CIRAD
Département BIOS, Unité DAP, Équipe ID

Valentin GUIGNON, 02/03/2009
Version 1.2.0

CONVENTIONS DE PROGRAMMATION EN PERL

South Green
bioinformatics platform

Sommaire

1. Bases

Quelques notions importantes:

- les programmes et les packages doivent être développés à partir des squelettes de base de l'équipe ID (incluant « `use strict;` », « `use Carp qw(confess cluck);` » et « `use warnings;` »). Ces squelettes sont nommés « `Template.pl` » pour les programmes, « `Template.cgi` » pour les scripts web CGI et « `Template.pm` » pour les modules;
- préférer l'utilisation de « `confess` » et « `cluck` » plutôt que respectivement « `die` » et « `warn` » (plus d'information pour le débugage);
- il ne faut pas utiliser « `die` » ou « `confess` » dans les packages sans que cela ne soit explicitement documenté (cf. section « documentation pod »).
- la rédaction du code se fait intégralement en anglais international dans un souci d'ouverture du code à la communauté internationale.

2. Nommage

A l'heure où les éditeurs de code source intègrent des fonctions de complémentation automatique des mots, les noms utilisés doivent être explicites (longs) et les acronymes et « raccourcis » doivent être évités dans la mesure du possible, ceci dans le but de lever un maximum d'ambiguïté sur les rôles des éléments nommés.

Les noms de variables à une ou deux lettres ne sont tolérés que pour les itérateurs.

Exemple:

Correct:

```
my $circle_coordinate_x;
my $circle_label_coordinate_x;
my $client_coordinate_x;

for my $x ($MIN_X .. $MAX_X)
# tolere mais on preferera:
for my $cell_x ($MIN_CELL_X .. $MAX_CELL_X)
# qui est plus parlant et evite les confusions futures
```

Incorrect:

```
my $cx;
my $cl_coordinate_x; # ce n'est pas parlant et peut correspondre
# a chacun des noms presentes dans le bon exemple.
```

2.1. Fichiers

Un fichier contenant un ou plusieurs packages porte le même nom (base name) que le package principal suivit de l'extension « .pm ».

Exemple:

SomeModule`.pm`

Les fichiers contenant des programmes doivent porter l'extension « `.pl` » et comporter de préférence des minuscules.

Exemple:

my_program`.pl`

Les fichiers contenant des programmes web doivent porter l'extension « `.cgi` » et comporter seulement des caractères alpha-numériques minuscules normaux (ie. sans accents, cédille ou autres) ou des « underscores » (« _ ») pour que leurs noms ne comportent pas de caractères encodés en URL.
Exemple:

hello_world`.cgi`

2.2. Format

Les noms de packages sont composés de mots dont seul la première lettre de chaque mot est en majuscule.
Exemple:

```
package PackageName;
```

Les noms des fonctions sont construits de la même façon que les variables à la différence près que le premier mot est entièrement en minuscules et est de préférence un verbe d'action.
Exemple:

```
sub doTheFunction
```

Les noms de constantes sont intégralement en majuscules avec des « underscores » (« _ ») pour séparer les mots.
Exemple:

```
our $CONSTANT_NAME = 0;
```

Les noms de variables sont intégralement en minuscules avec des « underscores » (« _ ») pour séparer les mots.
Exemple:

```
my $variable_name;
```

Les noms de références sur des variables passées en argument à une procédure ou une fonction doivent « idéalement » être suffixés par « _ref » voire le type de la variable avant le le « _ref » mais ce n'est pas obligatoire, juste fortement recommandé.
Exemple:

```
sub doSomeStuff
{
    my ($param_hash_ref, $some_table_ref) = @_;
    ...
```

Les noms commençant par « underscore » (« _ ») sont utilisés pour indiquer que leur usage est réservé au développement interne et qu'ils ne doivent pas être utilisés par des programmeurs tiers.
Exemple:

```
sub _doMyProtectedFunction
{
    …
    my $_protected_member_variable
```

Les noms de label (étiquettes) sont écrits de la même façon que les constantes et en début de ligne.
Exemple:

```
LABEL_1:
```

Les noms des délimiteurs de blocs de texte doivent être préfixés et suffixés de 3 « underscores » (« _ ») et sont composés de plusieurs parties séparées par un caractère « underscore ». La première partie du nom doit être le numéro de ligne où commence le bloc de texte. Ce numéro n'a pas à être maintenu à jour par la suite si la position du bloc change car il sert surtout à rendre unique le nom du bloc. La deuxième partie est le nombre de lignes que doit contenir le bloc. Ce nombre n'est pas forcément exact, est présent à titre indicatif et peut être omis. La troisième partie est le nom descriptif du bloc en caractères alpha-numériques intégralement en majuscules dont les différents mots sont séparés par des « underscore ».
Exemple:

```
…
# line 1233
print <<"___1234_3_WELCOME_MESSAGE___";
Welcome
and enjoy your stay!
                                The Staff
___1234_3_WELCOME_MESSAGE___
# line 1239
```

3. Syntaxe

3.1. Code

Une ligne de code ne doit contenir au maximum qu'une seule expression (un seul bloc suivit d'un « ; »).
Exemple:
Correct:

```
my $toto = 0;
my $tata = 1;
```

Incorrect:

```
my $toto = 0; my $tata = 1;
```

L'indentation (décalage horizontal à chaque ligne pour séparer visuellement les blocs entre accolades) se fait par 4 sauts de colonnes (4 espaces); Préférer les espaces aux tabulations.
Exemple:

```
sub Test
{
    my ($self, $test_level) = @_;
    my $result      = 0;
    my $attempts    = 10;
    while ($attempts--)
    {
        # first part of the tests
        if ($self->doTest1())
        {
            ++$result;
            $self->log("Test 1 failed!\n");
        }

        # second part of the tests that does a different thing
```

```
        if ($self->doTest2($test_level))
        {
            if (0 == $test_level)
            {
                ++$result;
                $self->log("Test 2 level 0 failed\n");
            }
            elsif (1 == $test_level)
            {
                ++$result;
                $self->log("Test 2 level 1 failed\n");
            }
            else
            {
                $self->log("Test 2 level unknown: $test_level\n");
            }
        }
    }
    return $result;
}
```

Deux règles sont possibles pour le positionnement des accolades ouvrantes et fermantes:

- style « BSD »: elles se trouvent sur des lignes séparées du reste du code et en vis-à-vis vertical de l'instruction précédente à laquelle elles sont rattachées (ie. « `sub` », « `if` », etc.);
- style « K&R »: l'accolade ouvrante se trouve précédée d'un seul espace à la fin de la ligne de l'instruction à laquelle elle se rattache et l'accolade fermante en vis-à-vis vertical du premier caractère de cette même instruction.

Dans tous les cas, une seule de ces 2 possibilités doit être choisie et utilisée pour l'ensemble d'un projet. Si l'auteur n'a pas d'habitude particulière, il est encouragé à utilisé le style « BSD ».

Exemple:

Style « BSD »:

```
sub doTest
{
    my ($toto, $tutu) = @_;
    my $result;
    if (1 == $toto)
    {
        $result = 23;
        if (0 != $tutu)
        {
            $result *= -1;
        }
    }
    else
    {
        $result = 806;
    }
    return $result;
}
```

Style « K&R »:

```
sub doTest {
    my ($toto, $tutu) = @_;
```

```
    my $result;
    if (1 == $toto) {
        $result = 23;
        if (0 != $tutu) {
            $result *= -1;
        }
    }
    else {
        $result = 806;
    }
    return $result;
}
```

Incorrect:

```
sub doTest # BSD, K&R and other styles are mixed up
{ # BSD style
    my ($toto, $tutu) = @_;
    my $result;
    if (1 == $toto) { # K&R style
        $result = 23;
        if (0 != $tutu)
          { # other style
            $result *= -1;
          } # other style
    } else { # yet an other style
        $result = 806;
    } # K&R style
    return $result;
} # BSD style
```

Il faut toujours finir une ligne d'instruction par un point virgule même si elle est seule dans son bloc.

Exemple:

Correct:

```
{
    $toto = 1;
}
```

Incorrect:

```
{
    $toto = 1
}
```

Il faut un espace après chaque opérateur.

Exemple:

Correct:

```
if ($self->doTest1())
```

Incorrect:

```
if($self->doTest1())
```

Il est recommandé de séparer d'une ligne vide les séries d'instructions effectuant des tâches différentes;

Exemple:

Correct:

```
while (--$attempts)
{
    # first part of the tests
    if ($self->doTest1())
    {
        ++$result;
        $self->log("Test 1 failed!\n");
    }

    # second part of the tests that does a different thing
    if ($self->doTest2($test_level))
    {
        if (0 == $test_level)
        {
            ++$result;
            $self->log("Test 2 level 0 failed\n");
        }
        elsif (1 == $test_level)
        {
            ++$result;
            $self->log("Test 2 level 1 failed\n");
        }
        else
        {
            $self->log("Test 2 level unknown: $test_level\n");
        }
    }
}
```

Non recommandé:

```
while (--$attempts)
{
    # first part of the tests
    if ($self->doTest1())
    {
        ++$result;
        $self->log("Test 1 failed!\n");
    }
    # second part of the tests that does a different thing
    if ($self->doTest2($test_level))
    {
        if (0 == $test_level)
        {
            ++$result;
            $self->log("Test 2 level 0 failed\n");
        }
        elsif (1 == $test_level)
        {
            ++$result;
            $self->log("Test 2 level 1 failed\n");
        }
        else
```

```
        {
            $self->log("Test 2 level unknown: $test_level\n");
        }
    }
}
```

Il faut au moins deux lignes vides pour séparer les blocs de procédures.

Exemple:

Correct:

```
sub doTest1
{
    ... # some code
}


sub doTest2
{
    ... # some other code
}
```

Incorrect:

```
sub doTest1
{
    ... # some code
}

sub doTest2
{
    ... # some other code
}
# note: on risque de mélanger une séparation "intra-bloc" avec une
séparation de procédures!
```

Il faut utiliser au moins quatre lignes vides pour séparer des packages dans un même fichier source.

Il ne faut pas d'espace entre une fonction et ses parenthèses.

Exemple:

Correct:

```
if ($self->doTest1())
```

Incorrect:

```
if ($self->doTest1 ())

if($self->doTest1())
# attention! le "if" est un opérateur! Il échappe à la règle!
```

Il ne faut pas d'espaces avant les points virgules.

Il ne faut pas d'espace avant les virgules mais en revanche, un espace après est nécessaire.

Exemple:

Correct:

```
my ($self, $test_level) = @_;
```

Incorrect:

```
my ($self,$test_level) = @_;
# l'espace manque

my ($self ,$test_level) = @_;
#l'espace est mal placé

my ($self , $test_level) = @_;
# un espace en trop avant la virgule
```

Les lignes de tests longues peuvent être coupées avant les opérateurs logiques séparant les blocs (`or`, `and`, etc.), les lignes supplémentaires ainsi créées sont alignées verticalement sur le bloc sur lequel porte l'opérateur.

Exemple:

```
if (($self->doMyTest1WhichHasALongName())
    || ($self->doTest2()
        and $self->doTest3()
        and $self->doTest4()
       )
   )
```

Les lignes d'appel de procédures longues peuvent être coupées après les virgules séparant les arguments.

Exemple:

```
$self->doMyFunc($arguement_number_one,
                $argumengt_number_two,
                $last_argument);
```

Les lignes longues en générale peuvent être coupées avant les opérateurs (`+`, `-`, `*`, `.`, `or`, `&&`, etc.).

Exemple:

```
$value = 123456789
         + 10111213
         - 987654321;

print "tototututatatiti" . "popopupupapapipi"
      . "lololululalalili";

$toto = $my_variable_with_a_very_long_name
        or die "Blablabla!";
```

Les éléments similaires de lignes consécutives doivent être de préférence alignés verticalement.

Exemple:

```
my $result      = 0;
my $attempts    = 10;

open(FILE, $fh)     or die $!;
open(FILE, $fh2)    or die $!;

$toto =~ tr[a-mn-z]
```

```
        [n-za-m];
```

Les éléments d'un même bloc de parenthèses ou de crochets répartis sur plusieurs lignes doivent être alignés verticalement de façon logique.

Il faut éviter les redondances de ponctuation si la clarté du code n'en souffre pas.
Exemple:

```
$self->{TOTO} = 0;
# est mieux que:
$self->{"TOTO"} = 0;
```

Il faut favoriser l'utilisation des parenthèses au maximum pour une meilleur clarté du code et lever d'éventuels doutes pour d'autres programmeurs.

Il faut des espaces avant et après les points de concaténation et les opérateurs de comparaison ou d'assignation.
Exemple:

```
print "toto" . "tutu" . "tata";
```

Il faut indenter de deux espaces en moins l'instruction « last » dans les boucles et l'instruction « return » au milieu d'une procédure (optionnel en fin de procédure).
Exemple:

```
LINE:
    for (;;)
    {
        ... # some code
        --$toto;
      last LINE if (0 == $toto);
      next LINE if /^#/;
        $tutu = $self->doSomething();
        ... # some other code
    }
```

3.2. Commentaires

Les commentaires de code doivent être placés pour décrire le but d'un groupe d'instructions et peuvent être omis lorsque le code est suffisamment explicite pour être compris par quelqu'un d'extérieur au programme. La rédaction se fait en anglais pour faciliter l'utilisation du code par des programmeurs internationaux.

3.3. Documentation POD

Les commentaires POD doivent être localisés juste avant le programme, le package ou la procédure décrite pour faciliter la synchronisation entre commentaire et sources. Comme pour les commentaires, la rédaction se fait en anglais pour faciliter l'utilisation de la documentation par des programmeurs de toutes nationalités. Dans le code suivant, les mots *en italique* doivent être remplacés par le texte approprié lors de la programmation. Chaque champ pod (commençant par « = » sans espace en début de ligne doit être précédé et suivi d'au moins une ligne vide. Les paragraphes qui doivent être représentés sous forme de lignes de code doivent être précédés de 4 espaces (pod peut alors décider d'utiliser une police à largeur fixe pour ceux-ci).

3.3.1. En tête de package:

```
=head1 NAME

PackageName - Short description in a few words

=head1 SYNOPSIS

    my $instance = Group::PackageName->new();
    $instance->subName();
    (...code showing various uses of the package...)

=head1 DESCRIPTION

Package (long) description.

=cut

package Group::PackageName;
```

3.3.2. Bloc des « `use` »:

```
use strict;
use Carp qw(confess cluck);
use warnings;
```

Dans le cas d'utilisation de variables globales extérieures, on peut ajouter la ligne suivante au bloc des « `use` »:

```
use vars qw($VARNAME1 $VARNAME2 ...);
```

Dans le cas où le package est dérivé d'un autre package, l'héritage se fait en ajoutant au bloc des « use »:

```
use base qw(OtherGroup::BaseClass);
```

3.3.3. Définition des constantes:

```
# Package constants
####################

=head1 CONSTANTS

B<$CONSTANT_NAME1>: (constant nature) (constant access)

constant description and use.

B<CONSTANT_NAME2()>: (constant nature) (constant access)

constant description and use.

=cut

our $CONSTANT_NAME1          = "value1";
```

```
sub CONSTANT_NAME2()          { "value2" }
```

Le champ « *constant_type* » peut prendre des valeurs telles que « integer », « character », « float », « string », « hash », « array », « object », et s'il s'agit d'une référence le type est suivi de « reference ». Lorsqu'il s'agit d'un type complexe, il peut être judicieux de détailler le contenu de certaines parties (ie. Pour un « array », détailler les types des éléments contenus ou mettre « array of » suivit du type). Dans le cas de constantes sous forme de fonctions (plus facilement exportables), le type de retour sera préfixé de « sub ».

3.3.4. Définition des variables

```
# Package variables
####################

=head1 VARIABLES

B<$variable_name>: (variable nature) (variable access)

variable description and use.
Default: default_value

=cut

my $variable_name = "value";
```

Les différents champs suivent les mêmes règles que pour la définition des constantes.

3.3.5. Définition des méthodes

La définition des méthodes représente le corps du package et est décomposé en plusieurs parties. La première comprend les méthodes dites « statiques » qui peuvent être appelées sans instances (constructeur,...); la seconde regroupe les accesseurs (« setters » et « getters ») des différents champs variables d'une instance; la troisième partie regroupe les méthodes d'instances qui peuvent éventuellement supporter les appels statiques. Enfin, la dernière partie concerne « l'autoload ». Chaque partie commence par un champ « =head1 *** » unique dans le package qui est suivi de l'ensemble des méthodes qu'il regroupe. Chaque méthode est précédée de sa documentation.

Le début de la définition des méthodes est signalé de la même façon que les constantes et les variables par le commentaire suivant:

```
# Package subs
###############
```

3.3.5.1. Méthodes statiques

```
=head1 STATIC METHODS

=cut
```

Les deux premières méthodes définies doivent être le constructeur et le destructeur.

```
=head2 CONSTRUCTOR
```

```
(constructor documentation, see 3.3.5.5)

=cut
```

et

```
=head2 DESTRUCTORS

(destructor documentation, see 3.3.5.5)

=cut
```

3.3.5.2. Accesseurs

```
=head1 ACCESSORS

=cut
```

3.3.5.3. Méthodes d'instance

```
=head1 METHODS

=cut
```

3.3.5.4. Autoload

```
=head1 AUTOLOADS

=cut
```

3.3.5.5. Structure générale de documentation d'une méthode

Chaque méthode est documentée de la façon suivante:

```
=head2 function_name

B<Description>: function description

B<Argscount>: number of possible arguments

=over 4

=item variable_name : (variable nature) (requirements)

variable description.

=item variable_name2 : (variable nature) (requirements)

variable description.
```

```
=back

B<ReturnType>: (returned type)

description of what is returned.

B<Caller>: what usually calls this function (may be omitted).

B<Exception>:

=over 4

=item * exception_type:

description, case when it occurs and a solution to the exception if it's
relevant.

=back

B<Example>:

    example of use (lines of code).

=cut
```

Le champ « `Argscount` » peut valoir un chiffre ou un ensemble de valeurs possibles (« `2` », « `3 to 5` », « `1, 2 or 4` », etc.). Chaque champ « `=item` » suivant « `Argscount` » sert à décrire un argument et ses contraintes (« requirments ») comme « `R` » pour « `Required` » (obligatoire), « `O` » pour « `Optional` » (optionnel et peut ne pas être présent dans la liste des arguments) ou « `U` » pour un paramètre optionnel qui doit être « undef » s'il n'est pas spécifié (« `Undef` »), à quoi peut s'ajouter « `> 0` » pour nombre qui doit être strictement positif, « `lowercase` » pour un string qui doit être en minuscules, ou d'autres règles explicites au besoin.

Si le type spécifié par « `ReturnType` » est un type complexe, il faut dans la mesure du possible détailler les éléments retournés (ie. dans un tableau, chaque élément du tableau).

Le champ « `Caller` » décrit l'endroit d'où est appelée la fonction en règle générale: « `general` » s'il n'y a pas de cas particulier, « `instance` » si la méthode est généralement appelée depuis une autre méthode de la même instance, « `package` » si la méthode est appelée depuis la même ou une autre instance du même package, « `perl system` » s'il s'agit d'une méthode appelée automatiquement par perl, « `NomPackage` » si cette méthode a été prévue pour être utilisée avec un package particulier nommé « NomPackage », etc.

3.3.5.6. Information sur les auteurs et la version

```
=head1 AUTHORS

 author1 name (company): email
 author2 name (company): email
 ...

=head1 VERSION
```

```
 version: version.subversion.build

 date: dd/mm/yyyy

=cut
```

3.3.5.7. Fin du package

La dernière ligne du package est:

```
return 1; # package return
```

4. Astuces et recommandations

Voici une petite liste d'astuces et de recommandation pour faciliter la maintenance du code, son optimisation ou sa sécurité:

- il faut toujours préférer « `@_` » à « `shift` » (*optimisation* et *lisibilité*).

Exemple:

Correct:

```
my ($self, $arg1, $arg2, $arg3) = @_;
```

Incorrect:

```
my $self = shift;
my $arg1 = shift;
# ou pire!
my ($arg2, $arg3) = (shift, shift);
```

- il faut de préférence écrire les tests de comparaison en mettant les constantes en membre de gauche pour repérer facilement lors de l'exécution les erreurs de syntaxe (*maintenance*).

Exemple:

```
# il vaut mieux ecrire:
if (0 == $toto)
# plutot que :
if ($toto == 0)
# ce qui empecherait l'execution potentielle d'une erreur de type (1 seul "="):
if ($toto = 0)
```

- il vaut mieux utiliser l'incrémenteur ou le décrémenteur préfixe lorsqu'une variable doit être incrémentée ou décrémentée sans avoir à être évaluée avant cette opération (*optimisation*). Cela évite au système de devoir créer une variable intermédiaire. Par exemple « `for (my $i=0; 10 > $i; ++$i)` ». En revanche, le comportement de l'incrémenteur postfix est important dans ce genre d'expressions: « `print $i++;` » et ne peut être remplacé tel quel;
- lors du passage d'arguments en nombre variable à une fonction, préférer le passage d'une seule référence sur un hash (éventuellement anonyme) en utilisant ses clés comme noms d'arguments (*maintenance*). Par exemple: « `doSomeStuff({'color red' => 255,`

`'color green' => 128, 'color blue' => 0, 'color mode' => 'RGB', 'alpha' => 160, 'text' => 'Toto'});` » où l'on pourrait omettre le paramètre « alpha » et utiliser une valeur par défaut si « `my ($hash_ref) = @_; if (not exists($hash_ref->{'alpha'})) {$hash_ref->{'alpha'} = '255';}` »;

- toujours tester si un fichier a bien été ouvert avant de travailler dessus par un « `if (open(FH, ">myfile.ext")) { ... }` » et utiliser « `use Fatal qw/:void open close/;` » pour éviter les problèmes liés aux éventuels oublis (*sécurité*);
- pour éviter de s'embêter à échapper chaque caractère spécial ou saut de ligne d'une expression, on peut utiliser soit les blocs de texte, soit l'opérateur de « quote » (*maintenance et lisibilité*). Par exemple:

```
        ...
        print <<"___51_4_HTML_CODE___";
<ul class="menu_list">
  <li class="leaf"><a href="/?q=content/databases"
title="Databases">Databases</a></li>
  <li class="leaf"><a href="/?q=content/tools"
title="Tools">Tools</a></li>
  <li class="collapsed"><a href="/?q=content/downloads"
title="Download">Downloads</a></li>
</ul>
___51_4_HTML_CODE___
```

ou

```
print q{j'"aime" les apostrophes "partout" et le signe $ aussi!};
```

ou encore, à la place d'un « `s/toto/tutu/` » sans commentaires:

```
s {toto}  # Remplace toto
  {tutu}  # par tutu
```

Note: plus de détails sur comment commenter les expressions régulières là: http://www.perl.com/pub/a/2004/01/16/regexps.html

- le mode « taint » (section perlsec de la doc Perl) permet de vérifier que l'on teste bien chaque paramètre passé à notre programme avant de l'utiliser (*sécurité*). Il suffit de changer la première ligne du script Perl en « `#!/usr/bin/perl -T` »;
- le « profiling » permet de voir où le code est lent et donc les parties à optimiser en priorité. Pour cela, il suffit d'ajouter l'argument « `-d:DProf` » à la ligne de commande Perl (*optimisation*) pour obtenir certaines informations mais plus de détails sont possible, il suffit de chercher sur Internet « Perl profiling »;
- la documentation Perl est toujours utile: http://perl.enstimac.fr/DocFr.html
  surtout les sections *perlfunc* et *perlop*.