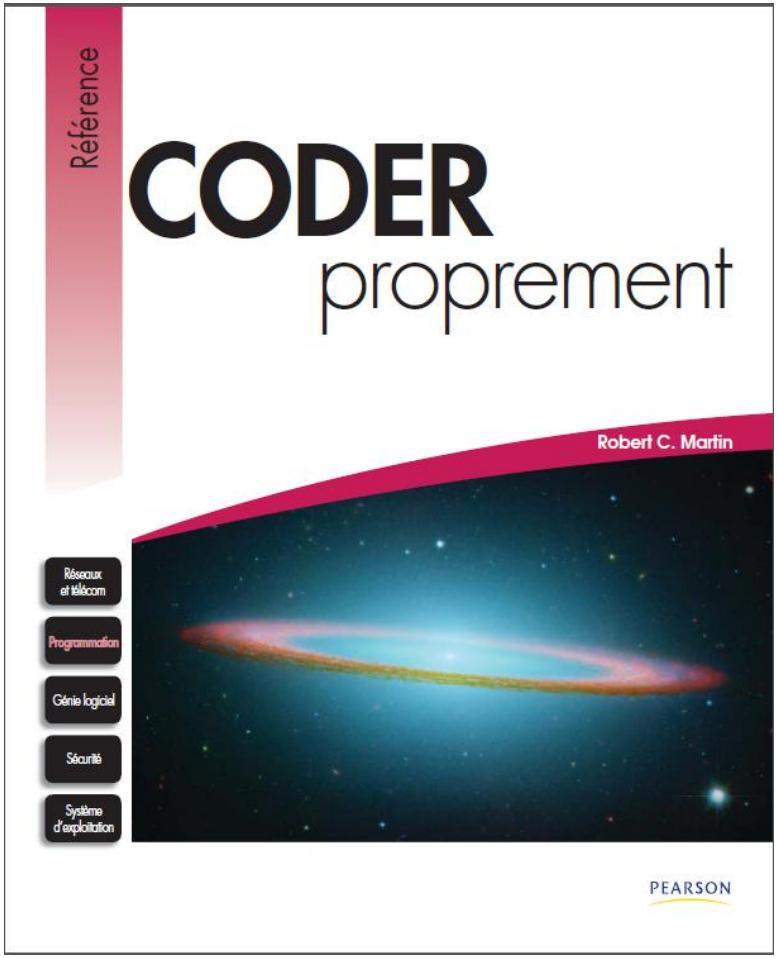
Un code propre pour les humains

Clean code for humans

Objet de la session

- Un code propre, à destination des humains
- Une bible des bonnes pratiques
- fondée sur l'expérience
- qui évolue avec le temps et les nouveautés
- plutôt issue du monde java / C
- … mais par certains côtés, critiquable

Le livre de référence



Les précédents

- Des sessions ont déjà concerné ce sujet

- En particulier celles de Thierry Nivelet / François Lepage en 2005 et 2007

« Optimiser ses développements VFP »

Eléments de réflexion

- Ecrire du code est facile, plus difficile est de le lire
- Technical Debt - la « dette technique » : le passif à combler, qui se rembourse avec les intérêts, lors des maintenances correctives ou évolutives.

Eléments de réflexions préliminaires

- Pour le patron / client, il faut aller vite et produire. Ne pas suivre, expliquer qu'il faut plus de temps si nécessaire : produire moins mais mieux.
- Ne soyez pas un verbe : « ça, c'est du code de Kévin », ou même, « là, tu as fait du Kévin », « ce code est complètement Kéviné! »

Le code propre sert de fondations

- Du code propre est le préalable au code robuste
- … Donc au code testable
- … Donc aux tests unitaires
- Il est nécessaire au Développement piloté par les tests,
- Il est nécessaire / accroché aux design patterns

Un développeur est un **auteur**

- Ecrire est sa profession, tout comme :
  - Quelqu'un qui construit un document électronique
  - Le concepteur d'un système
- Un auteur écrit pour être lu
- Un auteur « classique » a des outils :
- chapitres, paragraphes, figures de style, telles litote, métaphore, allitérations…
- Un développeur a des outils :
- langages, classes, méthodes et procédures…

La suite de la session…

- Une suite de principes : mais il faut faire la part des choses…
- Visual FoxPro n’est pas Eclipse, ni Visual Studio!
- Cependant, certains des principes suivants sont à méditer, à analyser, et souvent, méritent l’adhésion

Les grands principes

- **L'outil adapté à la tâche.**

Un bon outil peut parfois éviter le code. Si un outil existe, il faut l'utiliser.

- La tâche adaptée à l'outil

HTML, CSS, JavaScript,… ; VFP, C#, VB, ruby, python… ; T-SQL, PL-SQL… Ces différents aspects interagissent et s'interpénètrent… Mais chaque outil doit en principe s'occuper de ce qui le concerne : le code doit rester natif dans son propre environnement. Par exemple, éviter de générer du SQL dans du JS.

Les grands principes

- **Pas de bruit de fond : le bon code va à l'essentiel sans fioriture.**
- Le principe TED
  - **T**erse (laconique)
  - **E**xpressive (expressif)
  - **D**o one thing (ne fait qu'une chose)
- Lorsqu'on lit du code, notre cerveau est le compilateur: simplifions lui la tâche.

Les grands principes

- Le principe de moindre surprise

Le comportement d'une classe ou d'une fonction doit être « évident », c'est-à dire correspondre à ce qu'un autre développeur peut raisonnablement s'attendre à obtenir.

En général, la principale difficulté est de retrouver les intentions originales des programmeurs.

Les grands principes

- Le principe DRY : Don’t repeat yourself (ne vous répétez pas). Même principe que les règles de Codd : **éviter la redondance**. Mais c’est le cas de beaucoup de principes en programmation
- Moins il y a de lignes de code, moins il y a de bugs (dans le cas de code redondant).
- Chaque répétition de code représente une opportunité d’abstraction manquée.

Les grands principes

- Self-documenting code : le bon code se suffit à lui-même ; aucune documentation supplémentaire n’est nécessaire

Les grands principes

- Respect des niveaux d'abstraction : ne pas mélanger les concepts de niveau supérieur et ceux de bas niveau. Par ex, les éléments de l'implémentation détaillée ne doivent pas se trouver dans la classe de base - même si c'est plus commode au moment de l'écrire.

Les grands principes

- Le code mort: code jamais exécuté.

Soit parce que le système a évolué et le code n'est plus appelé, soit, par exemple, parce qu'on a prévu une condition qui ne se présentera jamais.

Quand on tombe sur un tel code, on se pose des questions pour rien. Si vous le décelez, offrez lui un bel enterrement, en le supprimant définitivement !

Les commentaires

- Selon certains, le commentaire est la panacée. Dans la plupart des livres sur les langages, la première instruction présentée est le commentaire.
  Est-ce vraiment exact ?
  - OUI!
  - … Mais on ne commente pas n'importe comment…

    Le commentaire doit apporter quelque chose de plus que ce que présente le code

Les commentaires

- **Commentaires inappropriés**, ou sans intérêt
- Ou parfois intéressants, mais ailleurs : gestion du code source, liste des bugs, auteurs…

(…)

```
oSplash = CREATEOBJECT("frm_Splash")
oSplash.Visible = .T.   && visible dès la création - modif 02/04/03 – JB
```

(…)

Les commentaires

- **Commentaires séparateurs**…
- Des lignes permettant de séparer en plusieurs parties de longues procédures : indice qu'il faut refactoriser, créer des sous-fonctions… (on y reviendra)

Les commentaires

- **Commentaires obsolètes**, à modifier ou à supprimer

(…)
```
***SCAN de la table pour trouver le mot de passe de
l'utilisateur
IF SEEK(ThisForm.txtLogin.Value)
```
(…)
Devient
(…)
```
***recherche du login de l'utilisateur
IF SEEK(ThisForm.txtLogin.Value)
```
(…)

Les commentaires

- Commentaires redondants, qui n'apportent aucune valeur ajoutée

```
LOCAL Var
Var = 1   && assigner 1 à Var

FUNCTION CT
* Calculer le montant des taxes

FUNCTION CalcTaxes
* Calculer le montant des taxes
```

(…)
***recherche du login de l'utilisateur
IF SEEK(ThisForm.txtLogin.Value)
(…)
Devient
(…)
IF SEEK(ThisForm.txtLogin.Value)
(…)

Les commentaires

- Commentaires d'excuse, d'avertissement, commentaires incompréhensibles, mal rédigés…

```
*J'étais vraiment crevé quand j'ai
  fait cette proc, faudra la revoir
*Gaffe à la variable lDanger !
*Les behav. des diff devices seront
  affect par la plq de conc. du
  mach.
```

Les commentaires

- Commentaires de code

(…)

```
*i = i + 1
*i = i + inc
*i = i + IIF(Flag, inc, 1)
i = m.i + IIF(m.Flag, m.inc, 1)
```

(…)

- TUEZ LE CODE ZOMBIE !
  Il gène la lecture, il empêche les recherches efficaces, on n'ose pas le modifier, on se pose des questions…

Le nommage

Le nommage est la pierre angulaire du code propre !

Nommer les variables

La problématique par un exemple : que fait cette fonction ?

```
FUNCTION CTOH (x, s)
   IF EMPTY(m.s)
       s = ":"
   ENDIF
RETURN LEFT(m.x,LEN(m.x) - 2) + m.s + RIGHT(m.x,2)
```

Le nommage

La même chose que celle-ci :

```
FUNCTION CTOH (cHeure, cSeparateur)
    IF EMPTY(m. cSeparateur)
            cSeparateur = ":"
    ENDIF
RETURN LEFT(m. cHeure,LEN(m. cHeure) - 2) + m. cSeparateur + RIGHT(m. cHeure,2)
```

```
FUNCTION CTOH (x, s)
    IF EMPTY(m.s)
            s = ":"
    ENDIF
RETURN LEFT(m.x,LEN(m.x) - 2) + m.s + RIGHT(m.x,2)
```

Quelques considérations sur le nommage des variables ou des constantes :

- Plus la portée de la variable est grande, plus son nom doit être spécifique
- Eviter les suffixes, la notation hongroise (nota : cette règle s'applique plutôt dans les environnements récents…)
- Eviter les abréviations : il n'y a plus de course à l'octet.
- Pour les variables booléennes, montrer dans le nom qu'il y a un état vrai/faux : pas Open, mais IsOpen
- Quand des variables ou des constantes montrent deux états différents, être symétrique : pas Allumé/Off, mais Allumé/Eteint ou On/Off, pas Min/Rapide mais Min/Max ou Lent/Rapide…

Le nommage

Nommer les fonctions

- Attention aux noms des méthodes qui indiquent le moment et pas ce qu'elles font (Load, Init…). Ce sont des nommages corrects pour les frameworks, mais pas un exemple à suivre dans nos développement.
- Principe de moindre surprise : le lecteur doit comprendre ce que fait la méthode en lisant. Il ne doit pas être surpris s'il va dans le code. A l'extrême, le lecteur ne devrait jamais avoir besoin de consulter l'implémentation.
- Par exemple, que fait la ligne suivante ?

```
DateLimite = DateAjoute(DateLimite, 5)
```

- Ajoute 5 jours à DateLimite? 5 semaines ? 5 mois ? Il faut le préciser.

Le nommage

- **Principe de responsabilité unique** : le nom d'une méthode doit annoncer ce qu'elle fait, et la fonction ne doit pas faire plus que ce qu'elle annonce.
- Attention quand la fonction contient « ET », « OU », « SI »… => cela signifie souvent que la fonction fait plusieurs choses.

  Exemple : « VerifierPassWord » ne devrait pas mettre à jour une log.

```
FUNCTION VerifierPassWord(Login, PassWord)
   (…) des tests divers de vérification, puis
   PassWordOk = m.PassWordOk + LogUser(Login)
RETURN m.PassWordOK
```

  Devrait plutôt ressembler à

```
IF VerifierPassWord(Login, PassWord)
    LogMaj=LogUser(Login)
ENDIF
```

Variables, constantes, affectation et utilisation

- Le point essentiel reste la lisibilité. Il faut favoriser tout ce qui facilite la lecture
- Par exemple, avec une variable booléenne

```
IF UserAccredite=.T.
```

=.T. allonge la lecture inutilement

```
IF UserAccredite
```

Variables, constantes, affectation et utilisation

- Déclarer les variables booléennes implicitement

```
LOCAL MacDoPossible
IF Monnaie > 6
    MacDoPossible = .T.
ELSE
    MacDoPossible = .F.
ENDIF
```

```
LOCAL MacDoPossible
MacDoPossible = .F.
IF Monnaie > 6
    MacDoPossible = .T.
ENDIF
```

```
LOCAL MacDoPossible
MacDoPossible = Monnaie > 6
```

Avec FoxPro, ça reste assez court, mais avec d'autres langages, tels c# ou delphi, une déclaration peut prendre rapidement une importante place verticale

Variables, constantes, affectation et utilisation

- Eviter les doubles négations

IF !IsNotLogged

devient

IF IsLogged

- Utiliser l'opérateur ternaire est élégant !

```
LOCAL MontantAPayer
IF Invité
   MontantAPayer = 0
ELSE
   MontantAPayer = 10
ENDIF
```

```
LOCAL MontantAPayer
MontantAPayer = 10
IF Invité
   MontantAPayer = 0
ENDIF
```

```
LOCAL MontantAPayer
MontantAPayer = IIF(Invité, 0, 10)
```

Variables, constantes, affectation et utilisation

- Utilisez des constantes (correctement nommées)

```
IF TypeEmploye = « Directeur »
IF TypeEmploye = K_TYPEEMPLOYE_DIRECTEUR
```

- Et éviter les nombres magiques

```
Resultat = Param * 19.6
Resultat = Param * 5280
Resultat = Param * 1440
Resultat = Param * 3.1415923765359
```

Variables, constantes, affectation et utilisation

- Eviter les nombres magiques

```
Resultat = Param * 19.6
Resultat = Param * 5280
Resultat = Param * 1440
Resultat = Param * 3.1415923765359

Resultat = Param * K_TVA
Resultat = Param * K_FEETPERMILES
Resultat = Param * K_MINUTESPARJOUR
Resultat = Param * K_PI
```

- Notes :
- la tva vient de changer ;
- Pi ne vaut pas `3.1415923765359`
  mais `3.14159265359`

Variables, constantes, affectation et utilisation

- Cacher la complexité de certaines conditions. Le code doit aussi être une auto-documentation

Se poser la question : que cherche t-on ?

```
IF !user.isRegistered ;
          and (File.Ext = « mpg » or File.Ext = « mp4 » or File.Ext= « avi ») ;
          and File.Duration<15
    && pour les utilisateurs non enregistrés,
          seule la version de présentation de la vidéo est accessible
    (…) envoi de la vidéo
```

Pour obtenir :

```
IF VideoAccessible(User, File)
(…) envoi de la vidéo

FUNCTION VideoAccessible(User, File)
RETURN !User.isRegistered ;
          and (File.Ext = « mpg » or File.Ext = « mp4 » ;
          or File.Ext= « avi ») and File.Duration<15
```

Variables, constantes, affectation et utilisation

- La déclaration des variables
  Les variables éphémères doivent être déclarées lorsqu'elles sont nécessaires, pas avant. Sinon, le lecteur a un travail de mémoire supplémentaire.
- Il faut éviter d'avoir à scroller.

Variables, constantes, affectation et utilisation

- Classes, switch, et polymorphisme

CASE doit si possible être remplacé par l'utilisation du polymorphisme
Exemple de classe qui, en raison de ses évolutions, entraîne le switching :

Etape 1 :
Classe AccesDB originale
- Connexion
- Log
- Déconnexion
- Etc.

Etape 2 :
Classe AccesDB modifiée
- Connexion
- ConnexionSQLSrv
- ConnexionOracle
- ConnexionPostgreSql
- Log
- LogSQLSrv
- LogOracle
- LogPostgreSql
- Déconnexion
- DéconnexionSQLSrv
- DéconnexionOracle
- DéconnexionPostgreSql
- Etc.
- EtcSQLSrv.
- EtcOracle.
- EtcPostgreSql.

Etape 3 : code parsemé d'instructions du genre :

```
FUNCTION Log
DO CASE
CASE cDb = « SQLSERVER »
     IF oAccesDB.ConnexionSqlSrv()
        oAccesDB.LogSqlSrv()
     ENDIF
CASE cDb = « ORACLE »
     IF oAccesDB.ConnexionOracle()
          oAccesDB.LogOracle()
     ENDIF
(…)
ENDCASE
```

Variables, constantes, affectation et utilisation

- Classes, switch, et polymorphisme

Etape 1 : classe « vide »

Classe AccesDB

-Connexion

```
MESSAGEBOX(« Connexion non implémentée ! »)
```

-Log

```
MESSAGEBOX(« Log non implémentée ! »)
```

- Déconnexion

```
MESSAGEBOX(« Déconnexion non implémentée ! »)
```

- Etc.

Etape 2 : classes implémentées

Classe AccesDBSql : AccesDB
- Connexion
- Log
- Déconnexion
- Etc.

Classe AccesDBOracle : AccesDB
- Connexion
- Log
- Déconnexion
- Etc.

Classe AccesDBPostgreSql : AccesDB
- Connexion
- Log
- Déconnexion
- Etc.

Etape 3 : instanciation

```
DO CASE
CASE cDb = « SQLSERVER »
    oAccesDB = CREATEOBJ(AccesDBSql)
CASE cDb = « ORACLE »
    oAccesDB = CREATEOBJ(AccesDBOracle)
(…)
ENDCASE
```

1 seul case au début, ensuite, il n'y a plus de question à se poser…

```
FUNCTION Log
     IF oAccesDB.Connexion()
          oAccesDB.Log ()
     ENDIF
```

Variables, constantes, affectation et utilisation

- « Table Driven Method »

Est-il nécessaire d'en parler ici ?

- Code en dur (en anglais, « Hard Coding »)

```
DO CASE
CASE Age<20
   Return 3.4
CASE BETWEEN(Age, 21, 29)
   RETURN 4.3
CASE BETWEEN(Age, 30, 45)
   RETURN 12
CASE BETWEEN(Age, 46, 65)
   RETURN 5
OTHERWISE
   RETURN 0
ENDCASE
```

Remplacer par une table et faire une recherche… Tout le monde ici le fait sans doute déjà !

Fonctions

- Quand les créer ?

➢ Eviter la duplication (DRY), *mais aussi*

➢ Eviter l'indentation (et la complexité)

➢ Exposer les intentions grâce à un nom bien choisi

➢ Atomiser les tâches

Fonctions

- Eviter la duplication (DRY)

Un des principes les plus anciens de la programmation, un des plus importants, déjà évoqué.

Fonctions

- Eviter l'indentation (et la complexité)

```
IF
    IF
         IF
                  DO quelque chose
         ENDIF
     ENDIF
ENDIF
```

Au delà de 3 niveaux, la compréhension s'éteint. Pour corriger :

- Extraction, (à l'image d'une note de bas de page);
- Emettre des RETURN dès que la réponse est connue (ou qu'une condition n'est pas remplie);
- Sortir dès que possible en utilisant Try… Catch et Throw

Fonctions

- Exposer les intentions grâce à un nom bien choisi

… point que nous avons déjà vu…

```
IF VideoAccessible(User, File)
```

Fonctions

- Atomiser les tâches
  Tout comme un livre est découpé en paragraphes, une fonction aide à la lecture, en découpant les tâches et les étapes.
  De plus, cela aide aux tests, aide à éviter les effets de bords, promeut la réutilisation...

Fonctions

- Les paramètres
  Plus il y a de paramètres, plus la fonction est compliquée à comprendre et à maintenir. De plus, c'est un *indice* que la fonction ne respecte pas le principe de responsabilité unique.
- C'est en particulier le cas des paramètres logiques, qui selon leur valeur, changent parfois totalement la façon dont la fonction agit.

Les fonctions

- Le principe de proximité
  Le lecteur lit de bas en haut : il faut essayer de mettre ensemble les fonctions liées entre elles.

Fonctions

- Longueur de la fonction
  Les indices de fonction trop longue :
- ➢ Obligé de scroller pour lire
- ➢ Obligé de scroller horizontalement
- ➢ Difficulté à trouver un nom
- ➢ Beaucoup de conditions
- ➢ Difficulté à lire et comprendre
- Selon Robert C. Martin :
- ➢ Rarement au-delà de 20 lignes
- ➢ Jamais plus de 100
- ➢ Pas plus de 3 paramètres

Fonctions

- Rendre physique les dépendances logiques ou temporelles (des étapes qui s'enchaînent dans un ordre précis doivent exposer ce comportement):
- Ex : 2 fonctions pourront paraître indépendantes alors qu'elles sont liées :

```
CreationTableCorrespondance()
FaireCorrespondre()
```

devient

```
Nb=CreationTableCorrespondance()
FaireCorrespondre(Nb)
```

Les classes

- Les classes suivent des principes similaires aux fonctions.
  Elles permettent la cohésion de l'ensemble.
  Elles sont l'équivalent des en-têtes dans les livres : parties, titres, chapitres, sections, paragraphes.

Les classes

- Permettent une abstraction supplémentaire
- Permettent d'être paresseux : la complexité est gérée, on ne s'en occupe plus.
- Mais il faut être rigoureux : une classe devrait être autonome et ne s'occuper que d'elle-même.

Les classes

- Une classe doit avoir un certain niveau d'abstraction, donc être stable. Elle n'a pas de raison de changer souvent.
  Soumise au principe de responsabilité unique, chaque classe doit être concentrée sur une fonction. Souvent, une classe trop complexe devrait être scindée en plusieurs.
- Par exemple :

- Classe Véhicule
  - Base
  - Options
  - Schéma électrique
  - Immobilisations
  - Financement
  - Calcul des loyers

- Classe Véhicule
  - Base
  - Options
- Classe technique
  - Schéma électrique
- Classe maintenance
  - Immobilisations
- Classe Financement
  - Financement
  - Calcul des loyers

Les classes

- Chaque clase doit avoir son niveau d'abstraction.
  Une sous-classe spécialise une classe parente.
- Donc, la classe parente ne devrait pas avoir à gérer les classes enfants. Elle n'est pas censée les connaître – alors que l'inverse est vrai.
- Parfois, ce principe ne s'applique pas, lorsque les classes dérivées correspondent à une énumération définie et fixe. Exemple du jeu de cartes : la classe « couleur » peut référencer ses sous-classes « trèfle », « carreau », « cœur » et « pique ».

Conclusion : la règle du boy scout

- Laissez toujours le code un peu plus propre que vous l'avez trouvé en entrant

  ;-)