Journées Perl 2006

V1.0

Vous avez dit « Kalité » ?

- ***Tentative* de définition**
  - La « Kalité » est une *approximation* de « Qualité »
  - Nul ne sait ce que c'est vraiment...
  - Mais on *pense* la reconnaître quand on la voit !
- **C'est avant tout une question de *confiance***
  - Construite grâce à la réussite de tests (mais ce n'est pas tout)
  - Mais absence de bugs (trouvés) n'implique pas Kalité !
  - Éventuellement si la couverture *fonctionnelle* des tests est *décente*
- **La chasse aux *bugs* reste cependant ouverte !**
  - Un bug est une différence entre test, documentation & code
  - Si la documentation est ambiguë, c'est bel et bien un bug !
  - L'idée, c'est de *progresser* en en trouvant de moins en moins

Avertissement !

Quand & Quoi [1]

- **Complètement avant**
  - Littérature
  - CPAN
  - Articles, conférences, /Perl Mon(k|geur)s/, etc.
- **Bien avant**
  - *Générer* le squelette du module
- **Avant**
  - Écrire les tests (un soupçon d'XP dans votre code)
- **Pendant *(la phase d'écriture de code)***
  - Documenter au fur et à mesure (et pourquoi pas, avant ?)
  - Ajouter des tests si nécessaire

Quand & Quoi [2]

- **Après *(entre codage & livraison)***
  - Tester (suite de tests – acceptation *et* non-régression)
  - Mesurer la couverture de POD
  - Mesurer la couverture de code des tests
  - Mesurer la couverture fonctionnelle des tests (Ah ! Ah !)
  - Générer des synthèses lisibles
- **Bien après *(la livraison)***
  - Suite à un rapport de bug...
  - Ajouter des tests
  - Reproduire le problème
  - Éradiquer le(s) bug(s)
  - Tester de nouveau (suite de tests – non-régression)

Les pré-requis [1]

- **SCM – gestion de code source (~ historique + //)**
  - Par exemple : cvs, subversion, svk, darcs, etc.
- **RT – gestion de tickets (~ intention)**
  - Par exemple : cvstrac, trac, RT, bugzilla, etc.
- **Éditeur avec « syntax highlighting »**
  - Par exemple : NEdit, vi, emacs, etc.
- **Des règles de codage consistantes**
  - Voire même les « bonnes » ;)
  - Cf PBP (livre) + Perl::Critic (module) + perltidy (outil)
- **On peut même utiliser un IDE**
  - Comme Eclipse + plugin Perl (mais pas trop envie d'essayer ;)

Les pré-requis [2]

- **On ne choisit pas forcément...**
  - SCM, RT, « bonnes » pratiques ou même l'éditeur de texte :(
  - Fonction de l'OS
  - Fonction des choix « corporate »
  - Fonction du client
  - Faute d'avoir ce que l'on aime, il faut aimer ce que l'on a !
- **Mais on *choisit* d'utiliser les directives « use »**
  - ```
    use strict;
    use warnings;
    ```
  - C'est même très fortement conseillé !
- **Sinon : « *Il y en a qui ont essayé...* »**
    ```

Les pré-requis [3]

- « ***Ils ont eu des problèmes !*** »

HINDENBURG BLAST KILLS 35

At Exact Moment That Zeppelin Blew Up

Experts Suspect Failure to 'use strict'

Giant of Sky Blows Up in Flames at Lakehurst Mooring Mast: 62 Survivors Include Capt. Lehmann and Skipper; Spectators Bowled Over; Heroes Dive Into Inferno

By GEORGE M. MAWHINNEY

Doomed Air Queen Caught by Camera as First Explosion Ripped Hull

Ne pas réinventer la roue [1]

- **Éviter de commettre les erreurs des autres**
  - Difficile de résister au syndrome NIH (« Not Invented Here »)
  - Moins d'orgueil, plus de paresse !
  - En plus, c'est ultra tendance (« knowledge management »)
- **Chercher plutôt à utiliser un module du CPAN**
  - « *Je code en CPAN, le reste c'est de la syntaxe* » – Audrey Tang
- **Mais faire au préalable une revue de module**
  - Utilité dans la pratique
  - Possibilités de configuration
  - Module OO classe de base à sous-classer
  - Développement actif du module
  - Etc.

Ne pas réinventer la roue [2]

- **Quelques tâches parmi tant d'autres...**
  - Des fois même pénibles à coder !
- **Analyser la ligne de commande**
  - Getopt::Long (un grand classique)
  - Getopt::Euclid (le POD sert de spécification, excellent !)
- **Gérer des configurations**
  - Config::Std (~ M$ INI)
  - YAML
  - Et non, pas de XML (« *même pas dans tes rêves les plus fous* ») !
- **Pèle-mêle... (cf Phalanx Top 100)**
  - HTML::Parser, XML::Twig, Spreadsheet::ParseExcel, Parse::RecDescent, RegExp::Common, List::MoreUtils, etc.

Ne pas réinventer la roue [3]

- **Littérature**
  - Perl en action (Perl cookbook – Christiansen & Torkington)
  - De l'art de programmer en Perl (Perl best practices – Conway)
  - Mastering algorithms with Perl (Orwant, Hietaniemi & MacDonald)
  - Perl testing: a developer's handbook (Ian Langworth & Chromatic)
  - The pragmatic programmer (Hunt & Thomas)
  - Lessons learned in software testing (Kaner, Bach & Pettichord)
- **Expériences**
  - Groupes (Mongueurs de Perl, Perl Mongers, Perl Monks)
  - Conférences (Journées Perl, Perl Workshops, YAPC)
  - Articles (Mongueurs de Perl / GLMF, perl.com)

Au commencement...

- **Bien construire son propre module**
  - Un module Perl est une arborescence très précise
  - Facile d'oublier l'un de ses nombreux fichiers
  - Difficile de se souvenir de la syntaxe de chacun d'entre-eux
- **Utiliser un module dédié du CPAN**
  - Par exemple : Module::Starter (voire même Module::Starter::PBP)
  - Crée des « gabarits » à compléter
  - Certains tests vérifieront qu'ils ont bien été modifiés
- **Cf l'article de Sébastien Aperghis-Tramoni**
  - « *Créer une distribution pour le CPAN* », GLMF #69
  - http://articles.mongueurs.net/magazines/linuxmag69.html

Le test pour les nuls [1]

- **Tester = confronter *intention* & *implémentation***
  - À l'aide de techniques (tests directifs ou aléatoires contraints)
  - Et d'un modèle de référence (OK ~ pas de ≠ avec ce dernier)
- **Intention**
  - Cristallisée dans une spécification, un plan de test, etc.
  - Quand lesdits documents sont bel et bien disponibles !
  - Documents *non-formels* car destinés à une lecture humaine
  - Donc bien évidemment sujets à *interprétation*
- **Implémentation**
  - Code
  - Décomposé en unités de base (modules = ∑ fonctions)

Le test pour les nuls [2]

- **Développement piloté par les tests (TDD)**
  - Tests unitaires (au standard xUnit ou non)
  - Tests d'acceptation (ou de recette – ce pour quoi le client a payé)
- **Suite de tests ≈ spécification exécutable**
  - C'est déjà un peu plus formel (ou moins informel ;)
  - « *Les vieux tests ne meurent jamais, ils deviennent des tests de non-régression !* » – Chromatic & Michael G Schwern
- **Mais un test réussi ne révèle pas grand chose !**
  - Ça doit même devenir frustrant pour le testeur !
- **Juste histoire d'enfoncer le clou...**
  - « *Tester un programme démontre la présence de bugs, pas leur absence.* » – Edsger Dijkstra

Le test pour les nuls [3]

- **Un ingénieur de test doit se poser 2 questions**
  - Est-ce correct ?
  - Ai-je fini ?
- **Est-ce correct ?**
  - Les tests de la suite sont-ils 100% OK ?
  - Avec les notions de SKIP/TODO, c'est plutôt binaire
  - C'est le rôle du protocole TAP et de Test::More + Test::Harness
- **Ai-je fini ?**
  - Mes tests ont-ils exercé toutes mes lignes de code ?
  - Notion de couverture de code (associée à une métrique)
  - C'est le domaine du module Devel::Cover
  - Mais... mes lignes de code implémentent-elles la fonctionnalité ?

Le test pour les nuls [4]

- **Couverture de code ≠ couverture fonctionnelle**
  - C'est en effet très tentant de confondre les deux
- **Couverture de code**
  - On peut couvrir le code à 100% mais...
  - Il peut manquer en fait celui réalisant la fonctionnalité demandée !
- **Couverture fonctionnelle**
  - Meilleure définition du « *ai-je fini ?* » en ces termes
  - Liée aux combinaisons possibles en entrée d'une fonction (cf CRT)
- **M'enfin ! Comment mesurer la CF en Perl ?**
  - C'est possible avec des HDVL comme SystemVerilog
  - C'est dans les TODO du module Test::LectroTest

Le test pour les nuls [5]

- **Couverture de code ≠ couverture fonctionnelle**
  - Le trivial contre-exemple suivant
  - ```
    =head2 foo

      Retourne 'foo' à 'bar' et 'gabuzomeu' à 'baz'. Retourne undef si entrée inconnue.

    =cut

    sub foo {
      my $s = shift;

      return 'gabuzomeu' if $s eq 'baz';

      undef;
    }

    use Test::More tests => 2;

    is ( foo( 'baz' ), 'gabuzomeu', "retourne 'gabuzomeu' à baz"        );
    is ( foo( 'foo' ), undef,       "retourne undef si entrée inconnue" );
    ```
  - Atteint 100% de CC... mais n'implémente pas foo ('bar') = 'foo' !
  - ```
    ---------------------------- ------ ------ ------ ------ ------ ------ ------
    File                          stmt   bran   cond    sub    pod   time  total
    ---------------------------- ------ ------ ------ ------ ------ ------ ------
    t_foo.t                      100.0  100.0    n/a  100.0    n/a  100.0  100.0
    Total                        100.0  100.0    n/a  100.0    n/a  100.0  100.0
    ---------------------------- ------ ------ ------ ------ ------ ------ ------
    ```

Tests unitaires

```
use Test::More;

plan(tests=>N);

use_ok('Foo');


# ...

is_deeply(
  bar    ($baz),
  refbar ($baz),
  'baz au bar'
);
# ...
```

stimulus

t/13-bar.t

```
use strict;
use warnings;
package Foo;
use Carp::Assert::More;

# ...

=head2 bar
  bar ( baz )

Rince baz.
=cut

sub bar {
  my $baz = shift;
  assert_nonblank $baz;
  # ...
}

# ...
```

lib/Foo.pm

Test::More
is(), is_deeply(), ...

≈

TAP:
OK, !OK
SKIP /
TODO

f()

modèle de référence

Le protocole TAP [1]

- **« Test Anything Protocol »**
  - Séparation des tests de l'interpréteur de résultats
  - Développé par des perlistes mais en fait indépendant du langage
- **La fonctionnalité à tester est une boîte noire**
  - Le programme de test doit juste sortir un flux TAP
  - A l'aide d'une boîte à outils
  - Par exemple le module Test::More (`plan()`, `use_ok()`, `is()`, etc.)
- **Le nombre de tests est annoncé à l'avance**
  - Notion de « plan » de test
  - Le test est déclaré faux si M tests reportés ≠ N tests annoncés
  - Car un test peut par exemple tout planter avant la fin des tests

Le protocole TAP [2]

- **Le plan de test**
  - `1..N (todo X Y)?`
- **Les tests**
  - `ok X      - description`
  - `not ok X - description`
  - `ok Y      # SKIP raison`
  - `not ok Y # TODO raison`
- **SKIP**
  - Éluder le test à cause d'un facteur externe (module ∃!, OS, etc.)
- **TODO**
  - Fonctionnalité pas encore implémentée (peut néanmoins être OK)

Le protocole TAP [3]

- **Cf article GLMF #88**
  - « *Les tests en Perl - Présentation et modules standards* » – Sébastien Aperghis-Tramoni & Philippe Blayo
- **Quelques interpréteurs TAP**
  - Test::Harness
  - Test::TAP::Model (MI bâti sur le flux TAP -> Test::TAP::HTMLMatrix)
- **Pour en savoir plus...**
  - La spécification est en fait le module TAP
  - Article sur Wikipedia : Test_Anything_Protocol
  - Présentation de Sébastien aux Journées Perl (demain !)

Test::Harness

- **make test**
  - % **make** test
    PERL_DL_NONLAZY=1 /usr/local/bin/perl "-MExtUtils::Command::MM" \
      "-e" "test_harness(0, 'blib/lib', 'blib/arch')" t/*.t

    t/00-load.....ok 2/6# Testing SOCK v1.0.2, \
    Perl 5.008007, /usr/local/bin/perl

    t/00-load.....ok

    Traçabilité

    Tests fonctionnels

    t/01-rip_fmxml.....ok
    t/02-rip_fmxml_again.....ok
    t/03-rip_register_bit_fields.....ok
    t/04-parse_fmxml_datasheet.....ok
    t/05-rip_fmxml_table.....ok
    t/06-evaluate.....ok
    t/07-scavenge_full_description....ok
    t/08-spirit_version.....ok
    t/09-frontend_tools.....ok

    Tests POD

    t/boilerplate.....ok
    t/pod-coverage.....ok
    t/pod.....ok

    Synthèse

    All tests successful.
    Files=13, Tests=141, 40 wallclock secs (20.52 cusr + 1.12 csys = 21.64 CPU)

Matrice TAP [1]

- **Offre une vue synthétique de la suite de tests**
  - Très appréciable dès lors que le nombre de tests augmente
- **À l'aide d'un interpréteur dédié**
  - Le module Test::TAP::Model::Visual
  - L'interpréteur analyse le flux TAP et construit un MI TTM
  - Le MI TTM est transformé en HTML (Test::TAP::HTMLMatrix)
- **Très simplement**

```
use Test::TAP::Model::Visual;
use Test::TAP::HTMLMatrix;

$ttm = Test::TAP::Model::Visual->new_with_tests( <t/*.t> );
$v   = Test::TAP::HTMLMatrix->new( $ttm );

open FH, "> matrice.html";
print FH $v->html;
```

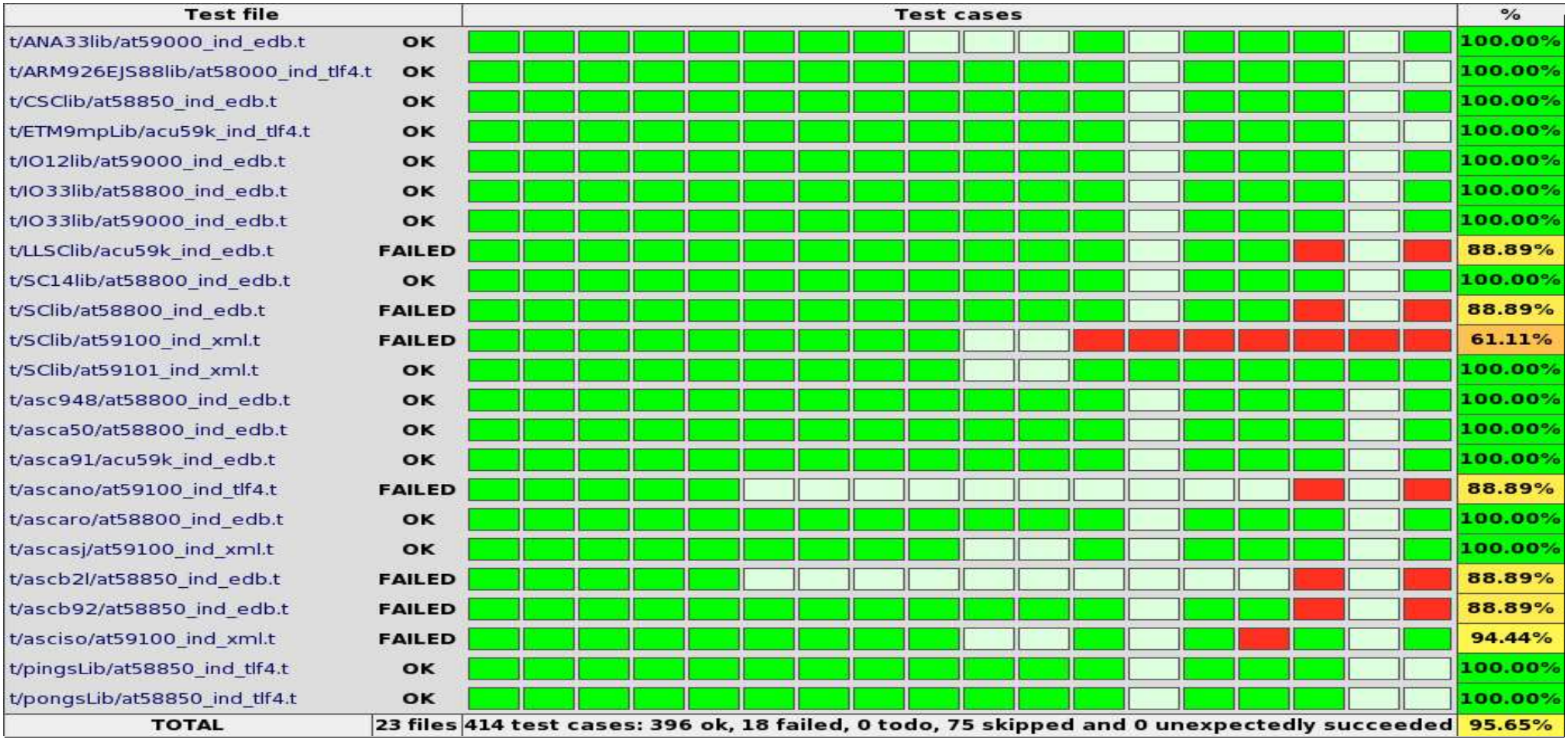
Matrice TAP [2]

- **Notre `make test` de tout à l'heure**

Test file		Test cases	%
t/00-load.t	OK		100.00%
t/01-rip_fmxml.t	OK		100.00%
t/02-rip_fmxml_again.t	OK		100.00%
t/03-rip_register_bit_fields.t	OK		100.00%
t/04-parse_fmxml_datasheet.t	OK		100.00%
t/05-rip_fmxml_table.t	OK		100.00%
t/06-evaluate.t	OK		100.00%
t/07-scavenge_full_description.t	OK		100.00%
t/08-spirit_version.t	OK		100.00%
t/09-frontend_tools.t	OK		100.00%
t/boilerplate.t	OK		100.00%
t/pod-coverage.t	OK		100.00%
t/pod.t	OK		100.00%
TOTAL	**13 files**	**141 test cases: 141 ok, 0 failed, 0 todo, 0 skipped and 0 unexpectedly succeeded**	**100.00%**

Matrice TAP [3]

- **Autre exemple : transformation de données**



Test file		Test cases	%
t/ANA33lib/at59000_ind_edb.t	OK		100.00%
t/ARM926EJS88lib/at58000_ind_tlf4.t	OK		100.00%
t/CSClib/at58850_ind_edb.t	OK		100.00%
t/ETM9mpLib/acu59k_ind_tlf4.t	OK		100.00%
t/IO12lib/at59000_ind_edb.t	OK		100.00%
t/IO33lib/at58800_ind_edb.t	OK		100.00%
t/IO33lib/at59000_ind_edb.t	OK		100.00%
t/LLSClib/acu59k_ind_edb.t	FAILED		88.89%
t/SC14lib/at58800_ind_edb.t	OK		100.00%
t/SClib/at58800_ind_edb.t	FAILED		88.89%
t/SClib/at59100_ind_xml.t	FAILED		61.11%
t/SClib/at59101_ind_xml.t	OK		100.00%
t/asc948/at58800_ind_edb.t	OK		100.00%
t/asca50/at58800_ind_edb.t	OK		100.00%
t/asca91/acu59k_ind_edb.t	OK		100.00%
t/ascano/at59100_ind_tlf4.t	FAILED		88.89%
t/ascaro/at58800_ind_edb.t	OK		100.00%
t/ascasj/at59100_ind_xml.t	OK		100.00%
t/ascb2l/at58850_ind_edb.t	FAILED		88.89%
t/ascb92/at58850_ind_edb.t	FAILED		88.89%
t/asciso/at59100_ind_xml.t	FAILED		94.44%
t/pingsLib/at58850_ind_tlf4.t	OK		100.00%
t/pongsLib/at58850_ind_tlf4.t	OK		100.00%
TOTAL	23 files	414 test cases: 396 ok, 18 failed, 0 todo, 75 skipped and 0 unexpectedly succeeded	95.65%

Couverture de code [1]

- **Charger le module Devel::Cover lors du test**
  - ```
    % cover -delete
    % HARNESS_PERL_SWITCHES=-MDevel::Cover make test
    % cover -report html
    ```

Coverage Summary

Database: /data/141/users/xavier/dev/SOCK/SOCK/cover_db

file	stmt	bran	cond	sub	pod	time	total
blib/lib/SOCK.pm	100.0	n/a	n/a	100.0	n/a	0.0	100.0
blib/lib/SOCK/BOT.pm	87.0	81.9	68.4	90.6	100.0	31.3	87.0
blib/lib/SOCK/DOM.pm	100.0	70.8	100.0	100.0	100.0	42.4	96.0
blib/lib/SOCK/DOM/Component.pm	94.8	81.8	64.7	100.0	100.0	6.1	91.8
blib/lib/SOCK/MJD.pm	100.0	n/a	n/a	100.0	100.0	0.1	100.0
blib/lib/SOCK/NRT.pm	100.0	100.0	n/a	100.0	100.0	20.1	100.0
Total	93.2	81.0	68.4	96.1	100.0	100.0	91.3
    ```

Couverture de code 2

- **Statements**
  - Toutes les instructions ont-elles été exécutées ?
- **Branches**
  - Vérifie les alternatives des branchements conditionnels (`if, ?:`)
- **Conditions**
  - Vérifie les différentes possibilités dans les expressions logiques
- **Subroutines**
  - Toutes les fonctions ont-elles été appelées ?
- **POD**
  - Utilisation du module POD::Coverage

Documentation

- **Du code testé et couvert, c'est déjà pas mal**
- **Du code documenté, c'est mieux !**
- **Documentation écrite en POD (« Plain Old Doc »)**
  - Il faut en vérifier la syntaxe à l'aide du module Test::POD
  - C'est le rôle du test `t/pod.t` (créé par Module::Starter)
- **Couverture de POD**
  - Tâche assumée par le module Test::POD::Coverage
  - Vérifie que toute fonction possède une documentation associée
  - En pratique, vérifie
    - `=item foo ... =cut`
    - `=head foo ... =cut`
  - C'est le rôle du test `t/pod-coverage.t` (créé par Module::Starter)

Kwalitee

- **Pour une définition plus exhaustive : CPANTS**
  - « CPAN Testing Service » – http://cpants.perl.org/kwalitee.html
  - Définit des indicateurs de Kalité (« ALPHA – Hic sunt dracones! »)
- **Obtenir les indicateurs : le module Test::Kwalitee**
  - Ajouter le test `t/kwalitee.t` à la suite de tests :
  - ```
    eval { require Test::Kwalitee };
    exit if $@;
    Test::Kwalitee->import;
    ```
  - ```
    ok  1 - extractable
    ok  2 - has_readme
    ok  3 - has_manifest
    ok  4 - has_meta_yml
    ok  5 - has_buildtool
    ok  6 - has_changelog
    ok  7 - no_symlinks
    ok  8 - has_tests
    ok  9 - proper_libs
    ok 10 - no_pod_errors
    ok 11 - use_strict
    ok 12 – has_test_pod
    ok 13 - has_test_pod_coverage
    ```

Assertions

- **Décrivent les hypothèses de travail d'une fonction**
  - Ses limites, ce qu'elle ne sait pas faire « du tout du tout » !
  - Il vaut mieux planter le programme dès que possible
  - Plutôt que de le laisser s'embarquer vers l'imprévisible
  - Un plantage à cause d'une assertion est plus facile à résoudre
  - « *Les programmes morts ne racontent pas de craques !* »
    – Hunt & Thomas in The Pragmatic Programmer
- **Assertions du module Carp::Assert::More**
  - Simples.....assert_ + (is, isnt, like, defined, nonblank)
  - Numériques..... assert_ + (integer, nonzero, positive, ...)
  - Références..assert_ + (isa, nonempty, nonref, hashref, listref)
  - Array/hash.....assert_ + (in, exists, lacks)

Test::LectroTest [1]

- **Les tests traditionnels sont dits « dirigés »**
  - Séquences de stimuli et de comparaisons à des valeurs attendues
  - Mais on ne pense pas forcément à tout (# de combinaisons)
- **Une alternative : des tests aléatoires contraints**
  - Ou en Anglais, CRT : « Constrained Random Testing »
  - On laisse la machine faire le sale boulot, (pseudo-)aléatoirement
- **La recette avec Test::LectroTest**
  - Associer un type à chacun des paramètres (des *types* en Perl ?)
  - Ajouter des contraintes aux paramètres (~ sous-ensembles)
  - Faire N itérations, mesurer la CF, bidouiller les contraintes, goto 0
- **Pas encore utilisé en production mais ça vient !**
  - Son alter-ego dans le monde matériel l'est (codé en SystemVerilog)

Test::LectroTest [2]

Le code suivant

```
use Test::LectroTest::Compat; # ::Compat permet d'interfacer Test::More
use Test::More tests => 1;

sub ref_foo {
  { bar => 'foo', baz => 'gabuzomeu' }->{shift()};
}

my $property = Property {

  ##[ s <- Elements( 'foo', 'bar', 'baz' ) ]##

  is( foo( $s ), ref_foo( $s ), 'foo / ref_foo' );

}, name => 'toutes les entrées possibles de foo' ;

holds( $property, trials => 100 ); # vérifie que la propriété "tient" sur 100 entrées aléatoires
```

Modèle de référence

Contraintes sur les entrées

Comparaison à la référence

Prouve que foo ('bar') ≠ 'foo'

```
#   Failed test 'property 'toutes les entrées possibles de foo' falsified in 4 attempts'
#   in lt_foo.t at line 30.
#         got: undef
#    expected: 'foo'
#    Counterexample:
#    $s = "bar";
```

CF ≈ statistiques sur les ≠ valeurs des entrées

En résumé...

- **A priori**
  - Lire, apprendre et ne pas hésiter à poser des questions
  - Utiliser des outils éprouvés (SCM, RT, éditeurs, etc.)
  - Avoir de « bonnes » pratiques
  - Ne pas réinventer la roue
  - Écrire les tests (voire même la documentation) en premier
- **A posteriori**
  - Utiliser le protocole de test TAP et ses interpréteurs associés
  - Analyser les taux de couverture de code (≠ CF !) et de POD
  - Insérer des assertions dans le code
  - Voire même laisser la machine faire le sale boulot à votre place !
  - Générer des rapports de test synthétiques et lisibles

Ingénierie sociale

- **Comme dans beaucoup d'activités humaines**
  - Il y a la technique **et** il y a l'engagement
- **La technique**
  - On vient d'en parler pendant 40 (longues) minutes
  - C'était loin d'être exhaustif !
- **L'engagement**
  - Sans la *motivation*, point de Kalité !
  - C'est un *objectif* (l'augmenter) et non pas un *but* (l'atteindre)
- **Une dernière citation pour conclure**
  - « *À cette époque (1909), l'ingénieur en chef était aussi presque toujours le pilote d'essai. Cela eu comme conséquence heureuse d'éliminer très rapidement les mauvais ingénieurs dans l'aviation.* »
    – Igor Sikorsky

Questions ?

Sur la toile...

- **Les hoplites de la Phalange veulent en découdre...**
  - Kwalitee : http://qa.perl.org/phalanx/kwalitee.html
- **Modules CPAN**
  - Module::Starter, Module::Starter::PBP
  - Carp::Assert, Carp::Assert::More
  - Devel::Cover
  - Test::More, Test::Harness
  - Test::POD, Test::POD-Coverage
  - Test::TAP::Model, Test::TAP::HTMLMatrix
  - Test::LectroTest
- **Talks**
  - Test::Tutorial, Devel::Cover & Test::LectroTest