double(q);
	n = Int(x);
	n = Int(ch);
	ch = Char(n);
	ch = ch + 1;
	ch = ch – 1;

Tableau 3: le transtypage en C++ et en Delphi

Par définition, le transtypage permet de faire des assignations entre des variables de types différents. Ceci peut entraîner inévitablement des conflits de formats de données et affecter la **Précision** des traitements. Plus l'outil est ouvert au transtypage, plus le risque de violer l'attribut **Précision** est grand.
En Delphi, le seul transtypage implicite possible est l'assignation d'un entier à un réel ou d'un caractère à une chaîne de caractère. Ceci est intrinsèque à la théorie des ensembles puisqu'un entier est aussi un réel et qu'une chaîne de caractères peut contenir seulement un caractère. La rigidité de Delphi envers le transtypage (voir tableau 3) permet d'éviter la violation de plusieurs attributs de qualité (**Consistance**, **Directe**, **Précision**) mais ne limite en rien ses capacités computationnelles de traitement.
Pour le transtypage explicite, Delphi a introduit le concept de type Variant. Ceci ne nous parait pas contradictoire avec la philosophie générale du langage Pascal. En fait, le type Variant concerne les objets dont on ne connaît pas le type. Et, à la déclaration d'une variable de type Variant, le programmeur s'engage explicitement à la traiter ainsi.
Nous pensons que le transtypage entre des variables typées est contre la logique formelle sur laquelle sont basés tous les traitements computationnels. Il est en fait difficile de donner un sens à une affectation de type 'a' ← 'a' - 2. Ceci possède pourtant un sens en C++ puisque l'opération d'assignation convertit automatiquement la partie droite au type de la partie gauche (voir tableau 3). Le compilateur se contente d'émettre un Warning si le format de codage de la partie gauche n'est pas suffisant pour contenir le type de la partie droite. En consultant le tableau 3, on comprend rapidement que la sémantique des opérations en C++ doit être revue de prêt. *En l'absence de la sémantique des opérations et des opérandes dans un langage de programmation, il est impossible de parler de Qualité logicielle dans le code qu'il produit.*

4.4. Structures de traitement: les modules

Delphi	C++

Procedure XYZ (arg1 : Integer; arg2 : char);
...
Begin
 	
End;
	Void XYZ (Int arg1, char arg2)
	
	{
	
	}

Function Somme(arg1, arg2 :Integer): **Real**;
...
Begin
  ...
  Somme:= <*expression*>;
End;
	Double Somme (Int arg1, Int arg2)
	{
	  ...
	  Return <*expression*>;
	}

Tableau 4: le modules en C++ et en Delphi

Un module est un ensemble de traitements qui visent à réaliser une sous tâche particulière et bien définie de la tâche globale du logiciel. Tous les attributs de qualité que le logiciel doit respecter doivent aussi être respecté dans le module. Un module est en fait lui-même un programme autonome qui communique avec les autres modules par le biais des paramètres. Il doit donc posséder ses propres variables (variables locales) et des paramètres formels. Il doit aussi avoir un output bien défini qui représente le résultat des traitements qui y sont effectués.
En Delphi, il y a deux façons d'implanter des modules (voir tableau 4). D'abord les procédures qui réalisent un des traitements sans retourner un résultat explicite (Affichage de menus, agencement des appels des modules, etc.) et les fonctions qui doivent obligatoirement retourner un résultat après les calculs. Dans les deux cas, le **genre et le nombre de paramètres sont formels**. Pour les fonctions, le **résultat est unique** et il est d'un **type formel non variant**. Ceci vise principalement la **Portabilité** des modules, leur **Réutilisation**, et assure un degré d'**Abstraction** assez élevé dans leur conception. C'est d'ailleurs, ces attributs de qualité qui, à notre avis, sont à la base du succès sans cesse croissant que connaissent les OCX et les ActiveX de Delphi.

En C++ tous les modules ainsi que le programme principal sont des fonctions (voir tableau 4). Ils retournent tous explicitement une valeur d'un type formel. Quand cette valeur n'est pas requise pour le reste des traitements, on utilise le type Void pour la fonction. L'utilisation massive du transtypage rend le typage formel des fonctions presque sans sens. De plus, **le genre et le nombre de paramètres peuvent changer** selon le besoin. Par exemple, dans la fonction Int(X) qui rend la partie entière de son argument, X peut aussi bien être de type numérique que de type alphanumérique. Si on l'assigne à une variable de type chaîne, son résultat sera automatiquement converti sous la forme d'une chaîne de caractères. Nous pouvons donc, ici aussi, voir plusieurs attributs de qualité violés tels que la **Précision**, la **Portabilité**, la **Réutilisation** et l'**Abstraction** dans la conception des modules.

4.5. Structures de traitement: les expressions arithmétiques

Delphi	C++
:= (Assignation) + – * / (division réelle) Div (division entière) Mod (reste de la division entière)	= (Assignation) + – * / (x = 2. / 3; implique que X =0.6667 x = 2 / 3 ; implique que X = 0) % (reste de la division entière).

Tableau 5: les expressions en C++ et en Delphi

En arithmétique, une expression est une suite d'opérateurs et d'opérandes possédant une certaine valeur en tout moment. Le calcul arithmétique suit des règles fondamentales qui établissent son harmonie et forment sa cohérence. L'une des règles fondamentales est l'utilisation d'opérandes qui appartiennent à des domaines de définition sémantiquement.

En Delphi, les règles élémentaires d'arithmétique sont respectées. Le seul entremêlement possible dans les expressions s'effectue entre les types dits homogènes, i.e., les chiffres entre eux et les caractères entre eux. Ceci permet d'être **Direct** dans la manipulation des expressions arithmétiques, d'assurer la **Précision** des traitements et de conserver la **Sémantique** de l'arithmétique.
En C++, les expressions arithmétiques entremêlent tous les types de données prédéfinis. Aussi, peut-on associer une sémantique (un peu particulière bien sûr!) à presque toute expression même celles de la forme «Char=Char*5» ou «Char=Char-1» . Ceci viole la **Sémantique** des opérateurs et met en jeu la **Précision** des traitements. Par ailleurs, en C++, les opérateurs agissent selon le type des opérandes traités. Ainsi, les résultats de «2./3» et «2/3» sont complètement différents (voir tableau 5). Encore là, il y a violation claire de la **Sémantique** des opérateurs et de **l'Abstraction** au niveau de la conception. Finalement, un autre gros problème du C++ concerne l'opération d'assignation en tant que telle, qui est considérée comme une expression et non comme une instruction. Ceci permet d'effectuer des opérations d'assignation pendant l'évaluation d'une expression et crée inévitablement des **effets de bord**. *Toute l'harmonie des traitements entre modules (et dans chaque module) peut être sérieusement atteinte.*

4.6. Structures de traitement: opérateurs arithmétiques composés

Les expressions arithmétiques composées sont des énoncés courts qui permettent de réaliser plusieurs opérations consécutivement (voir tableau 6). L'ordre de l'exécution de ces opérations dépend de la forme de l'expression utilisée. Le tableau 6 montre des exemples d'expressions composées et illustre bien le niveau de difficulté requis pour leurs compréhension et évaluation. C'est à notre avis beaucoup plus une source de confusion que d'efficacité computationnelle. D'ailleurs, ces expressions ne procurent aucun gain significatif au niveau de l'efficacité et favorisent implicitement la propagation d'effets de bord. Elles ressemblent curieusement à des instructions de type assembleur et s'éloignent considérablement des formes des langages de troisième et quatrième générations.
En consultant le tableau 6, on se rend compte que ces expressions violent beaucoup d'attributs de qualité et *qu'elles ne doivent simplement pas être utilisées si la qualité logicielle est réellement un souci.*

Delphi	C++
N'existent pas	a++ <=> a = a + 1; b = a++; <=> {b= a ; a= a+1;} b = ++a; <=> {a= a+1; b= a;}
	a--; <=> a = a - 1; b= a--; <=> {b= a ; a=a-1;} b= --a; <=> {a=a-1 ; b= a;}

N'existent pas	a += b; <=> a = a + b;
	a - =b; <=> a = a - b;
	a *= b; <=> a = a * b;
	a /= b; <=> a = a / b;
	a %= b; <=> a = a % b;

Tableau 6: les opérateurs composés en C++ et en Delphi

On a l'impression qu'on assiste à la destruction de C++.

4.7. Structures de traitement: opérateurs relationnels et logiques

Théoriquement, les opérateurs relationnels et logiques doivent produire un résultat booléen. Or, le type booléen n'existe pas en C++ ce qui nous fait croire que l'implantation interne des Guards n'est pas très claire et qu'elle peut causer des problèmes d'ordre logique.
Par ailleurs, les symboles utilisés en Pascal sont plus simples et intuitifs (voir tableau 7). Ils sont **Auto-descriptifs** et permettent de comprendre les opérations qu'ils représentent. Ceci

n'est malheureusement pas le cas en C++, où les symboles utilisés sont très spécifiques et **non Auto-descriptifs**.
Une autre différence entre Delphi et C++ est celle qui concerne l'évaluation des expressions. En Delphi, on utilise par défaut l'évaluation complète des expressions logiques tandis qu'en C++ c'est l'évaluation écourtée qui est effectuée par défaut. Nous n'avons pas pu identifier les conséquences de cette différence sur le plan de la qualité.

Delphi	C++
= <> < <= > >= And Or Not	== != < <= > >= && || !

Tableau 7: les opérateurs logiques et relationnels en C++ et en Delphi

4.8. Structures de traitement: Case / Switch

Delphi	C++

Case <expression> Of
 	
<Valeur1>: <Instruction1>;
.
 	
<ValeurN>: <InstructionN>;
 	
End;
	Switch (<expression>)
	{
	Case <Valeur1>:
	<Instruction1>;
	
	Case <ValeurN>:
	<InstructionN>;
	}

Tableau 8: la structure Case en C++ et en Delphi

Au niveau conceptuel, la structure de Case est utilisée pour éviter des imbrications multiples de la structure If Then Else. Donc, seulement le cas positif devrait être exécuté et les autres devraient être ignorés. Ceci est le cas en Delphi (voir tableau 8). En cas de besoin, on peut une séquence pour exécuter plus d'une instruction pour un même cas. Ceci permet une bonne **Structuration** du code, une **Clarté** accrue, et un accès **Direct** au cas concerné.
En C++, la structure Switch est très astucieuse et demande beaucoup d'attention. En effet, après l'exécution du cas positif, tous les cas qui le suivent seront aussi exécutés. De plus, le

compilateur effectue un retour au début de la structure Case si la variable de contrôle est affectée et réévalue tous les autres cas. S'il y a un cas positif, il ré exécute le traitement et continue avec le même cheminement. L'utilisation des séquences {...} n'est pas permise et il faut recourir explicitement à l'instruction Break si on a plus qu'une instruction pour un même cas.
La structure de Case en C++ est très mal Structurée et doit être revue complètement. Elle est fondamentalement **Redondante**, **non Claire** et **non Directe**.

5. Conclusion

Dans cet article, nous avons présenté que l'outil de développement joue un rôle primordial dans l'atteinte des objectifs de la qualité logicielle. Nous pensons que le choix de l'outil de développement est une décision stratégique qui doit être bien planifiée et en concertation avec les responsables de l'assurance qualité dans l'organisation (s'il y en a !). En fait, les répercussions de ce choix sont majeurs et peuvent affecter sérieusement les objectifs de qualité fixés.
Nous avons aussi expliqué que le Modèle de Dromey est directement applicable en contexte de développement (programmation) et qu'on peut donc viser la qualité tout au long de la phase de développement d'un logiciel. Le programmeur se verra attribué une plus grande responsabilité et devra s'adapter et orienté son code vers la qualité. Ceci demande un grand changement dans la culture Informatique mais nous pensons qu'il nécessaire de le faire (et le plutôt possible) si on veut garantir un meilleur avenir à cette profession.
L'outil de développement mis à la disposition des programmeurs doit être adéquat et cohérent avec les objectifs de qualité. Nous pensons, et nous l'avons argumenté dans ce travail, qu'un langage de programmation comme Pascal dispose de caractéristiques intrinsèques qui encouragent et favorisent la production d'un code de qualité. L'outil Delphi offre un environnement de développement riche pour le codage de systèmes informatique tout en maintenant et en soutenant les attributs de la qualité logicielle qu'il hérite du Pascal.

Références Bibliographies

[Cantu 97] : Delphi 3, Marcu Cantu, SYBEX 1997.

[Crosby 79] : Quality is free, Peter Crosby, McGraw-Hill, New-York, 1979.

[Dromey 94] : A model for software product quality, R.G. Dromey, Griffith University, Australia 1994.

[Fenton 91] : Softwre Metrics-A Rigorous Approach, Chapman and Hall, 1991.

[Humphry 95] : A Discipline for Software Engineering, Watts S. Humphrey, Addison-Wesley 1995.

[IEEE 94] : Software Engineering Standard Collection, IEEE Press, 1994.

[Koffman 97] : PASCAL, Eliot B. Koffman, Addison-Wesley 1997.

[Murray 93] : C++ Strategies and Tactics, Robert B. Murray, Addison-Wesley 1993.

[Sommerville 98] : Software Engineering, Ian Sommerville, Addison-Wesley, 1998.

Revue
D'Information
Scientifique &
Technique

Centre de Recherche sur l'Information Scientifique et Technique

FORMULAIRE CONFIDENTIEL D'EVALUATION D'UN PROJET DE PUBLICATION

N° de code : ……
Date d'envoi : ……
Nom du référé : ……
Intitulé : ……
……

EVALUATION DE L'ARTICLE

Mettre une croix dans l'une des cases suivantes :
(A – très bien, B- Bien, C- Assez bien, D- Passable, E- Médiocre)

	A	B	C	D	E
01- Originalité du travail ……..……..	☐	☐	☐	☐	☐
02- Pertinence par rapport à la connaissance scientifique dans le domaine ……..……	☐	☐	☐	☐	☐
03- Fondements théoriques et hypothèses…. …….	☐	☐	☐	☐	☐
04 – Méthodologie : (description adéquate et appropriée des matériaux et des méthodes)……..	☐	☐	☐	☐	☐
05-Adéquation hypothèses / résultats ……..……..…..	☐	☐	☐	☐	☐
06- Qualité des références bibliographiques…..…… (75% des références doivent dater de la dernière décennie)	☐	☐	☐	☐	☐
07- Qualité et clarté des tableaux et schémas…..……..	☐	☐	☐	☐	☐

08- Maîtrise de la langue utilisée…… ☐ ☐ ☐ ☐ ☐

09- Style…… ☐ ☐ ☐ ☐ ☐

Avis du référé :
(cocher l'une des cases suivantes)

- Accepté dans la forme présentée ……..…….. ☐
- Accepté après révision mineure …… ☐
- Accepté après révision majeure …….. ☐
(Assez long. Le manuscrit doit être condensé davantage).
- Rejeté …… ☐

Signature du référé **Date** : ……

Commentaires destinés aux auteurs **N° de Code** :
Date :

Outre des commentaires sur l'intérêt scientifique et l'originalité de l'article, vous pouvez exprimer vos recommandations sur la présentation, le style employé, l'organisation de l'étude, donner des indications pour réduire éventuellement l'article, l'adéquation et la disponibilité des références bibliographiques,… Préciser les éventuels points qui doivent faire l'objet de modification ou d'amélioration pour que le thème soit publiable.
(Utilisez éventuellement une feuille supplémentaire) Attention ne pas signer ce document.

……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……

Commentaire destinés au Rédacteur en Chef

N° de code : ……..

(Facultatifs)

Ces commentaires doivent être considérés comme confidentiels et ne peuvent être communiqués aux auteurs. (Utilisés éventuellement une feuille supplémentaire)

……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……

Signature eu référé **Date** : ……..