perltoot

Table des matières

1 NAME/NOM

perltoot - Tutoriel orienté objet de Tom.

2 DESCRIPTION

La programmation orientée objet se vend bien de nos jours. Certains managers préféreraient même s'arrêter de respirer plutôt que de se passer d'objets. Pourquoi cela ? Qu'est-ce qu'un objet a de si particulier ? Et plus simplement, qu'est-ce qu'un objet ?

Un objet n'est rien de plus qu'une manière de cacher des comportements complexes derrière un petit ensemble clair et simple à utiliser. (C'est ce que les professeurs appellent l'abstraction.) De brillant programmeurs qui n'ont rien à faire si ce n'est de réfléchir pendant des semaines entières sur des problèmes vraiment difficiles créent, pour les résoudre, de chouettes objets que des gens normaux peuvent utiliser (C'est ce que les professeurs appellent la réutilisation de programmes). Les utilisateurs (en fait, des programmeurs) peuvent jouer comme ils veulent avec ces objets tout prêts, mais ils n'ont pas à les ouvrir et à mettre la pagaille à l'intérieur. C'est comme un équipement coûteux : le contrat spécifie que la garantie disparaît si vous ouvrez le capot. Donc ne le faîtes pas.

Le coeur des objets est la classe, un espace de nommage protégé et un peu privé contenant données et fonctions. Une classe est un ensemble de routines relatives à un même problème. Vous pouvez aussi la considérer comme un type défini par l'utilisateur. Le mécanisme de paquetage de Perl, qui est aussi utilisé pour des modules traditionnels, sert pour les modules de classe. Les objets « vivent » dans une classe ce qui signifie qu'ils appartiennent à un paquetage.

La plupart du temps, la classe fournit à l'utilisateur de petits trucs. Ces trucs sont des objets. Ils savent à quelle classe ils appartiennent et comment se comporter. Les utilisateurs demandent quelque chose à la classe comme « donne-moi un objet ». Ou ils peuvent demander à l'un de ces objets de faire quelque chose. Demander à une classe de faire quelque chose pour vous consiste à appeler une *méthode de la classe*. Demander à un objet de faire quelque chose pour vous consiste à appeler une *méthode d'objet*. Demander soit à une classe (le cas usuel), soit à un objet (parfois) de vous retourner un objet consiste à appeler un *constructeur* qui est simplement une sorte de méthode.

Bon d'accord, mais en quoi un objet est-il différent de n'importe quel autre type de donnée en Perl ? Qu'est ce qu'un objet *réellement* ? Autrement dit, quel est son type de base ? La réponse à la première question est simple. Un objet n'a qu'une seule différence avec n'importe quel autre type de donnée en Perl : vous pouvez le déréférencer non seulement par une chaîne ou un nombre comme les tables de hachage ou les tableaux, mais aussi par des appels à des routines nommées. En un mot, les *méthodes*.

La réponse à la seconde question est que c'est une référence, mais pas n'importe quelle référence. Une référence qui a été *bénie* (blessed en anglais) par une classe particulière (lire: paquetage). Quel type de référence ? Bon, la réponse a cette question est un peu moins concrète, parce qu'en Perl, le concepteur de la classe peut utiliser n'importe quel type de référence. Ce peut être un scalaire, un tableau ou une table de hachage. Cela peut même être une référence à du code. Mais de par sa flexibilité inhérente, un objet est dans la plupart des cas une référence à une table de hachage.

3 Créer une classe

Avant de créer une classe, vous devez choisir son nom. Car le nom de la classe (du paquetage) détermine le nom du fichier qui la contiendra exactement comme pour les modules. Ensuite, cette classe devrait fournir un ou plusieurs moyens de créer des objets. Finalement, elle devrait proposer des mécanismes pour permettre à l'utilisateur de ces objets de les manipuler indirectement à distance.

Par exemple, créons un simple module pour la classe Person. Elle doit être stockée dans le fichier Person.pm. Si elle s'appelait Happy::Person, elle devrait être stockée dans le fichier Happy/Person.pm et son paquetage deviendrait Happy::Person au lieu de Person. (Sur un ordinateur personnel n'utilisant pas Unix ou Plan 9 mais quelque chose comme MacOS ou VMS, le séparateur de répertoires peut être différent, mais le principe reste le même.) Ne voyez aucune forme de relation entre les

modules en vous basant sur le nom de leur répertoire. C'est simplement une facilité de regroupement et cela n'a aucun effet sur l'héritage, l'accessibilité des variables ou quoi que ce soit d'autres.

Pour ce module, nous n'utiliserons pas l'Exporter puisque nous construirons une classe bien élevée qui n'exportera rien du tout. Pour créer des objets, une classe doit avoir un *constructeur*. Un constructeur ne vous renvoie pas seulement un type normal de donnée, mais un objet tout neuf de cette classe. Cela est fait automagiquement par la fonction bless() dont le seul rôle est d'autoriser l'utilisation d'une référence en tant qu'objet. Rappelez-vous : être un objet ne signifie rien de plus que pouvoir appeler des méthodes à partir de soi-même.

Bien qu'un constructeur puisse porter le nom que l'on veut, la plupart des programmeurs Perl semble les appeler new(). Par contre new() n'est pas un mot réservé et une classe n'a aucune obligation de proposer une méthode portant ce nom. Quelques programmeurs utilisent aussi une fonction avec le même nom que la classe du constructeur.

3.1 Représentation des objets

Pour représenter une structure C ou une classe C++, le mécanisme le plus couramment utilisé en Perl, et de loin, est une table de hachage anonyme. Parce qu'une table de hachage peut contenir un nombre arbitraire de champs tous accessibles par un nom arbitrairement choisi.

Si vous voulez une émulation simple d'une structure, vous devriez écrire quelque chose comme :

```
$rec = {
    name  => "Jason",
    age   => 23,
    peers => [ "Norbert", "Rhys", "Phineas"],
};
```

Si vous préférez, vous pouvez créer un peu de différence visuelle en mettant les clés en majuscules :

```
$rec = {
    NAME  => "Jason",
    AGE   => 23,
    PEERS => [ "Norbert", "Rhys", "Phineas"],
};
```

Et vous utilisez `$rec->{NAME}` pour trouver "Jason" ou `@{ $rec->{PEERS} }` pour obtenir "Norbert", "Rhys", et "Phineas". (Avez-vous remarqué combien de programmeurs de 23 ans semblent s'appeler "Jason" ces temps-ci ? :-)

Ce modèle est parfois utilisé pour des classes bien que la possibilité offerte à n'importe qui à l'extérieur de la classe de modifier directement et impunément les données internes (les données de l'objet) ne soit pas considérée comme le summum d'une programmation de qualité. En général, un objet devrait apparaître comme un truc opaque auquel on accède via des *méthodes de l'objet*. Visuellement, les méthodes permettent de déréférencer une référence en utilisant un nom de fonction plutôt que des crochets ou des accolades.

3.2 L'interface d'une classe

Certains langages disposent d'une interface syntaxique formelle vers les méthodes d'une classe. Perl ne possède rien de tout cela. C'est à vous de lire la documentation de chaque classe. Si vous appelez une méthode non-définie pour un objet, Perl ne dira rien, mais votre programme engendrera une exception durant son exécution. De même, si vous appelez une méthode qui attend un nombre premier comme argument avec un nombre non-premier à la place, vous ne pouvez espérer que le compilateur le détecte. (En fait, vous pouvez toujours espérer, mais cela n'arrivera pas.)

Supposons que l'utilisateur de votre classe Person soit respectueux (quelqu'un qui a lu la documentation expliquant l'interface prescrite). Voici comment il devrait utiliser la classe Person:

```
use Person;

$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( "Norbert", "Rhys", "Phineas" );

push @All_Recs, $him;  # stockage dans un tableau pour plus tard
```

```
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", $him->peers), "\n";

printf "Last rec's name is %s\n", $All_Recs[-1]->name;
```

Comme vous pouvez le voir, l'utilisateur de la classe ne sait pas (ou au moins, n'a pas à faire attention) si l'objet a une implémentation particulière ou une autre. L'interface vers la classe et ses objets se fait exclusivement via des méthodes et c'est la seule chose avec laquelle l'utilisateur travaille.

3.3 Constructeurs et méthodes d'objet

Par contre, *quelqu'un* doit savoir ce qu'il y a dans l'objet. Ce quelqu'un c'est la classe. Elle implémente les méthodes que le programmeur utilise pour accéder à l'objet. Voici le manière d'implémenter la classe Person en utilisant l'idiome standard objet-référence-vers-table-de-hachage. Nous fabriquons un méthode de classe appelée new() en guise de constructeur et trois méthodes d'objet appelées name(), age() et peers() pour cacher l'accès à nos données d'objet dans notre table de hachage anonyme.

```
package Person;
use strict;

################################################
## le constructeur d'objet (version simpliste)  ##
################################################
sub new {
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless($self);           # voir ci-dessous
    return $self;
}

################################################
## méthodes pour accéder aux données d'objet      ##
##                                              ##
## Avec des arguments, elles changent la(les)     ##
## valeur(s)                                    ##
## Sans, elles ne font que la(les) retrouver.     ##
################################################

sub name {
    my $self = shift;
    if (@_) { $self->{NAME} = shift }
    return $self->{NAME};
}

sub age {
    my $self = shift;
    if (@_) { $self->{AGE} = shift }
    return $self->{AGE};
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return @{ $self->{PEERS} };
}

1;  # ainsi le 'require' ou le 'use' réussi
```

Nous avons créé trois méthodes pour accéder aux données de l'objet: name(), age() et peers(). Elles sont similaires. Si elles sont appelées avec un argument, elles stockent la nouvelle valeur du champ. Sans argument, elles retournent la valeur contenu dans le champ correspondant, c'est-à-dire la valeur indexée par cette clé dans la table de hachage.

3.4 En prévision du futur: de meilleurs constructeurs

Bien que pour l'instant vous ne sachiez peut-être pas du tout ce que c'est, il faut faire attention à l'héritage. (Vous pouvez ne pas vous en occuper pour l'instant et n'y revenir que plus tard.) Pour être sûr que tout cela fonctionne correctement, vous devez utiliser la fonction bless() sous sa forme à deux arguments. Le second argument est la classe par laquelle la référence doit être bénie (blessed). Si vous utilisez notre propre classe comme second argument par défaut plutôt que de retrouver la classe qui vous est passée, vous rendez votre constructeur inhéritable.

```
sub new {
    my $class = shift;
    my $self  = {};
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless ($self, $class);
    return $self;
}
```

C'est tout ce qu'il y a à savoir sur les constructeurs. Ces méthodes donnent vie aux objets et retournent à l'utilisateur un truc opaque qui sera utilisé plus tard pour appeler des méthodes.

3.5 Destructeurs

Toutes les histoires ont un début et une fin. Le début de l'histoire d'un objet est son constructeur qui est explicitement appelé à la création de l'objet. Le *destructeur* représente la fin de l'objet. Il est appelé implicitement lorsque l'objet disparaît. Tout code de nettoyage lié à l'objet doit être placé dans le destructeur qui (en Perl) doit s'appeler DESTROY.

Pourquoi les constructeurs peuvent-ils avoir des noms arbitraires et pas les destructeurs ? Parce qu'un constructeur est appelé explicitement ce qui n'est pas le cas du destructeur. La destruction se fait automatiquement via le système de ramasse-miettes (GC) de Perl qui est un GC à base de référence souvent rapide, mais parfois un peu indolent. Pour savoir quoi appeler, Perl impose que le destructeur s'appelle DESTROY. La notion de bon moment pour appeler le destructeur n'est pas très bien définie actuellement en Perl. C'est pourquoi vos destructeurs ne doivent pas dépendre du moment où ils sont appelés.

Pourquoi DESTROY est tout en majuscule ? C'est une convention en Perl que toutes les fonctions entièrement en majuscule peuvent être appelées automatiquement par Perl. Il en existe d'autres qui sont appelées implicitement comme BEGIN, END, AUTOLOAD, plus tous les méthodes utilisées par les objets tied (cravatés ?) et décrites dans *perltie*.

Dans les langages réellement orientés objets, l'utilisateur n'a pas à se préoccuper d'appeler le destructeur. Cela arrive automagiquement quand il faut. Dans les langages de bas niveau sans ramasse-miettes du tout, il n'y a aucun moyen d'automatiser cela. C'est donc au programmeur d'appeler explicitement le destructeur pour nettoyer la mémoire en croisant les doigts pour que ce soit le bon moment. Au contraire de C++, un destructeur est très rarement nécessaire en Perl et même quand il l'est, il n'est pas nécessaire de l'appeler explicitement. Dans le cas de notre classe Person, nous n'avons pas besoin de destructeur, car Perl s'occupe tout seul de pas mal de choses, comme la libération de la mémoire.

La seule situation où le ramasse-miettes de Perl échoue, c'est en présence de références circulaires comme celle-ci :

```
$this->{WHATEVER} = $this;
```

Dans ce cas, vous devez détruire manuellement la référence circulaire si vous voulez que votre programme n'ait pas de fuites mémoire. C'est l'idéal. Par contre, assurez-vous que lorsque votre programme se termine, tous les destructeurs de ces objets sont appelés. Ainsi vous êtes sûr qu'un objet sera détruit proprement, sauf dans le cas où votre programme ne se termine jamais. (Si vous faites tourner Perl à l'intérieur d'une autre application, cette passe complète du ramasse-miettes peut arriver plus souvent – par exemple, à la fin de chaque fil d'exécution.)

3.6 Autres méthodes d'objets

Les méthodes dont nous avons parlé jusqu'à présent sont soit des constructeurs, soit de simples méthodes d'accès aux données stockées dans l'objet. Cela ressemble un peu aux données membres des objets du monde C++, sauf que dans ce cas les "étrangers" n'y accèdent pas comme des données. À la place, ils doivent y accéder indirectement via des méthodes. Règle importante : en Perl, l'accès aux données d'un objet ne devrait se faire *que* par l'intermédiaire des méthodes.

Perl n'impose aucune restriction sur qui utilise quelles méthodes. La distinction publique/privée n'est pas syntaxique. C'est une convention. (Sauf si vous utilisez le module Alias décrit plus bas dans Les données membres comme des variables.) De temps en temps, vous verrez des méthodes dont le nom commence ou finit par un ou deux caractères de soulignement. C'est une convention signifiant que ces méthodes sont privées et utilisables uniquement par cette classe, et éventuellement des classes proches ou ses sous-classes. Mais Perl ne fait pas respecter cette distinction. C'est au programmeur de bien se comporter.

Il n'y a aucune raison de limiter l'usage des méthodes à l'accès aux données de l'objet. Une méthode peut faire tout ce qu'on veut. Le point clé est de savoir si elle est appelée comme méthode de classe ou comme méthode d'objet. Supposons que vous vouliez écrire une méthode objet qui fasse plus que donner l'accès en lecture et/ou écriture à un champ particulier :

```
sub exclaim {
    my $self = shift;
    return sprintf "Hello, Je suis %s, j'ai %d ans, je travaille avec %s",
        $self->{NAME}, $self->{AGE}, join(", ", @{$self->{PEERS}});
}
```

ou quelque chose comme ça :

```
sub joyeux_anniversaire {
    my $self = shift;
    return ++$self->{AGE};
}
```

Certains diront qu'il vaut mieux les écrire comme ça :

```
sub exclaim {
    my $self = shift;
    return sprintf "Hello, Je suis %s, j'ai %d ans, je travaille avec %s",
        $self->name, $self->age, join(", ", $self->peers);
}

sub joyeux_anniversaire {
    my $self = shift;
    return $self->age( $self->age() + 1 );
}
```

Mais comme ces méthodes sont toutes exécutées dans la classe elle-même, ce n'est pas critique. Le choix est une histoire de compromis. L'utilisation directe de la table de hachage est plus rapide (d'un ordre de grandeur pour être précis). Mais l'usage des méthodes (c'est à dire l'interface externe) protège non seulement l'utilisateur de votre classe, mais aussi vous-même des éventuels changements dans la représentation interne de vos données objet.

4 Données de classe

Que dire des données de classe, les données communes à tous les objets de la classe ? À quoi peuvent-elles vous servir ? Supposons que dans votre classe Person, vous vouliez garder une trace du nombre total de personnes. Comment implémenter cela ?

Vous *pourriez* en faire une variable globale appelée $Person::Census. Mais la seule justification valable pour agir ainsi serait de donner au gens la possibilité d'accéder directement à votre donnée de classe. Ils pourraient utiliser $Person::Census pour faire ce qu'ils veulent. Peut-être est-ce ce qu'il vous faut. Vous pourriez même en faire une variable exportée. Pour être exportable, une variable doit être globale (au paquetage). Si c'était un module traditionnel plutôt qu'un module orienté objet, c'est comme cela qu'il faudrait faire.

Bien qu'étant l'approche utilisée dans la plupart des modules traditionnels, cette manière de faire est considérée comme mauvaise dans une approche objet. Dans un module objet, vous devriez pouvoir séparer l'interface de l'implémentation. Il faut donc proposer des méthodes de classe pour accéder aux données de la classe comme le font les méthodes objets pour accéder aux données des objets.

Donc, vous *pourriez* garder $Census comme une variable globale en espérant que les autres respecteront le contrat de modularité en n'accédant pas directement à son implémentation. Vous pourriez même être plus rusé (voire retors) et faire de $Census un objet « tied » comme décrit dans *perltie* et donc intercepter tous les accès.

Le plus souvent, il vous suffira de déclarer votre donnée de classe avec une portée lexicale locale au fichier. Pour cela, il vous suffit de mettre la ligne suivante au début de votre fichier :

```
my $Census = 0;
```

Normalement, la portée d'une variable my() se termine en même temps que le bloc dans lequel elle est déclarée (dans notre cas ce serait l'ensemble du fichier requis (par require()) ou utilisé (par use())), mais en fait le mécanisme de gestion des variables lexicales implémenté par Perl garantit que la variable ne sera pas désallouée et restera accessible à toutes les fonctions ayant la même portée. Par contre, cela ne fonctionne pas avec les variables globales auxquelles on donne une valeur temporaire via local().

Indépendamment de la méthode choisie ($Census comme variable globale du paquetage ou avec une portée lexicale locale au fichier), vous devrez modifier le constructeur $Person::new() de la manière suivante :

```
sub new {
    my $class = shift;
    my $self  = {};
    $Census++;
    $self->{NAME}   = undef;
    $self->{AGE}    = undef;
    $self->{PEERS}  = [];
    bless ($self, $class);
    return $self;
}

sub population {
    return $Census;
}
```

Ceci étant fait, nous avons maintenant besoin d'un destructeur afin de décrémenter $Census lorsqu'un objet Person est détruit. Voici ce destructeur :

```
sub DESTROY { --$Census }
```

Remarquez qu'il n'y a pas de mémoire à désallouer dans le destructeur ! C'est Perl qui s'en occupe pour vous tout seul.

Pour faire tout cela vous pouvez aussi utiliser le module Class::Data::Inheritable disponible sur CPAN.

4.1 Accès aux données de classe

Il s'avère que ce n'est pas vraiment une bonne chose que de manipuler les données de classe. Un bonne règle est : *vous ne devriez jamais référencer directement des données de classe directement dans une méthode objet*. Car, sinon, vous ne construisez pas une classe héritable. L'objet doit être le point de rendez-vous pour toutes les opérations, et en particulier depuis les méthodes objets. Les données globales (les données de classe) peuvent être dans le mauvais paquetage du point de vue des classes héritées. En Perl, les méthodes s'exécutent dans le contexte de la classe où elles sont définies et *pas* dans celui de l'objet qui les a appelées. Par conséquent, la visibilité des variables globales d'un paquetage dans les méthodes n'est pas liée à l'héritage.

Ok, supposons qu'une autre classe emprunte (en fait hérite de) la méthode DESTROY telle qu'elle est définie précédemment. Lorsque ses objets sont détruits, c'est la variable originale $Census qui est modifiée et non pas celle qui est dans l'espace de noms du paquetage de la nouvelle classe. La plupart du temps, cela ne correspond pas à ce que vous vouliez.

Bon, voici comment y remédier. Nous allons stocker une référence à la variable dans la valeur associée à la clé "_CENSUS" de la table de hachage de l'objet. Pourquoi un caractère de soulignement au début ? Principalement parce que cela évoque une notion magique à un programmeur C. C'est en fait un moyen mnémotechnique pour nous souvenir que ce champ est spécial et qu'il ne doit pas être utilisé comme une donnée publique telle que NAME, AGE ou PEERS. (En raison de l'utilisation du pragma strict, dans les versions antérieures à la 5.004, nous devons entourer de guillemets le nom du champ.)

```
sub new {
    my $class = shift;
    my $self  = {};
    $self->{NAME}     = undef;
    $self->{AGE}      = undef;
    $self->{PEERS}    = [];
    # données "privées"
```

```
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub population {
    my $self = shift;
    if (ref $self) {
        return ${ $self->{"_CENSUS"} };
    } else {
        return $Census;
    }
}

sub DESTROY {
    my $self = shift;
    -- ${ $self->{"_CENSUS"} };
}
```

4.2 Méthodes de déboggage

Une classe propose souvent un mécanisme de déboggage. Par exemple, vous pourriez vouloir voir quand les objets sont créés et détruits. Pour cela, il vous faut ajouter un variable de déboggage de portée lexicale limitée au fichier. Nous utiliserons aussi le module standard Carp pour émettre nos avertissements (warnings) et nos messages d'erreur. Ainsi ces messages s'afficheront avec le nom et le numéro de la ligne du fichier de l'utilisateur plutôt qu'avec ceux de notre fichier. Si nous les voulons de notre point de vue, il nous suffit d'utiliser respectivement die() et warn() plutôt que croak() et carp()

```
use Carp;
my $Debugging = 0;
```

Ajoutons maintenant une nouvelle méthode de classe pour accéder à cette variable.

```
sub debug {
    my $class = shift;
    if (ref $class)  { confess "Class method called as object method" }
    unless (@_ == 1) { confess "usage: CLASSNAME->debug(level)" }
    $Debugging = shift;
}
```

Modifions DESTROY pour afficher un petit quelque chose lorsqu'un objet disparaît:

```
sub DESTROY {
    my $self = shift;
    if ($Debugging) { carp "Destroying $self " . $self->name }
    -- ${ $self->{"_CENSUS"} };
}
```

Il est concevable d'avoir un mécanisme de déboggage par objet. Afin de pouvoir utiliser les deux appels suivants :

```
Person->debug(1);   # toute la classe
$him->debug(1);     # juste un objet
```

Nous avons donc besoin de rendre "bimodale" notre méthode debug afin qu'elle fonctionne à la fois sur la classe *et* sur les objets. Modifions donc debug() et DESTROY comme suit :

```
sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self))  {
        $self->{"_DEBUG"} = $level;         # juste moi-même
    } else {
        $Debugging        = $level;         # toute la classe
    }
}

sub DESTROY {
    my $self = shift;
    if ($Debugging || $self->{"_DEBUG"}) {
        carp "Destroying $self " . $self->name;
    }
    -- ${ $self->{"_CENSUS"} };
}
```

Que se passe-t-il si une classe dérivée (que nous appellerons Employee) hérite de ces méthodes depuis la classe de base Person ? Eh bien, `Employee->debug()`, lorsqu'elle est appelée en tant que méthode de classe, modifie $Person::Debugging et non $Employee::Debugging.

4.3 Destructeurs de classes

Le destructeur d'objet traite la disparition de chaque objet. Mais parfois, vous devez faire un peu de nettoyage lorsque toute la classe disparaît. Ce qui n'arrive actuellement qu'à la fin du programme. Pour faire un *destructeur de classe*, créez une fonction END dans le paquetage de la classe. Elle fonctionne exactement comme la fonction END des modules traditionnels. Ce qui signifie qu'elle est appelée lorsque votre programme se termine, à moins qu'il n'effectue un 'exec' ou qu'il meurt sur un signal non capté. Par exemple :

```
sub END {
    if ($Debugging) {
        print "All persons are going away now.\n";
    }
}
```

Quand le programme se termine, tous les destructeurs de classes (les fonctions END) sont exécutés dans l'ordre inverse de leur chargement (LIFO).

4.4 La documentation de l'interface

Jusqu'ici nous n'avons exhibé que *l'implémentation* de la classe Person. Son *interface* doit être sa documentation. Habituellement cela signifie d'ajouter du pod ("plain old documentation") dans le même fichier. Dans le cas de notre exemple Person, nous devons placer la documentation suivante quelque part dans le fichier Person.pm. Bien que cela ressemble à du code, ce n'en est pas. C'est de la documentation intégrée utilisable par des programmes comme pod2man, pod2html ou pod2text. Le compilateur Perl ne tient pas compte des parties pod. À l'inverse, les traducteurs pod ne tiennent pas compte des parties de code. Voici un exemple de pod décrivant notre interface :

```
=head1 NAME

Person - classe pour implémenter des gens

=head1 SYNOPSIS

 use Person;
```

```
    ######################
    # méthodes de classe #
    ######################
    $ob    = Person->new;
    $count = Person->population;

    ########################################
    # méthodes d'accès aux données d'objets   #
    ########################################

    ### accès en lecture ###
        $who   = $ob->name;
        $years = $ob->age;
        @pals  = $ob->peers;

    ### accès en écriture ###
        $ob->name("Jason");
        $ob->age(23);
        $ob->peers( "Norbert", "Rhys", "Phineas" );

    ############################
    # autres méthodes d'objets #
    ############################

    $phrase = $ob->exclaim;
    $ob->joyeux_anniversaire;

=head1 DESCRIPTION

La classe Person permet de blah, blah, blah...
```

C'est donc tout ce qui concerne l'interface et non pas l'implémentation. Un programmeur qui ouvre le module pour jouer avec toutes les astuces d'implémentation qui sont cachées derrière le contrat d'interface rompt la garantie et vous n'avez pas à vous préoccuper de son sort.

5 Agrégation

Supposons que, plus tard, vous vouliez modifier la classe pour avoir une meilleure gestion des noms. Par exemple, en gérant le prénom, le nom de famille, mais aussi les surnoms et les titres. Si les utilisateurs de votre classe Person y accèdent proprement via l'interface documentée, vous pouvez alors changer sans risque l'implémentation sous-jacente. S'ils ne l'ont pas fait, ils ont tout perdu, mais c'est leur faute puisqu'en rompant le contrat, ils perdent la garantie.

Nous allons donc créer une nouvelle classe appelée Fullname. À quoi ressemblera cette classe Fullname ? Pour pouvoir répondre, nous devons d'abord examiner comment nous comptons l'utiliser. Par exemple:

```
$him = Person->new();
$him->fullname->title("St");
$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
printf "His normal name is %s\n", $him->name;
printf "But his real name is %s\n", $him->fullname->as_string;
```

Ok. Changeons Person::new() pour qu'il accepte un champ fullname :

```
sub new {
    my $class = shift;
    my $self  = {};
    $self->{FULLNAME} = Fullname->new();
    $self->{AGE}      = undef;
    $self->{PEERS}    = [];
    $self->{"_CENSUS"} = \$Census;
    bless ($self, $class);
    ++ ${ $self->{"_CENSUS"} };
    return $self;
}

sub fullname {
    my $self = shift;
    return $self->{FULLNAME};
}
```

Puis, pour accepter le vieux code, définissons Person::name() de la manière suivante :

```
sub name {
    my $self = shift;
    return $self->{FULLNAME}->nickname(@_)
      ||   $self->{FULLNAME}->christian(@_);
}
```

Voici maintenant la classe Fullname. Là encore, nous utilisons une table de hachage pour stocker les données associées à des méthodes avec le nom qui va bien pour y accéder :

```
package Fullname;
use strict;

sub new {
    my $class = shift;
    my $self  = {
        TITLE       => undef,
        CHRISTIAN   => undef,
        SURNAME     => undef,
        NICK        => undef,
    };
    bless ($self, $class);
    return $self;
}

sub christian {
    my $self = shift;
    if (@_) { $self->{CHRISTIAN} = shift }
    return $self->{CHRISTIAN};
}

sub surname {
    my $self = shift;
    if (@_) { $self->{SURNAME} = shift }
    return $self->{SURNAME};
}

sub nickname {
    my $self = shift;
    if (@_) { $self->{NICK} = shift }
    return $self->{NICK};
}
```

```
sub title {
    my $self = shift;
    if (@_) { $self->{TITLE} = shift }
    return $self->{TITLE};
}

sub as_string {
    my $self = shift;
    my $name = join(" ", @$self{'CHRISTIAN', 'SURNAME'});
    if ($self->{TITLE}) {
        $name = $self->{TITLE} . " " . $name;
    }
    return $name;
}

1;
```

Pour finir, voici un programme de test :

```
#!/usr/bin/perl -w
use strict;
use Person;
sub END { show_census() }

sub show_census ()  {
    printf "Current population: %d\n", Person->population;
}

Person->debug(1);

show_census();

my $him = Person->new();

$him->fullname->christian("Thomas");
$him->fullname->surname("Aquinas");
$him->fullname->nickname("Tommy");
$him->fullname->title("St");
$him->age(1);

printf "%s is really %s.\n", $him->name, $him->fullname->as_string;
printf "%s's age: %d.\n", $him->name, $him->age;
$him->happy_birthday;
printf "%s's age: %d.\n", $him->name, $him->age;

show_census();
```

6 Héritage

Tous les systèmes de programmation orienté objets incluent sous une forme ou une autre la notion d'héritage. L'héritage permet à une classe d'englober une autre classe afin d'éviter de ré-écrire la même chose plusieurs fois. C'est en rapport avec la réutilisation logicielle et donc avec la paresse qui est, comme chacun sait, la vertu principale d'un programmeur. (Le mécanisme d'import/export des modules traditionnels est aussi une forme de réutilisation de code, mais beaucoup plus simple que le vrai héritage qu'on trouve dans les modules objets.)

Parfois la syntaxe de l'héritage est construite au coeur du langage, mais ce n'est pas toujours le cas. Perl n'a aucune syntaxe spéciale pour spécifier la classe (ou les classes) dont on hérite. À la place, tout est fait purement sémantiquement. Chaque paquetage peut contenir une variable appelée @ISA qui pilote l'héritage (des méthodes). Si vous essayez d'appeler une

méthode d'un objet ou d'une classe et que cette méthode n'est pas trouvée dans le paquetage de l'objet, Perl trouve dans @ISA le nom d'autres paquetages où chercher la méthode manquante.

Comme les variables spéciales de paquetages reconnues par Exporter (telles que @EXPORT, @EXPORT_OK, @EXPORT_FAIL, %EXPORT_TAGS, et $VERSION), le tableau @ISA *doit* être une variable de portée globale au paquetage et non une variable de portée lexicale limitée au fichier et créée par my(). La plupart des classes n'ont qu'un seul nom dans leur tableau @ISA. C'est ce que nous appellerons de l'"Héritage Simple" ou HS pour faire court.

Examinons la classe suivante :

```
package Employee;
use Person;
@ISA = ("Person");
1;
```

Ce n'est pas grand chose, hein ? Tout ce que ça fait c'est de charger une autre classe et d'indiquer qu'on hérite des méthodes de cette autre classe, si nécessaire. Nous n'avons spécifié aucune de ses propres méthodes. Nous obtenons donc un Employee qui se comporte exactement comme une Person.

Une telle classe vide est appelée une "sous-classe vide de test". C'est une classe qui ne fait rien si ce n'est d'hériter d'une classe de base. Si la classe originale est correctement conçue, alors la nouvelle classe dérivée peut être utilisée en remplacement de celle d'origine. Cela veut dire que vous pouvez écrire un programme comme celui-ci :

```
use Employee;
my $empl = Employee->new();
$empl->name("Jason");
$empl->age(23);
printf "%s is age %d.\n", $empl->name, $empl->age;
```

Par correctement conçue, nous entendons toujours utiliser bless() dans sa forme à deux arguments, éviter l'accès direct aux données globales et ne rien exporter. En regardant la fonction Person::new() que nous avons définie précédemment, vous pourrez vérifier tout cela. Quelques données du paquetage sont utilisées dans le constructeur, mais leur référence est stockée dans l'objet lui-même et toutes les autres méthodes accèdent à ces données via ces références. Ça doit donc fonctionner.

Qu'entendons-nous par la fonction Person::new(), – n'est-ce pas une méthode ? En principe, oui. Une méthode est simplement une fonction qui attend comme premier paramètre soit un nom de classe (de paquetage), soit un objet (une référence bénie). Person::new() est la fonction que les deux méthodes `Person->new()` et `Employee->new()` appellent. Bien qu'un appel à une méthode ressemble à un appel de fonction, il faut bien comprendre que ce n'est pas exactement la même chose. Si vous les considérez comme identiques, vous vous retrouverez très rapidement avec des programmes complètement boggués. Tout d'abord, les conventions d'appel sous-jacentes sont différentes: l'appel à une méthode ajoute implicitement un argument supplémentaire. Ensuite, les appels de fonctions n'utilisent pas les mécanismes d'héritage mis en oeuvre pour les méthodes.

```
Appel de méthode        Appel de fonction résultant
-----------             -----------------------
Person->new()           Person::new("Person")
Employee->new()         Person::new("Employee")
```

Donc, n'utilisez pas un appel de fonction à la place d'un appel de méthode.

Si un Employee est juste une Person, cela n'est pas très utile. Ajoutons donc quelques méthodes. Nous allons ajouter à nos Employee des données d'objet pour leur salaire (salary), leur identification (ID number) et leur date d'embauche (start date).

Si vous êtes fatigué de créer ces méthodes d'accès toutes bâties sur le même principe, ne désespérez pas. Plus tard, nous décrirons différentes techniques pour automatiser cela.

```
sub salary {
    my $self = shift;
    if (@_) { $self->{SALARY} = shift }
    return $self->{SALARY};
}
```

```
sub id_number {
    my $self = shift;
    if (@_) { $self->{ID} = shift }
    return $self->{ID};
}

sub start_date {
    my $self = shift;
    if (@_) { $self->{START_DATE} = shift }
    return $self->{START_DATE};
}
```

6.1 Polymorphisme

Qu'arrive-t-il lorsqu'une classe dérivée et sa classe de base définissent toutes deux la même méthode ? La méthode utilisée est la version de la classe dérivée. Par exemple, supposons que nous voulions que la méthode peers() agisse différemment lorsque nous l'appelons pour un Employee. Au lieu de renvoyer simplement la liste des peers, nous aimerions obtenir une autre chaîne. De telle sorte que :

```
$empl->peers("Peter", "Paul", "Mary");
printf "His peers are: %s\n", join(", ", $empl->peers);
```

produira :

```
His peers are: PEON=PETER, PEON=PAUL, PEON=MARY
```

Pour obtenir ce résultat, il faut ajouter la définition suivante dans le fichier Employee.pm :

```
sub peers {
    my $self = shift;
    if (@_) { @{ $self->{PEERS} } = @_ }
    return map { "PEON=\U$_" } @{ $self->{PEERS} };
}
```

Nous venons d'illustrer le concept de *polymorphisme*. Nous avons réutilisé la forme et le comportement d'un objet existant, puis nous l'avons modifié pour l'adapter à nos besoins. C'est une forme de Paresse. (Être polymorphe c'est aussi ce qui vous arrive quand un magicien décide que vous seriez mieux sous la forme d'une grenouille.)

Supposons maintenant que nous voulions un appel de méthode qui déclenche à la fois la version de la classe dérivée (aussi appelée sous-classe) et la version de la classe de base (aussi appelée super-classe). En pratique, c'est le cas des constructeurs et des destructeurs et probablement celui de la méthode debug() dont nous avons parlé précédemment.

Pour cela, ajoutons ce qui suit dans Employee.pm :

```
use Carp;
my $Debugging = 0;

sub debug {
    my $self = shift;
    confess "usage: thing->debug(level)"    unless @_ == 1;
    my $level = shift;
    if (ref($self))  {
        $self->{"_DEBUG"} = $level;
    } else {
        $Debugging = $level;            # toute la classe
    }
    Person::debug($self, $Debugging);   # à ne pas faire !
}
```

Comme vous pouvez le constater nous appelons directement la fonction debug() du paquetage Person. Mais cela est beaucoup trop spécifique pour être une bonne conception. En effet, que se passe-t-il si Person n'a pas de fonction debug(), mais en hérite d'ailleurs ? Il aurait été préférable de dire :

```
Person->debug($Debugging);
```

Mais même cela est encore de trop bas niveau. Il vaudrait mieux dire :

```
$self->Person::debug($Debugging);
```

C'est une drôle de manière pour demander de commencer la recherche de la méthode debug() à partir de la classe Person. Cette stratégie est plus souvent utilisée pour des méthodes d'objets que pour des méthodes de classe.

Cela laisse encore à désirer. Nous avons codé en dur le nom de notre super-classe. Ce qui n'est pas bon, en particulier si nous changeons la classe dont on hérite ou si on en ajoute d'autres. Heureusement, la pseudo-classe SUPER a été créée pour nous sauver.

```
$self->SUPER::debug($Debugging);
```

Comme ça, la recherche s'effectue dans tous les classes du tableau @ISA. Cela n'a un sens *que* lors de l'appel d'une méthode. N'essayez pas d'utiliser SUPER dans un autre contexte parce qu'elle n'existe que dans le cas d'appel à des méthodes redéfinies. Notez aussi que `SUPER` se réfère à la super-classe du package courant et *non* à celle de `$self`.

Tout cela est devenu un peu compliqué maintenant. Avons-nous tout fait comme il fallait ? Comme précédemment, pour vérifier que notre classe est correctement conçue, nous allons utiliser une sous-classe vide de test. Puisque nous avons déjà une classe Employee que nous voulons tester, notre classe de test sera dérivée de Employee. En voici une :

```
package Boss;
use Employee;        # :-)
@ISA = qw(Employee);
```

Et voici le programme de test :

```
#!/usr/bin/perl -w
use strict;
use Boss;
Boss->debug(1);

my $boss = Boss->new();

$boss->fullname->title("Don");
$boss->fullname->surname("Pichon Alvarez");
$boss->fullname->christian("Federico Jesus");
$boss->fullname->nickname("Fred");

$boss->age(47);
$boss->peers("Frank", "Felipe", "Faust");

printf "%s is age %d.\n", $boss->fullname->as_string, $boss->age;
printf "His peers are: %s\n", join(", ", $boss->peers);
```

Son exécution nous montre que tout marche bien. Si vous voulez afficher votre objet sous une forme lisible comme le fait la commande 'x' du debogguer, vous pouvez utiliser le module Data::Dumper disponible au CPAN :

```
use Data::Dumper;
print "Here's the boss:\n";
print Dumper($boss);
```

Ce qui devrait vous afficher quelque chose comme :

```
Here's the boss:
$VAR1 = bless( {
     _CENSUS => \1,
     FULLNAME => bless( {
                          TITLE => 'Don',
                          SURNAME => 'Pichon Alvarez',
                          NICK => 'Fred',
                          CHRISTIAN => 'Federico Jesus'
                        }, 'Fullname' ),
     AGE => 47,
     PEERS => [
                'Frank',
                'Felipe',
                'Faust'
              ]
   }, 'Boss' );
```

Heu... Il manque quelque chose. Où sont les champs salary, start_date et ID ? Nous ne leur avons jamais donné de valeur, même pas undef, donc ils n'apparaissent pas dans les clés de la table de hachage. La classe Employee ne définit pas sa propre méthode new() et la méthode new() de la classe Person ne sait rien des Employee (elle ne le doit pas : une bonne conception orientée objet suppose qu'une sous-classe a le droit de connaître les super-classes dont elle hérite, mais jamais le contraire). Créons donc Employee::new() comme suit :

```
sub new {
    my $class = shift;
    my $self  = $class->SUPER::new();
    $self->{SALARY}        = undef;
    $self->{ID}            = undef;
    $self->{START_DATE}    = undef;
    bless ($self, $class);          # reconsecrate
    return $self;
}
```

À présent, si vous afficher un objet Employee ou Boss, vous verrez ces nouveaux champs.

6.2 Héritage multiple

Bon, au risque d'ennuyer les gourous OO et de perturber les débutants, il est temps d'avouer que le système objet de Perl propose la notion très controversée d'héritage multiple (ou HM pour faire court). Cela signifie qu'au lieu d'hériter d'une classe qui elle-même peut hériter d'une autre classe et ainsi de suite, vous pouvez hériter directement de plusieurs classes parentes. Il est vrai que le HM peut rendre les choses confuses, même si en Perl cela l'est un peu moins qu'avec des langages OO douteux comme le C++.

La manière dont cela fonctionne est vraiment très simple : il suffit de mettre plus d'un nom de paquetage dans le tableau @ISA. Lorsque Perl cherche une méthode pour votre objet, il regarde dans chacun de ces paquetages dans l'ordre. C'est une recherche récursive en profondeur d'abord (par défaut ; voir *mro* pour utiliser d'autres ordres de recherche de méthodes). Supposons un ensemble de tableaux @ISA comme :

```
@First::ISA    = qw( Alpha );
@Second::ISA   = qw( Beta );
@Third::ISA    = qw( First Second );
```

Si vous avez un objet de la classe Third :

```
my $ob = Third->new();
$ob->spin();
```

Comment allons-nous trouver la méthode find() (ou une méthode new()) ? Puisque la recherche s'effectue en profondeur d'abord, les classes seront explorées dans l'ordre suivant : Third, First, Alpha, Second et enfin Beta.

En pratique, très peu de modules connus font usage de l'HM. La plupart préfèrent choisir l'inclusion d'une classe dans une autre plutôt que l'HM. C'est pourquoi notre objet Person *contient* un objet Fullname. Cela ne veut pas dire qu'il en *est* un.

Par contre, il y a un domaine où l'HM de Perl est très répandu : lorsqu'une classe emprunte des méthodes à une autre classe. C'est assez commun spécialement pour quelques classes "pas très objet" comme Exporter, DynaLoader, AutoLoader et SelfLoader. Ces classes ne proposent pas de constructeurs. Elles n'existent que pour vous permettre d'hériter de leurs méthodes. (Le choix de l'héritage plutôt que de l'importation traditionnelle n'est pas entièrement clair.)

Par exemple, voici le tableau @ISA du module POSIX :

```
package POSIX;
@ISA = qw(Exporter DynaLoader);
```

Ce module POSIX n'est pas vraiment un module objet pas plus qu'un Exporter ou un DynaLoader. Ces derniers ne font que prêter leur comportement à POSIX.

Pourquoi n'utilise-t-on pas plus l'HM pour les méthodes objet ? La raison principale réside dans les effets de bord complexes. Par exemple, votre graphe (ce n'est pas obligatoirement un arbre) d'héritage peut converger vers la même classe de base. Bien que Perl empêche l'héritage récursif, avoir des parents qui se réfèrent à un même ancêtre n'est pas interdit aussi incestueux que cela paraisse. Que se passe-t-il si dans notre classe Third nous voulons que sa méthode new() appelle aussi les constructeurs de ses deux classes parentes ? La notation SUPER ne trouvera que la méthode de la première classe. D'autre part, qu'arrive-t-il si les classes Alpha et Beta ont toutes les deux un ancêtre commun ? disons Nought. Si vous parcourez l'arbre d'héritage afin d'appeler les méthodes cachées, vous finirez par appeler deux fois la méthode Nought::new(), ce qui est sûrement une mauvaise chose.

6.3 UNIVERSAL: la racine de tous les objets

Ne serait-ce pas pratique si tous les objets héritaient d'une même classe de base ? Ainsi, on pourrait avoir des méthodes communes à tous les objets sans avoir à ajouter explicitement cette classe dans chaque tableau @ISA. En fait, cela existe déjà. Vous ne le voyez pas, mais Perl suppose tacitement qu'il y a un élément supplémentaire à la fin de @ISA : la classe UNIVERSAL. Dans la version 5.003, il n'y avait aucune méthode prédéfinie dans cette classe, mais vous pouviez y mettre tout ce que vouliez.

Maintenant, depuis la version 5.004 (ou quelques versions subversives comme la 5.003_08), UNIVERSAL contient déjà des méthodes. Elles sont incluses dans votre binaire Perl et ne consomment donc aucun temps supplémentaire en chargement. Ces méthodes prédéfinies comprennent isa(), can() et VERSION(). isa() vous permet de savoir si un objet (respectivement une classe) "est" un autre objet (respectivement une autre classe) sans que vous ayez à parcourir vous-même la hiérarchie d'héritage :

```
$has_io = $fd->isa("IO::Handle");
$itza_handle = IO::Socket->isa("IO::Handle");
```

La méthode can(), appelée à partir d'un objet ou d'une classe, vous indique si la chaîne passée en argument est le nom d'une méthode appelable de cette classe. En fait, elle vous renvoie même une référence vers la fonction de cette méthode :

```
$his_print_method = $obj->can('as_string');
```

Finalement, la méthode VERSION vérifie que la classe (ou la classe de l'objet) possède une variable globale appelée $VERSION dont la valeur est suffisamment grande. Exemple :

```
Some_Module->VERSION(3.0);
$his_vers = $ob->VERSION();
```

En général, vous n'avez pas à appeler VERSION vous-même. (Souvenez-vous qu'un nom de fonction tout en majuscule est une convention Perl pour indiquer que cette fonction est parfois appelée automatiquement par Perl.) Dans le cas de VERSION, cela arrive quand vous dites :

```
use Some_Module 3.0;
```

Si vous voulez ajouter un contrôle de version à votre classe Person, ajoutez juste dans Person.pm :

```
our $VERSION = '1.1';
```

Et vous pouvez donc dire dans Employee.pm :

```
use Person 1.1;
```

Et il sera sûr que vous avez au moins ce numéro de version. Ce n'est pas la même chose que d'exiger un numéro de version précis. Actuellement, aucun mécanisme n'existe pour installer simultanément de multiples versions d'un même module. Lamentable.

6.4 Approndissements de quelques détails de UNIVERSAL

Il est tout à fait possible (même si c'est le plus souvent contre-indiqué) d'ajouter de noms de packages dans `@UNIVERSAL::ISA`. Ces packages seront alors implicitement hérités pour toutes les classes, comme l'est `UNIVERSAL`. En revanche, ni `UNIVERSAL` ni aucun de ses parents ne sont des classes de base explicites des objets. Essayons de clarifier cela en posant ce qui suit :

```
@UNIVERSAL::ISA = ('REALLYUNIVERSAL');

package REALLYUNIVERSAL;
sub special_method { return "123" }

package Foo;
sub normal_method { return "321" }
```

L'appel à `Foo->special_method()` retournera "123" mais les appels `Foo->isa('REALLYUNIVERSAL')` et `Foo->isa('UNIVERSAL')` retourneront la valeur fausse.

Même si votre classe utilise un ordre de résolution non-standard pour retrouver ses méthodes tel l'ordre C3 (voir *mro*), la résolution des méthodes via `UNIVERSAL` / `@UNIVERSAL::ISA` restera celle par défaut (en profondeur d'abord et de gauche à droite) et n'arrivera que si votre résolution via C3 n'aboutit pas.

Tout ce qu'on vient d'exposer ci-dessus est sûrement plus compréhensible en étudiant ce qui se passe réellement lors de la recherche d'une méthode et qui ressemble approximativement au pseudo-code suivant :

```
get_mro(class) {
    # recurses down the @ISA's starting at class,
    # builds a single linear array of all
    # classes to search in the appropriate order.
    # The method resolution order (mro) to use
    # for the ordering is whichever mro "class"
    # has set on it (either default (depth first
    # l-to-r) or C3 ordering).
    # The first entry in the list is the class
    # itself.
}

find_method(class, methname) {
    foreach $class (get_mro(class)) {
        if($class->has_method(methname)) {
            return ref_to($class->$methname);
        }
    }
    foreach $class (get_mro(UNIVERSAL)) {
        if($class->has_method(methname)) {
            return ref_to($class->$methname);
        }
    }
    return undef;
}
```

En revanche, le code qui implémente `UNIVERSAL::isa` ne cherche pas dans `UNIVERSAL` lui-même. Il explore uniquement le véritable `@ISA` du package.

7 Autre représentation d'objet

Rien n'oblige à implémenter des objets sous la forme de référence à une table de hachage. Un objet peut-être n'importe quel type de référence tant que cette référence a été bénie (blessed). Il peut donc être une référence à un scalaire, à un tableau ou à du code.

Un scalaire peut suffire si l'objet n'a qu'une valeur à stocker. Un tableau fonctionne dans la plupart des cas, mais rend l'héritage plus délicat puisqu'il vous faut créer de nouveaux indices pour les classes dérivées.

7.1 Des objets sous forme de tableaux

Si l'utilisateur de votre classe respecte le contrat et colle à l'interface déclarée, vous pouvez changer l'interface sous-jacente quand vous le voulez. Voici une autre implémentation qui respecte la même spécification d'interface. Cette fois, pour représenter l'objet, nous utilisons une référence à un tableau plutôt qu'une référence à une table de hachage.

```
package Person;
use strict;

my($NAME, $AGE, $PEERS) = ( 0 .. 2 );

################################################
## le constructeur de Person (version tableau) ##
################################################
sub new {
    my $self = [];
    $self->[$NAME]   = undef;  # this is unnecessary
    $self->[$AGE]    = undef;  # as is this
    $self->[$PEERS]  = [];     # but this isn't, really
    bless($self);
    return $self;
}

sub name {
    my $self = shift;
    if (@_) { $self->[$NAME] = shift }
    return $self->[$NAME];
}

sub age {
    my $self = shift;
    if (@_) { $self->[$AGE] = shift }
    return $self->[$AGE];
}

sub peers {
    my $self = shift;
    if (@_) { @{ $self->[$PEERS] } = @_ }
    return @{ $self->[$PEERS] };
}

1;  # so the require or use succeeds
```

Vous pourriez penser que l'accès au tableau est plus rapide que l'accès à la table de hachage, mais ils sont en fait comparables. Le tableau est *un tout petit peu* plus rapide, mais pas plus de 10 ou 15%, même si vous remplacez les variables comme $AGE par des nombres comme 1. La plus grande différence entre les deux approches est l'utilisation de la mémoire. La représentation par table de hachage prend plus de mémoire que la représentation par tableau parce qu'il faut allouer de la mémoire pour stocker les clés d'accès en plus des valeurs. Par contre, ce n'est pas vraiment mauvais puisque, depuis la version 5.004, le mémoire n'est allouée qu'une seule fois pour une clé donnée indépendamment du nombre de tables de hachage qui utilisent cette clé. Il est même prévu qu'un jour ces différences disparaissent, quand des représentations sous-jacentes efficaces seront inventées.

Ceci étant, le petit gain en vitesse (ainsi que celui en mémoire) est suffisant pour inciter des programmeurs à choisir la représentation par tableau pour des classes simples. Il reste encore un petit problème d'extensibilité. Par exemple, quand vous aurez besoin de créer des sous-classes, vous constaterez que les tables de hachage marchent mieux.

7.2 Des objets sous forme de fermeture (closure)

Utiliser une référence à du code pour représenter un objet ouvre des perspectives fascinantes. Vous pouvez créer une nouvelle fonction anonyme (une fermeture) qui est la seule à pouvoir accéder aux données de l'objet. Parce que vous mettez les données dans une table de hachage anonyme dont la portée lexicale est limitée à la fermeture que vous créez, bénissez et renvoyez comme objet. Les méthodes de cette objet appellent la fermeture comme n'importe quelle subroutine normale en lui passant le champ qu'elles veulent modifier. (Oui, le double appel de fonction est lent, mais si vous voulez de la vitesse, vous ne devriez pas utiliser d'objets du tout, non ? :-)

L'utilisation devrait rester comme précédemment :

```
use Person;
$him = Person->new();
$him->name("Jason");
$him->age(23);
$him->peers( [ "Norbert", "Rhys", "Phineas" ] );
printf "%s is %d years old.\n", $him->name, $him->age;
print "His peers are: ", join(", ", @{$him->peers}), "\n";
```

mais l'implémentation est radicalement (et peut-être même sublimement) différente :

```
package Person;

sub new {
     my $class = shift;
     my $self = {
        NAME  => undef,
        AGE   => undef,
        PEERS => [],
     };
     my $closure = sub {
        my $field = shift;
        if (@_) { $self->{$field} = shift }
        return    $self->{$field};
    };
    bless($closure, $class);
    return $closure;
}

sub name   { &{ $_[0] }("NAME",  @_[ 1 .. $#_ ] ) }
sub age    { &{ $_[0] }("AGE",   @_[ 1 .. $#_ ] ) }
sub peers  { &{ $_[0] }("PEERS", @_[ 1 .. $#_ ] ) }

1;
```

Le concept de fermeture provient de la programmation fonctionnelle et, à ceux qui sont résolument attachés à la programmation procédurale ou à la programmation orientée objet, l'objet caché derrière une référence à du code doit vraisemblablement rester mystérieux. L'objet créé et retourné par la méthode new() n'est plus une référence vers des données comme nous l'avons vu auparavant. C'est une référence à du code anonyme qui a accès à sa propre version des données de l'objet (liaison lexicale et instanciation) qui est stockée dans la variable privée $self. Bien que ce soit la même fonction (NDT: le même code) à chaque fois, il contient une version différente de $self.

Quand une méthode comme `$him->name("Jason")` est appelée, son "zéroième" argument implicite est l'objet appelant – exactement comme pour tous les appels de méthodes. Mais dans notre cas, c'est une référence à notre code (quelque chose comme un pointeur sur fonction en C++, mais avec une liaison vers des variables lexicales). À part l'appeler, on ne peut pas faire grand chose d'une référence vers du code. C'est donc exactement ce que nous faisons en disant `&{$_[0]}`. C'est juste un appel à une fonction et non pas un appel de méthode. Le premier argument est la chaîne "NAME", tous les autres arguments sont ceux qui ont été passés à la méthode.

Lors de l'exécution de la fermeture créée par new(), la référence $self à la table de hachage devient soudainement visible. La fermeture retrouve son premier argument ("NAME" dans ce cas puisque c'est ce que lui passe la méthode name()) et l'utilise comme clé d'accès à cette table de hachage privée qui est cachée dans sa propre version de $self.

Rien ne permet à quiconque d'accéder à ces données cachées en dehors de l'exécution de cette méthode. Ou presque rien : vous *pourriez* utiliser le debogueur en allant pas à pas jusque dans le code de la méthode et voir les données, mais en dehors de cela aucune chance d'y accéder.

Bon, si tout ça n'intéresse pas les adeptes de Scheme, je ne vois ce qui pourrait les intéresser. La transposition de ces techniques en C++, Java ou tout autre langage de conception statique est laissée comme exercice futile à leurs aficionados.

Via la fonction caller(), vous pouvez même ajouter un peu plus de confidentialité en contraignant la fermeture à n'accepter que les appels provenant de son propre paquetage. Cela devrait sans aucun doute satisfaire les plus exigeants...

Vous avez eu ici de quoi satisfaire votre orgueil (la troisième vertu principale du programmeur). Plus sérieusement, l'orgueil est tout simplement la fierté de l'artisan qui vient d'écrire un bout de code bien conçu.

8 AUTOLOAD: les méthodes mandataires

L'auto-chargement (ou autoload) est un moyen d'intercepter l'appel à une méthode non définie. Une subroutine d'auto-chargement peux soit créer une nouvelle fonction au vol, soit en charger une depuis le disque, soit l'évaluer elle-même. Cette stratégie de définition au vol est ce qu'on appelle l'auto-chargement.

Mais ce n'est qu'une approche possible. Dans une autre approche, c'est la méthode d'auto-chargement qui fournit elle-même le service demandée. Utilisée de cette manière, on peut alors considérer cette méthode d'auto-chargement comme une méthode "proxy".

Quand Perl essaie d'appeler une fonction non définie dans un paquetage particulier et que cette fonction n'est pas définie, il cherche alors dans le même paquetage une fonction appelé AUTOLOAD. Si elle existe, elle est appelée avec les mêmes arguments que la fonction originale. Le nom complet de la fonction dans la variable $AUTOLOAD du paquetage. Une fois appelée, la fonction peut faire tout ce qu'elle veut et, entre autres, définir une nouvelle fonction avec le bon nom, puis faire une sorte de `goto` vers elle en s'effaçant de la pile d'appel.

Quel rapport avec les objets ? Après tout, nous ne parlons que de fonctions, pas de méthodes. En fait, puisqu'une méthode n'est qu'une fonction avec un argument supplémentaire et quelques sémantiques sympathiques permettant de la retrouver, nous pouvons aussi utiliser l'auto-chargement pour les méthodes. Perl ne commence la recherche de méthodes par auto-chargement que lorsqu'il a fini l'exploration via @ISA. Certains programmeurs ont même défini une méthode UNIVERSAL::AUTOLOAD pour intercepter tous les appels à des méthodes non définies pour n'importe quelle sorte d'objets.

8.1 Méthodes auto-chargées d'accès aux données

Vous êtes peut-être resté un peu sceptique devant la duplication de code que nous avons exposé tout d'abord dans la classe Person, puis dans la classe Employee. Chaque méthode d'accès aux données de la table de hachage est virtuellement identique. Cela devrait chatouiller votre plus grande vertu de programmeur: l'impatience. Mais votre paresse l'a emporté et vous n'avez rien fait. Heureusement, les méthodes proxy sont le remède à ce problème.

Au lieu d'écrire une nouvelle fonction à chaque fois que nous avons besoin d'un nouveau champ, nous allons utiliser le mécanisme d'auto-chargement pour générer (en fait, simuler) les méthodes au vol. Pour vérifier que nous accédons à un membre valide, nous le rechercherons dans le champ `_permitted` (qui, en anglais, se prononce "under-permitted"). Ce champ est une référence à une table de hachage de portée lexicale limitée au fichier (comme une variable C statique d'un fichier) contenant les champs autorisés et appelée %fields. Pourquoi le caractère de soulignement ? Pour le même raison que celui de _CENSUS : c'est un indicateur qui signifie "à usage interne seulement".

Voici à quoi ressemblent le code d'initialisation et le constructeur de la classe si nous suivons cette approche :

```
package Person;
use Carp;
our $AUTOLOAD;  # it's a package global

my %fields = (
    name        => undef,
    age         => undef,
    peers       => undef,
);
```

```
sub new {
    my $class = shift;
    my $self  = {
        _permitted => \%fields,
        %fields,
    };
    bless $self, $class;
    return $self;
}
```

Si nous voulons spécifier des valeurs par défaut pour nos champs, nous pouvons remplacer les `undef` dans la table de hachage %fields.

Avez-vous remarqué comment nous stockons la référence à nos données de classe dans l'objet lui-même ? Souvenez-vous qu'il est fondamental d'accéder aux données de classe à travers l'objet lui-même plutôt que d'utiliser directement %fields dans les méthodes. Sinon vous ne pourrez pas convenablement hériter.

La vraie magie réside dans notre méthode proxy qui gérera tous les appels à des méthodes non définies pour des objets de la classe Person (ou des sous-classes de Person). Elle doit s'appeler AUTOLOAD. Encore une fois, son nom est tout en majuscule parce qu'elle est implicitement appelée par Perl et non pas directement par l'utilisateur.

```
sub AUTOLOAD {
    my $self = shift;
    my $type = ref($self)
                or croak "$self is not an object";

    my $name = $AUTOLOAD;
    $name =~ s/.*://;   # strip fully-qualified portion

    unless (exists $self->{_permitted}->{$name} ) {
        croak "Can't access `$name' field in class $type";
    }

    if (@_) {
        return $self->{$name} = shift;
    } else {
        return $self->{$name};
    }
}
```

Assez chouette, non ? La seule chose à faire pour ajouter de nouveaux champs est de modifier %fields. Il n'y a aucune nouvelle fonction à écrire.

J'aurais même pu supprimer complètement le champ `_permitted`, mais je voulais illustrer comment stocker une référence à une donnée de classe dans un objet de manière à ne pas accéder à cette donnée directement dans les méthodes.

8.2 Méthodes d'accès aux données auto-chargées et héritées

Qu'en est-il de l'héritage ? Pouvons-nous définir la classe Employee de la même manière ? Oui, si nous faisons un peu attention.

Voici comment faire :

```
package Employee;
use Person;
use strict;
our @ISA = qw(Person);

my %fields = (
    id          => undef,
    salary      => undef,
);
```

```
sub new {
    my $class = shift;
    my $self = $class->SUPER::new();
    my($element);
    foreach $element (keys %fields) {
        $self->{_permitted}->{$element} = $fields{$element};
    }
    @{$self}{keys %fields} = values %fields;
    return $self;
}
```

Une fois cela fait, nous n'avons même pas à définir une fonction AUTOLOAD dans le paquetage Employee puisque la version fournie par Person via l'héritage fonctionne très bien.

9 Méta-outils classiques

Même si l'approche par les méthodes proxy est plus pratique pour fabriquer des classes ressemblant à des structures que l'approche fastidieuse nécessitant le codage de chacune des fonctions d'accès, elle laisse encore un peu à désirer. Par exemple, vous devez gérer les appels erronés que vous ne voulez pas capter via votre proxy. Il faut aussi faire attention lors de l'héritage comme nous l'avons montré précédemment.

Les programmeurs Perl ont répondu aux besoins en créant différentes classes de construction de classes. Ces méta-classes sont des classes qui créent d'autres classes. Deux d'entre elles méritent notre attention : Class::Struct et Alias. Celles-ci ainsi que d'autres classes apparentées peuvent être récupérées dans le répertoire modules du CPAN.

9.1 Class::Struct

La plus vieille de toutes est Class::Struct. En fait, sa syntaxe et son interface étaient esquissées alors même que perl5 n'existait pas encore. Elle fournit le moyen de "déclarer" une classe d'objets dont le type de chaque champ est spécifié. La fonction permettant de faire cela s'appelle (rien de surprenant) struct(). Puisque les structures ou les enregistrements ne sont pas des types de base de Perl, à chaque fois que vous voulez créer une classe pour fournir un objet de type structure, vous devez vous-même définir la méthode new() ainsi que les méthodes d'accès aux données pour chacun des champs. Très rapidement, vous trouverez cela enquiquinant (pour ne pas dire autre chose). La fonction Class::Struct::struct() vous soulagera de cette tâche ennuyeuse.

Voici un simple exemple d'utilisation :

```
use Class::Struct qw(struct);
use Jobbie;  # défini par l'utilisateur; voir plus bas

struct 'Fred' => {
    one        => '$',
    many       => '@',
    profession => 'Jobbie',  # n'appelle pas de Jobbie->new()
};

$ob = Fred->new(profession => Jobbie->new());
$ob->one("hmmmm");

$ob->many(0, "here");
$ob->many(1, "you");
$ob->many(2, "go");
print "Just set: ", $ob->many(2), "\n";

$ob->profession->salary(10_000);
```

Les types des champs dans la structure peuvent être déclarés comme des types de base de Perl ou comme des types utilisateurs (classes). Les types utilisateurs seront initialisés par l'appel de la méthode new() de la classe.

Attention au fait que l'objet `Jobbie` n'est pas créé automatiquement par la méthode new() de la class `Fred`. Vous devez donc spécifier un objet `Jobbie` lorsque vous créez un instance de `Fred`.

Voici un exemple réel d'utilisation de la génération de structure. Supposons que vous vouliez modifier le fonctionnement des fonctions Perl gethostbyname() et gethostbyaddr() pour qu'elles renvoient des objets qui fonctionnent comme des structures C. Nous ne voulons pas d'OO, ici. Nous voulons seulement que ces objets se comportent comme des structures au sens C du terme.

```
use Socket;
use Net::hostent;
$h = gethostbyname("perl.com");  # object return
printf "perl.com's real name is %s, address %s\n",
    $h->name, inet_ntoa($h->addr);
```

Voici comment faire en utilisant le module Class::Struct. Le point crucial est cet appel :

```
struct 'Net::hostent' => [          # notez le crochet
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 ];
```

qui crée les méthodes d'objets avec les bons noms et types. Il crée même la méthode new() pour vous.

Vous auriez pu implémenter votre objet de la manière suivante :

```
struct 'Net::hostent' => {          # notez l'accolade
    name       => '$',
    aliases    => '@',
    addrtype   => '$',
    'length'   => '$',
    addr_list  => '@',
 };
```

Class::Struct aurait alors utilisé une table de hachage anonyme comme type d'objet plutôt qu'un tableau anonyme. Le tableau est plus rapide et plus petit, mais la table de hachage fonctionne mieux si éventuellement vous voulez faire de l'héritage. Puisque dans le cas de cet objet de type structure nous ne prévoyons pas d'héritage, nous opterons cette fois-ci plutôt pour la rapidité et le gain de place que pour une meilleure flexibilité.

Voici l'implémentation complète :

```
package Net::hostent;
use strict;

BEGIN {
    use Exporter   ();
    our @EXPORT      = qw(gethostbyname gethostbyaddr gethost);
    our @EXPORT_OK   = qw(
                           $h_name         @h_aliases
                           $h_addrtype     $h_length
                           @h_addr_list    $h_addr
                         );
    our %EXPORT_TAGS = ( FIELDS => [ @EXPORT_OK, @EXPORT ] );
}
our @EXPORT_OK;

# Class::Struct interdit l'usage de @ISA
sub import { goto &Exporter::import }

use Class::Struct qw(struct);
struct 'Net::hostent' => [
   name       => '$',
   aliases    => '@',
   addrtype   => '$',
   'length'   => '$',
   addr_list  => '@',
];
```

```
sub addr { shift->addr_list->[0] }

sub populate (@) {
    return unless @_;
    my $hob = new();  # Class::Struct made this!
    $h_name      =    $hob->[0]              = $_[0];
    @h_aliases   = @{ $hob->[1] } = split ' ', $_[1];
    $h_addrtype =    $hob->[2]              = $_[2];
    $h_length    =    $hob->[3]              = $_[3];
    $h_addr      =                             $_[4];
    @h_addr_list = @{ $hob->[4] } =         @_[ (4 .. $#_) ];
    return $hob;
}

sub gethostbyname ($)  { populate(CORE::gethostbyname(shift)) }

sub gethostbyaddr ($;$) {
    my ($addr, $addrtype);
    $addr = shift;
    require Socket unless @_;
    $addrtype = @_ ? shift : Socket::AF_INET();
    populate(CORE::gethostbyaddr($addr, $addrtype))
}

sub gethost($) {
    if ($_[0] =~ /^\d+(?:\.\d+(?:\.\d+(?:\.\d+)?)?)?$/) {
       require Socket;
       &gethostbyaddr(Socket::inet_aton(shift));
    } else {
       &gethostbyname;
    }
}

1;
```

Outre la création dynamique de classes, nous n'avons qu'effleuré certains concepts comme la redéfinition des fonctions de base, l'import/export de bits, le prototypage de fonctions, les raccourcis d'appels de fonctions via `&whatever` et le remplacement de fonction par `goto &whatever`. Ils ont tous un sens du point de vue des modules traditionnels, mais comme vous avez pu le constater, vous pouvez aussi les utiliser dans un module objet.

Dans la version 5.004 de Perl, vous pouvez trouver d'autres modules objet qui redéfinissent les fonctions de base avec des structures : File::stat, Net::hostent, Net::netent, Net::protoent, Net::servent, Time::gmtime, Time::localtime, User::grent et User::pwent. Le composant final du nom de tous ces modules est entièrement en minuscules qui, par convention, sont réservés aux pragma du compilateur parce qu'ils modifient la compilation et les fonctions internes. Ils proposent en outre les noms de type qu'un programmeur C attend.

9.2 Les données membres comme des variables

Si vous utilisez des objets C++, vous êtes habitué à accéder aux données membres d'un objet par simples variables lorsque vous êtes dans une méthode. Le module Alias offre entre autres cette fonctionnalité, ainsi que la possibilité d'avoir des méthodes privées que les objets peuvent appeler, mais qui ne peuvent l'être depuis l'extérieur de la classe.

Voici un exemple de classe Person créée en utilisant le module Alias. Quand vous mettez à jour ces instances des variables, vous mettez à jour automagiquement les champs correspondants de la table de hachage. Pratique, non ?

```
package Person;

# C'est la même chose qu'avant...
sub new {
     my $class = shift;
     my $self = {
```

```
            NAME  => undef,
            AGE   => undef,
            PEERS => [],
        };
        bless($self, $class);
        return $self;
    }

    use Alias qw(attr);
    our ($NAME $AGE $PEERS);

    sub name {
        my $self = attr shift;
        if (@_) { $NAME = shift; }
        return    $NAME;
    }

    sub age {
        my $self = attr shift;
        if (@_) { $AGE = shift; }
        return    $AGE;
    }

    sub peers {
        my $self = attr shift;
        if (@_) { @PEERS = @_; }
        return    @PEERS;
    }

    sub exclaim {
        my $self = attr shift;
        return sprintf "Hi, I'm %s, age %d, working with %s",
            $NAME, $AGE, join(", ", @PEERS);
    }

    sub happy_birthday {
        my $self = attr shift;
        return ++$AGE;
    }
```

La déclaration `our` est nécessaire parce que le module Alias bidouille les variables globales du paquetage qui ont le même nom que les champs. Pour pouvoir utiliser des variables globales avec `use strict`, vous devez les déclarer au préalable. Ces variables globales du paquetage sont localisées dans le bloc englobant l'appel à attr() exactement comme si vous aviez utilisé local(). Par contre, cela signifie qu'elles sont considérées comme des variables globales avec des valeurs temporaires comme avec tout autre local().

Il serait joli de combiner Alias avec quelque chose comme Class::Struct ou Class::MethodMaker.

10 NOTES

10.1 Terminologie objet

Dans la littérature OO, un grand nombre de mots différents sont utilisés pour nommer quelques concepts. Si vous n'êtes pas déjà un programmeur objet vous n'avez pas à vous en préoccuper. Dans le cas contraire, vous aimeriez sûrement savoir à quoi correspondent ces concepts en Perl.

Par exemple, un objet est souvent appelé une *instance* d'une classe et les méthodes objet *méthodes d'instance*. Les champs spécifiques de chaque objet sont souvent appelés *données d'instance* ou *attributs d'objet*. Tandis que les champs communs à tous les membres de la classe sont nommés *donnée de classe*, *attributs de classe* ou <données membres statiques>.

Classe de base, *classe générique* et *super classe* recouvrent tous la même notion. *Classe dérivée*, *classe spécifique* et *sous-classe* décrivent aussi la même chose.

Les programmeurs C++ ont des *méthodes statiques* et des *méthodes virtuelles*, Perl ne propose que les *méthodes de classe* et les *méthodes d'objet*. En fait, Perl n'a que des méthodes. Qu'une méthode s'applique à une classe ou à un objet ne se fait qu'à l'usage. Vous pouvez accidentellement appeler une méthode de classe (une méthode qui attend une chaîne comme argument implicite) comme une méthode d'objet (une méthode qui attend une référence comme argument implicite) ou vice-versa.

Du point de vue C++, toutes les méthodes en Perl sont virtuelles. C'est la raison pour laquelle il n'y a jamais de vérification des arguments par rapport aux prototypes des fonctions comme cela peut se faire pour les fonctions prédéfinies ou définies par l'utilisateur.

Puisque une classe... (NDT: je n'ai pas compris ce paragraphe !) Because a class is itself something of an object, Perl's classes can be taken as describing both a "class as meta-object" (also called *object factory*) philosophy and the "class as type definition" (*declaring* behaviour, not *defining* mechanism) idea. C++ supports the latter notion, but not the former.

11 VOIR AUSSI

Vous trouverez sans aucun doute de plus amples informations en lisant les documentations suivantes : *perlmod*, *perlref*, *perlobj*, *perlbot*, *perltie* et *overload*.

perlboot est un tutoriel simple et facile pour la programmation orientée objet.

perltooc donne plus de détails sur les données de classe.

Quelques modules particulièrement intéressants sont Class::Accessor, Class::Class, Class::Contract, Class::Data::Inheritable, Class::MethodMaker et Tie::SecureHash.

12 AUTEURS ET COPYRIGHT

12.1 Remerciements

Merci à Larry Wall, Roderick Schertler, Gurusamy Sarathy, Dean Roehrich, Raphael Manfredi, Brent Halsey, Greg Bacon, Brad Appleton et à beaucoup d'autres pour leurs remarques pertinentes.

13 TRADUCTION

La traduction française est distribuée avec les même droits que sa version originale (voir ci-dessus).

13.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

13.2 Traducteur

Tradiction et mise à jour : Paul Gaborit (Paul.Gaborit at enstimac.fr).

13.3 Relecture

Philippe de Visme (*philippe@devisme.com*).

14 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.