V 1.0

SOLUTION D'ARCHITECTURE LOGICIELLE

MISE EN ŒUVRE DES DEVELOPPEMENTS POKEMON MASTER

Date :	**28 décembre 2017**
Rédacteur :	**Stéphane WICQUART**
Objet :	**Mise en œuvre des développements PokemonMaster**
PIECES JOINTES :	**-**

SOLUTION D'ARCHITECTURE LOGICIELLE

MISE EN ŒUVRE DES DEVELOPPEMENTS
POKEMON MASTER

SOLUTION D'ARCHITECTURE LOGICIELLE

MISE EN ŒUVRE DES DEVELOPPEMENTS
POKEMON MASTER

1. Besoin

Cette note décrit les bonnes pratiques de codage pour les développements associés à Pokemon Master.

Le projet Pokemon Master nécessite des règles pour :

- Code Langage c

2. La solution

2.1 Convention de codage

2.1.1 Structuration, Syntaxe & Apparence

2.1.1.1 Structuration d'un fichier source

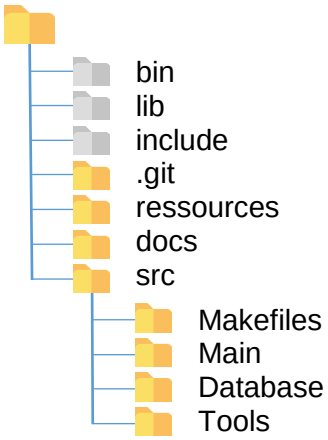
Un fichier .h, doit respecter les règles suivantes:

- Commencer par un entête,

```
#ifndef HEADER_NAME
#define HEADER_NAME

*! \file nomDuFichier.h
   \brief Description simple.
   \version 0.1

   Description complete.
*/

…
#endif /* HEADER_NAME */
```

- Après l'entête, les includes sont déclarés,
- La longueur totale des fichiers sources (sans commentaires et lignes vides) ne devraient pas excéder 2000 lignes,

2.1.1.2 Structuration du projet



PROJET/
Répertoire de base, contenant l'ensemble des répertoires du projet ainsi que le fichier de configuration de doxygene.

PROJET/bin
PROJET/lib
PROJET/include: réperoire autogénérer suite à la compilation.

PROJET/.git : contient la base local git des sources.
PROJET/ressources/bases: contient les fichiers de la base de donnée.
PROJET/docs: contient les documents du projet.
PROJET/src: contient l'ensemble des fichiers est repertoire des sources du projet.

PROJET/src/Makefiles: contient les makefiles utile pour la compilation du projet.
PROJET/src/Main: contient les sources du programme principale et génére l'executable.
PROJET/src/Database: contient les sources utiles pour la gestion de la base de donnée et génére une librairie.
PROJET/src/Tools: contient les sources d'utilitaire pour le bon fonctionnement du programme et génére une librairie.

2.1.1.3 Compilation & génération

Ce sont des modules en code natif, compilés avec gcc via un ensemble de makefile.
La compilation est automatique. Pour se faire il faut lancer le scrit Compilation.sh dans le repertoire PROJET/src avec la commande: ./Compilation.sh ou sh Compilation.sh.

2.1.1.4 Rédaction de code en anglais

Le code doit systématiquement être rédigé en anglais.
Cette consigne s'applique à tout élément de code, que ce soit aux identifiants, méthodes, ou autre.

Seul les commentaires sont en français.

2.1.1.5 Nommage des identifiants (structure & variables)

La structuration d'un identifiant doit respecter la convention type:

- Succession de mots commençant par des majuscules ou acronymes en majuscules.

Les règles suivantes doivent être appliquées:

- Un identifiant fait au moins 3 caractères. Les identifiants monolettre type "i, j, k ..." sont tolérés uniquement pour les compteurs de boucle mais sont déconseillés.
- Un identifiant doit être explicites, le nom doit dénoter le contenu et la fonction de l'objet nommé,
- Un identifiant doit être succincts, pas de noms long ou de phrase (1-2, 3 mots courts max),
- Les acronymes apparaissant dans les identifiants doivent être entièrement en majuscules,
- L'utilisation des "_" (underscore) est à proscrire,
- Le typage de la variable ne doit pas apparaître dans l'identifiant ("FloatResultat"...).
- L'utilisation de termes triviaux tel que "myStruct", "aStruct", "totoStruct" est interdite,
- La réutilisation de terme générique n'est pas acceptable.

2.1.1.6 Nommage des méthodes

Le nom des méthodes doit:

- commencer par un verbe
- être saisi avec des caractères majuscules et minuscules, la première lettre du verbe étant en minuscule et la première lettre de tous les autres mots composant le nom de la méthode devant être en majuscule.

2.1.1.7 Nommage des attributs

Le nom des attributs de structure, des variables et des paramètres des méthodes doit:

- être saisi avec des caractères majuscules et minuscules, la première lettre de chaque mot composant le nom de la classe devant être en majuscule;
- être le plus simple possible et décrire au mieux l'élément en question;
- ne pas contenir d'acronyme ou d'abréviation (excepté pour les abréviations et acronymes courants comme par exemple API, URL …);
- ne pas contenir de dollar '$' et de soulignement '_'.

Les noms des collections d'objets doivent être au pluriel.

2.1.1.8 Nommage des constantes

Le nom des constantes doit:

- être entièrement en majuscule,
- le séparateur de mot est le caractère de soulignement '_'.

2.1.1.9 Alignement des blocs (accolades)

Les accolades dissymétriques sont la norme retenue:

- o l'accolade ouvrante '{' apparaît à la fin de la ligne de l'entité associée,
- o l'accolade fermante '}' apparaît seule sur une ligne, à la verticale de l'entité associée.

typedef struct {

```
   int ivar1;
   int ivar2;
}Test;
```

Dans les deux cas, les deux accolades peuvent figurer sur la ligne de l'entité, si le bloc est vide.

```
void emptyMethod() {}
```

2.1.1.10 Indentation

Le caractère \t (touche tab) est utilisé (notamment pour réduire la taille des fichiers js).
La configuration d'eclipse permet d'afficher les \t sous forme de 4 caractères. C'est la configuration préconisée.

2.1.1.11 Césures

Lorsqu'une expression ne tient pas sur une seule ligne, il est nécessaire de la couper selon les principes généraux suivants:

- la couper après une virgule,
- la couper avant un opérateur,
- aligner la nouvelle ligne avec le début de l'expression de la ligne précédente,
- si l'application des règles précédentes conduit à un code confus, indenter simplement la ligne avec 2 "tab".

2.1.1.12 Longueur de ligne

La longueur d'une ligne doit se limiter à ce qui peut être affiché sur un écran 24" en police 12. Un chiffre type sera 200 caractères.

2.1.1.13 Commentaires

Les commentaires devront être rédigés en français comme précisé précédemment.
Privilégier une rédaction synthétique.

L'utilisation de doxygene implique une syntaxe spéciale des commentaires. Les commentaire doxygene ne seront que dans les fichiers header du programme.

2.1.1.13.1 Commentaire d'entête (structure, méthode, variables, ...)

2.1.1.13.1.1 Fichier

```
/*! \file nomDuFichier.h
    \brief Une description courte.
    \version 0.1

    Description complête.
*/
```

Ce type de commentaire doit être placé tout en haut du fichier header.

2.1.1.13.1.2 Enum

```
/*! \enum NON_ENUM
    \brief Decription
*/
typedef enum {
NON_ENUM_1 = -1, /**< Description de la valeur */
NON_ENUM_2 /**< Description de la valeur */
} NON_ENUM;
```

2.1.1.13.1.3 Structure

```
/*! \struct NomDeLaStructure
    \brief Description
*/
typedef struct {
  int champ;   /**< Description du champ. */
} NomDeLaStructure;
```

2.1.1.13.1.4 Definition

```
/*! \def NOM_DEF
    \brief Description
*/
```

2.1.1.13.1.1 Fonctions

```
/*! \fn Prototype de la fonction
    \brief Description de la fonction

    \param nom de l'argument Description de l'argument
    \return Description du retour de la fonction
*/
```

2.1.1.13.2 Commentaire interne au code

Toute section de code qui n'est pas évidente à sa simple lecture peut nécessiter un commentaire explicatif. Ce commentaire peut être monoligne (//) ou multiligne (/* */).

2.1.1.13.3 Neutralisation de code par commentaires

Utiliser la notation "//"

```
// float x;
```

Cette pratique est à proscrire sur le code livré, il s'agit généralement de code "mort" ou désactivé.

2.1.1.14 Déclarations

2.1.1.14.1 Variables

Une déclaration par ligne est à privilégier.
Ainsi,

```
int level = 0; // indentation level
int size = 0; // size of table
```

sera préféré à

```
int level = 0, size = 0;
```

Le commentaire n'est nécessaire que si la variable n'a pas un nom qui permet de comprendre immédiatement sa fonction. Il est donc en général inutile si les autres règles sont respectées.

2.1.1.14.2 Méthodes

Le nom d'une méthode et la parenthèse ouvrante avant la liste des paramètres doivent être accolés (pas d'espace).
Un espace doit apparaître après chaque virgule dans des listes d'arguments.
Les méthodes sont séparées par des lignes vides.
Exemple:

```
void traiterUnChamp( ChampDescr champ ) {
   ...
}

void traiterUnChamp( ChampDescr champ, Provenance source ) {
   ...
}
```

2.1.1.14.3 Ordre

Les déclarations doivent figurer au début des blocs.

```
void myMethod() {
   int int1 = 0; // bien
   if (condition) {
      int int2 = 0; // bien
      ...
   }

   int foo = 2 ; // pas bien
   ...
}
```

La seule exception à cette règle est la déclaration dans la section d'initialisation de la boucle « for »:

```
for (int i = 0; i < maxLoops; i++)
```

Les déclarations locales qui cachent les déclarations de plus haut niveau sont proscrites.

```
int count;
...
void myMethod()
{
  if (condition) {
    int count; // NON !
    ...
  }
  ...
}
```

2.1.2 Instructions

2.1.2.1.1 Instruction simple

Chaque ligne doit contenir au plus une instruction.
Exemple:

```
argc++; // Correct
argc++; r = a * 2; // INTERDIT!
```

De même, on n'utilisera pas plusieurs pré / post incrémentations dans une même ligne de code:

```
int i = 3 ;
int res = i++ + ++i * i-- - --i ; // relativement peu lisible.
```

2.1.2.2 Structures de contrôles

2.1.2.2.1 Structure « for »

Une instruction « *for* » doit suivre le format suivant:

```
for (initialization; condition; update) {
  …
}
```

Une instruction « *for* » vide (dans laquelle tout figure dans les clauses d'initialisation, de condition et mise à jour) doit suivre un des formats suivants:

```
for (initialization; condition; update) {
  // Boucle volontairement vide
}
```

```
for (initialization; condition; update){ // no data}
```

On notera la présence d'accolades en fin d'instruction pour signifier explicitement qu'il s'agit d'une boucle sans traitement (par exemple, recherche d'indice) et non pas d'une faute de frappe.
Un espace suivra chaque ';' ainsi que chaque ','.

2.1.2.2.2 Structure « while »

Une instruction « while » doit suivre le formatage suivant:

```
while (condition) {
   …
}
```

Une instruction « *while* » vide doit suivre les formalismes suivants:

```
while (condition) {
   // Boucle volontairement vide
}

while (condition) {}
```

2.1.2.2.3 Structure « do – while »

Une instruction « *do while* » doit suivre le formalisme suivant:

```
do {
   …
} while (condition);
```

2.1.2.2.4 Structure « if »

Les instructions conditionnelles doivent suivre le formatage suivant:

```
if (condition) {
   …
}

if (condition) {
   …
} else {
   …
}

if (condition) {
   …
} else {
   if (condition) {
      …
```

```
  } else {
    …
  }
}

if (condition) {
  …
} else if (condition) {
  …
} else if (condition) {
  …
} else {
  …
}
```

Les instructions « *if* » doivent toujours être utilisées avec des accolades.

```
if (condition)
  … // NON!
```

Il est recommendé de limiter le nombre de if ... else if à un maximum de 8 blocs. Au delà, il est préconisé d'utiliser un switch…case.

2.1.2.2.5 Opérateur ternaire

L'expression conditionnelle doit être mise entre parenthèses.
Exemple:

```
(x >= 0) ? x : -x;
```

Attention cet opérateur est à réserver pour des usages spécifique de type arithmétique conditionnelle. Il ne faut pas l'utiliser pour gérer des conditions et des blocs comme un if "classique".

2.1.2.2.6 Structure « switch »

Une instruction « *switch* » doit suivre le formalisme suivant:

```
switch (condition) {
  case 0:
  {
    …
  }
  break;

  case 1:
  {
    …
  }
  break;

  case 2:
```

```
  {
    …
  }
  break;

  default:
  {
    …
  }
  break;
}
```

Chaque cas doit être pourvu d'une instruction break.
Chaque instruction « *switch* » doit inclure un cas par défaut.

Il est autorisé d'utiliser des « *case* » sans block, généralement pour traiter une succession de cas avec un seul bloc de code.
Exemple:

```
switch (condition) {
  case 1:
  case 2:
  case 3:
  {
    …
  }
  break;

  default:
  {
    …
  }
  break;
}
```

2.1.2.3 Ligne vide et espace

2.1.2.3.1 Ligne vide

Une ligne vide doit toujours être utilisée dans les circonstances suivantes:

- entre les méthodes,
- dans une méthode, entre la définition des variables locales et la première instruction,
- avant un bloc isolé ou une simple ligne de commentaire,
- entre des sections logiques à l'intérieur d'une méthode pour regrouper les blocs successifs de code et améliorer la compréhension.

2.1.2.3.2 Espace

- Un mot clé suivi par une parenthèse doit être séparé de la parenthèse par un espace.

Exemple:

```
while (true) {
   ...
}
```

- Un espace ne doit pas être utilisé entre le nom d'une méthode et sa parenthèse ouvrante.
- Un espace doit apparaître après les virgules dans des listes d'arguments et les « for ».
- Tous les opérateurs binaires, sauf ".", (ex: =, >, <, +, &, …) doivent être séparés de leurs opérandes par des espaces; des espaces ne doivent jamais séparer des opérateurs unaires comme le moins unaire, l'incrément ("++") et le décrément ("--") de leurs opérandes.

Exemple:

```
a += c + d;
a = (a + b) / (c * d);
while (d++ = s++) {
   n++;
}
printf("size is %d", foo);
```

- Les ';' dans une instruction « for » doivent être suivis par des espaces.

Exemple:

```
for (expr1; expr2; expr3)
```

- Les conversions de types sont suivies d'un espace.

Exemple:

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

- Les parenthèses ouvrantes peuvent être suivies d'un espace; de même les parenthèses fermantes peuvent être précédées d'un espace.

Exemple:

```
myMethod(aNum, x);
myMethod( aNum, x );
```

2.2 Règles de programmation

Les règles générales suivantes sont applicables:

- Déclaration des variables au plus près de leur utilisation (en respectant la logique de blocs et la déclaration en début de bloc).
- Utilisation du bloc "static { }" à restreindre à des initialisations de singletons ou au

chargement de bibliothèques natives

- Centralisation des constantes communes. Attention, parfois il est contreproductif et dangereux en terme de dépendance de trop centraliser les constantes (créations de dépendances enchevêtrées complexes).
- Pas d'assignations multiples sur une seule ligne: a = b = 1;
- Pas de modification des compteurs de boucle for.
- Code mort: le code mort doit être systématiquement supprimé. Tout code en commentaire doit être précédé d'un commentaire "monoligne" (// this code is disabled because ...) indiquant pourquoi le code est conservé mais commenté.

2.2.1 Logger

Un logiciel doit générer des logs, permettant de vérifier son bon fonctionnement. Les logs sont répartis en différents niveaux. Chaque niveau a une destination précise:

- Les logs de niveau DEBUG sont ciblés développement / intégration logicielle,
- Le niveau INFO est destiné à un administrateur de niveau "utilisateur" final,
- Le niveau MENU est destiné à un administrateur de niveau "utilisateur" final,
- Le niveau WARN correspond à une erreur non bloquante prévue par le développement ou les spécifications,
- Le niveau ERROR correspond à une erreur grave, mais non bloquante, dont la conséquence peut être l'arrêt du logiciel,

Les règles suivantes sont à respecter obligatoirement:

- Les traces via "printf(…)" sont interdites sur le code livré.
- Les logs ne doivent pas impacter le "fonctionnement" du programme.