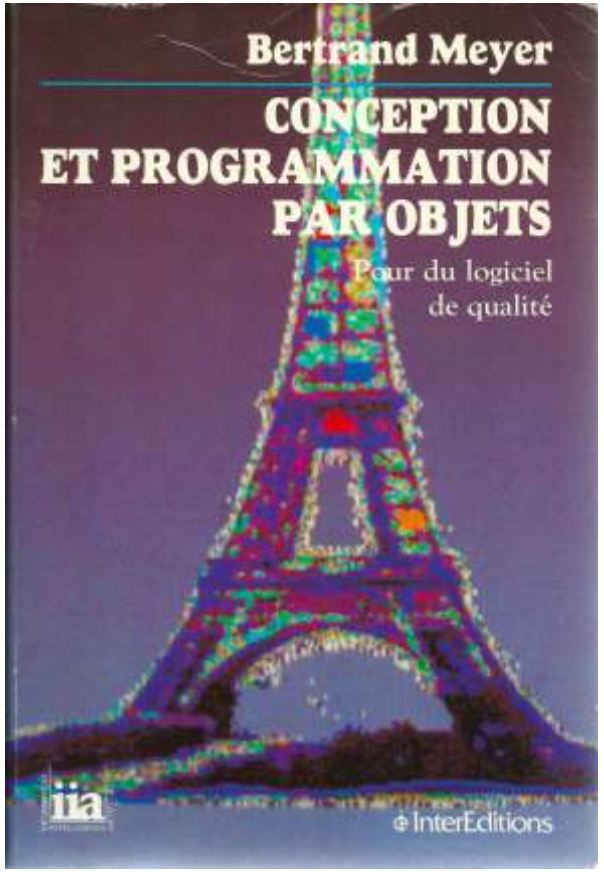
Programmation orientée objet dans VFP

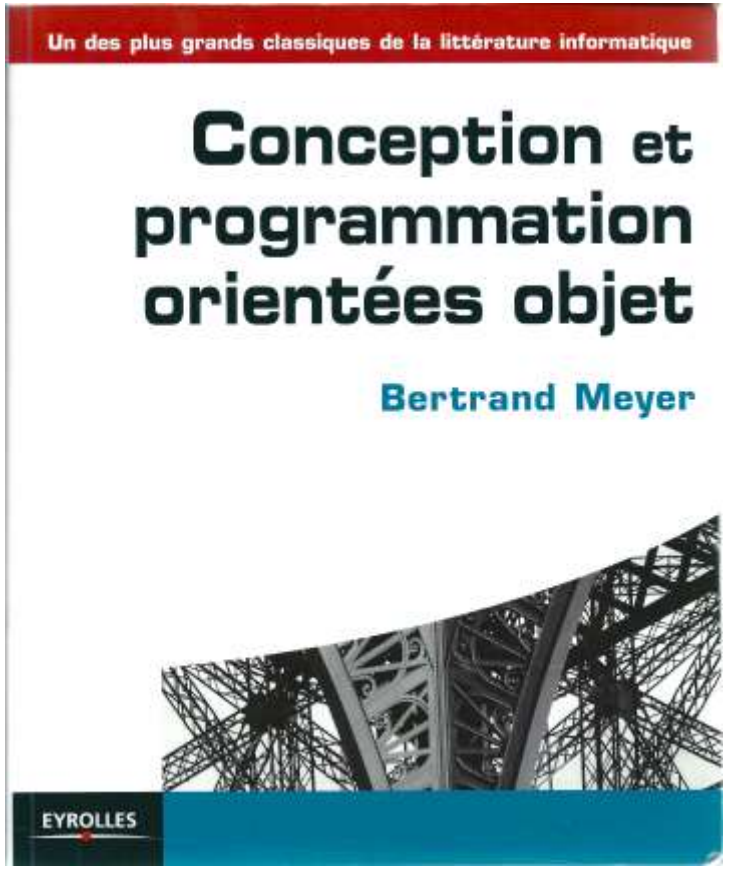
QUELQUES CONCEPTS FONDAMENTAUX ET LEURS APPLICATIONS DANS VFP

Avertissement : qui suis-je ? d'où viens-je ?

- 30 années de développement de progiciels de gestion dans le cadre de sociétés éditrices de logiciels
- Gestion de bases de données de volume important
- Interface utilisateur riche, intuitive et conviviale
- Pratique professionnelle de Delphi (version 3) et VFP (essentiellement version 6)
- Lecture, et vaguement écriture, de Java, Python, Perl et C#

L'ouvrage de référence de la POO





Un autre ouvrage (plus didactique)

- La programmatio orientée objet (cours et exercices en ULM2 avec Java 6, C# 4, Python, PHP 5 et LinQ
- Hugues Bersini, Editions Eyrolles (5ème édition - 2011)

Préambule (pour éviter les confusions)

- Nos environnements de développement privilégient les notions de formulaires et d'événements
- Trois niveaux : concepts POO, programmation événementielle, formulaire comme conteneur ultime
- Exemple : la propriété *Parent*
- Dans les exemples de cette session, les méthodes ne font rien et les propriétés sont vides

Introduction : la POO, qu'est-ce ?

- Une méthode d'analyse, de conception et d'implémentation centrée sur les données (quelques exemples tirés de la gestion)
- Une classe (ou type) est un ensemble structuré de données répondant à des messages
- Trois principes : l'encapsulation, l'héritage, le polymorphisme
- La classe instancie un objet sur lequel pointent une ou plusieurs références

Introduction : l'encapsulation

- Une entité : la classe ou le type
- Des champs, attributs ou propriétés
- Des méthodes, procédures ou fonctions
- Protéger les données par la rétention de l'information : modificateurs de visibilité et accesseurs
- Autre forme de protection des données : organisation du code, packages

Introduction : l'héritage

- Application à la programmation du principe de moindre effort
- Une classe dérivée (ou sous-classe) hérite des méthodes et propriétés de sa classe parente (ou superclasse)
- Elle définit des méthodes et propriétés correspondant à sa spécialisation
- Ne pas confondre héritage et composition (la propriété d'une classe est un objet d'une autre classe)

Introduction : le polymorphisme

- Surcharge de méthode : un même nom, plusieurs signatures (exemple : la méthode DrawingText présentée dans C# par Cesar hier matin)
- Redéfinition : une sous-classe implémente à nouveau une méthode définie dans une classe parente
- Ad hoc : un même nom pour des classes indépendantes (exemple *Refresh*)

Introduction : les objets

- La classe est un texte statique dans le code
- Un objet est l'instance d'une classe et possède un cycle de vie
- Les variables qui référencent des objets sont des pointeurs vers la zone mémoire stockant le contenu de celui-ci
- Un objet est l'instance de sa classe effective mais aussi des parents de celle-ci (il peut être manipulé en étant sous-classé)

Introduction : ce dont on ne parlera pas

- Les classes, méthodes, propriétés abstraites ou virtuelles
- Les interfaces
- Les méthodes et les propriétés statiques ou de classe
- Les classes génériques

Et VFP dans tout ça ?

- Les moins : le typage (*01-type*) et au moins un bug (*hidden*)
- VFP trop procédural pour faire de la POO ?
- VFP implémente un ensemble cohérent qui assume le cœur POO dont nous avons besoin pour des progiciels de gestion de données

Première étape : définir une classe

- Pas de classe racine dans VFP
- Liste des classes de base dans l'aide : *Reference | Language Reference | Object, Collections, and Classes*
- Classe *Custom* comme classe de base de mes exemples
- Une première expérimentation : *02-classe1*
- Deux sections : les propriétés et les méthodes
- Trois niveaux de visibilité : public (*rien*), protected (*protected*) et private (*hidden*)
- Instanciation d'un objet par *CreateObject* ou *NewObject* (afin de préciser le module comprenant la classe à instancier)

Première étape : les accesseurs

- Comment protéger l'accès aux données avec les méthodes *_assign* (set) et *_access* (get)
- Application avec la classe précédente : *03-classe1*
- Dans VFP, pour protéger efficacement une donnée il faut la définir comme publique !

Première étape : THIS

- La classe fait référence à l'objet courant par le mot-clé *THIS*
- L'usage de ce mot-clé est impératif (voir *04-classe1*)
- Attention : ne pas confondre une propriété et une variable locale de même nom

Deuxième étape : une sous-classe

- Avertissement linguistique : parent versus owner
- Avec *05-classe2* la spécialisation de Classe1 en Classe2
- Classe2 hérite des propriétés et méthodes de Classe1 mais ne peut manipuler que les attributs publics et protégés
- Avec 06-classe2 nous voyons comment implémenter le polymorphisme
- La méthode d'une classe peut appeler la méthode de son parent avec *Dodefault()*
- La méthode d'une classe peut appeler la méthode d'un grand-parent avec l'opérateur de résolution de portée ::

Vie et mort d'un objet (phase un)

- Pour visualiser le cycle de vie d'un objet nous allons redéfinir les méthodes *Init* et *Destroy*
- À toute classe nous ajoutons la méthodes *Release* dont le code est *Release This*
- Un objet est instancié soit avec *CreateObject* soit avec *NewObject*
- Avec *08-classe1*, quatre modes de désaffectation d'une variable objet
  - Release loObjet1 → loObjet1 n'existe plus l'objet pointé est détruit
  - loObjet1=Null → loObjet1=Null l'objet pointé est détruit
  - loObjet1='' → loObjet1='' l'objet pointé est détruit
  - loObjet1.Release() → loObjet1=Null l'objet pointé est détruit

Vie et mort d'un objet (phase deux)

- *Release loObjet1* et *loObjet1.Release()* exécutent à priori le même code
- Dans tous les cas l'objet pointé est détruit
- Avec *09-classe1*, nous observons comment les quatre modes de désaffectation réagissent en présence d'une seconde référence sur l'objet
  - Release loObjet1 → loObjet1 n'existe plus → loObjet2 pointe sur l'objet
  - loObjet1=Null → loObjet1=Null → loObjet2 pointe sur l'objet
  - loObjet1='' → loObjet1='' → loObjet2 pointe sur l'objet
  - loObjet1.Release() → loObjet1=Null → loObjet2=Null

Vie et mort d'un objet : Release loObjet1

- Le programme *11-classe1* illustre les différents cas où cette commande pourrait être utile
- Une instance globale qui doit se libérer avant la fin du programme courant
- Inutile dans le cas d'une variable locale : la sortie de la méthode provoque la destruction de l'objet
- Inutile aussi dans le cas d'une propriété de classe, qu'elle soit définie par *NewObject*, *CreateObject* ou *AddObject* : la destruction du propriétaire provoque la destruction de l'objet référencé par la propriété

Vie et mort d'un objet : loObjet1=Null

- Déréférence la variable loObjet1
- Le garbage collector détruit un objet quand aucune référence ne pointe plus sur lui (le programme *10-classe1* illustre ce phénomène)

Vie et mort d'un objet : loObjet1=''

- Quelle curieuse idée de vouloir transtyper une variable ! Mais il fallait bien voir ce que cela donnait.

Vie et mort d'un objet : loObjet1.Release()

- Cette méthode détruit l'objet
- Il positionne à Null toutes les variables qui référencent l'objet détruit
- Effet secondaire : une variable objet peut devenir null d'une seconde à l'autre

Et le programme 07-classe2 ?

- Où l'on voit qu'une variable cachée peut devenir visible par à peu près tout le monde

Et les Thisform, Load, Parent…

- Nous ne sommes plus vraiment dans la POO
- Thisform : une tentative de reprise du concept de délégation
- Load : une méthode qui vient interférer avec le processus de création
- Parent : ambiguë (bien que très pratique) car c'est au client d'utiliser les classes qu'il définit comme propriétés, mais ces classes n'ont à priori pas à invoquer leur client
- Si le temps le permet voir les programmes *12-parent*, *13-locale* et *14-composition* (et *15-paramètre* s'il existe)

Ce qui fait souffrir

- L'absence massive de classes non visuelles (listes, arbres...)
- L'impossibilité de décrire des classes génériques
- L'absence d'accès aux sources (comment comprendre le fonctionnement de ce qui est caché dans une boîte noire ?)

Conclusion (pourquoi développer en POO)

- Parce que c'est la mode
- Parce que la plupart des langages récents utilisent peu ou prou les concepts de la POO
- Parce que la POO rapproche nos objets logiciels des données que manipulent nos clients
- Parce que la POO permet une implémentation agréable de classes orientées métier
- **Donc : programmez objet, il en restera toujours quelque chose et, avec VFP, ce quelque chose est le cœur de votre métier**