YAPC Europe 2007

v2.0
(translated, edited & augmented from French Perl Workshop 2006 talk)

“Kwalitee”?

- **Definition *attempt***
  - “Kwalitee” is an *approximation* of “Quality”
  - Nobody knows what it *is* actually...
  - Anyway, we *believe* we're able to recognize it when we see it!
- **It's mainly a matter of *confidence***
  - Built through passed tests (but it's not enough as we'll see later)
  - Yet, absence of bugs (read “unfound”) does not imply Kwalitee!
  - Although a correlation exists if tests *functional* coverage is *decent*
- ***“Go ahead bug, make my day!”***
  - A bug is a difference between expectation and implementation
  - It is also a difference between test, documentation & code
  - If documentation is missing, this is a bug indeed!

Achtung!

** Truth is NOT out there!

* Truth is, there is no truth.
(including this one)**

When & What [1]

- **Ages before**
  - Literature
  - CPAN
  - Articles, conferences, /Perl Mon(k|ger)s/, etc.
  - "*Read. Learn. Evolve.*" – Klortho the Magnificent
- **Before**
  - *Generate* module skeleton
  - Use an OO class builder (if applicable)
  - Write tests (a tad of XP in your code)
- **While *(coding)***
  - Document in parallel (and why not, before?)
  - Add more tests if required

When & What [2]

- **After *(between coding & release)***
  - Test (test suite – acceptance *and* non-regression)
    - Measure POD coverage
    - Measure tests code coverage
    - Measure tests functional coverage (Ha! Ha!)
  - Generate synthetic reports
    - For one-glance checking purposes or for traceability's sake
- **Way after *(release)***
  - Refactor early, refactor often
    - Test suite (non-regression) *should* ensure nothing got broken in the process
  - Following a bug report...
    - First add test(s) to reproduce the code defect(s)
    - Then – and only then – nuke bug(s) out
    - Test again (test suite – non-regression)

Hey! Hold on! Social Perl?

- **What's *social* about this ramble anyway?**
  - *"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."* – Damian Conway

- **From SICP's preface**
  - *"Thus, programs must be written for people to read, and only incidentally for machines to execute."*

Pre-requesites [1]

- **SCM – source code management (~ history + //)**
  - For example: cvs, subversion, svk, darcs, git, etc.
  - Beware! Needs some kind of *etiquette* (tagging, branching, etc.)
  - Use a good “difference engine” (GNU diff), possibly w/ GUI (tkdiff)
- **RT – request tracker (~ intention)**
  - For example: cvstrac, trac, RT, bugzilla, etc.
- **Text editor with syntax highlighting**
  - For example: NEdit, vi, emacs, etc.
- **Consistent coding “rules” (OK, “best practices”)**
  - It might even be the “good” ones ;^)
  - Cf PBP (book) + Perl::Critic (module) + perltidy (tool)

Pre-requesites [2]

- **An IDE might also be of some help**
  - Like Eclipse + Perl plugin (but I'm not too eager to try ;^)
- **Indeed, we may not be allowed to choose...**
  - SCM, RT, “good” practices or even the text editor :^(
  - Due to OS, “corporate” practices, customer, etc.
  - If you don't have what you like, you should like what you have!
- **But we *choose* to use compiler directives**
  - ```
    use strict;   # for code
    use warnings; # for test
    ```
  - It is even *strongly* advised!
- **Else: “*Some people have tried...*”**
    ```

Pre-requesites [3]

- ***"They had some problems!"***

VOL. 236, NO. 129 PHILADELPHIA, FRIDAY MORNING, MAY 7, 1937 TWO CENTS

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

LAKEHURST, May 6.—The giant German Zeppelin Hindenburg exploded and plunged to flaming doom on the Naval air station field here at 7.23 o'clock tonight, carrying 35 to 50 men and women passengers and members of her crew to instant, flame-seared death while, with several hundred others, I looked on, deafened and half stunned.

Doomed Air Queen Caught by Camera as First Explosion Ripped Hull

Do not reinvent the wheel [1]

- **Avoid repeating others' errors**
  - Hard to escape from *NIH* syndrome (“*Not Invented Here*”), isn't it?
  - Less hubris, more laziness!
- **Consider using a CPAN module instead**
  - *“I code in CPAN, the rest is syntax.”* – Audrey Tang
- **But first do a module review**
  - Practical utility
  - Configurability
  - Active development (if applicable – might have reached stability)
- **Anyway, if you still want to reinvent the wheel...**
  - At least, try to reinvent a better one!

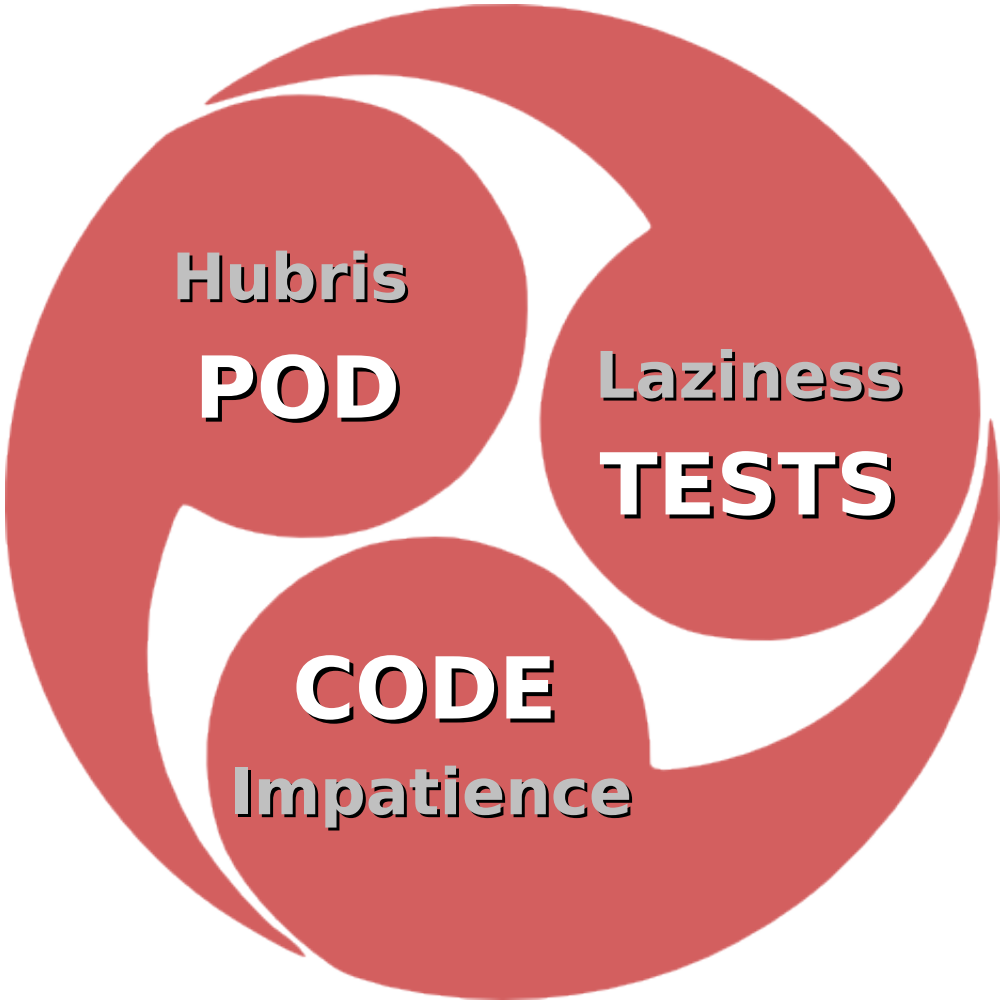
Do not reinvent the wheel [2]

- **Some not-so-uncommon tasks...**
  - Sometimes even a bore to code!
- **Command line parsing**
  - Getopt::Long (an all-time classic)
  - Getopt::Euclid (POD is used to describe switches)
- **Configuration handling**
  - Config::Std (~ M$ INI)
  - YAML
  - And no way, no XML (“*not even in your wildest dreams*”)!
- **Off the top of my head... (cf Phalanx Top 100)**
  - HTML::Parser, XML::Twig, Spreadsheet::ParseExcel, Parse::RecDescent, RegExp::Common, List::MoreUtils, etc.

Do not reinvent the wheel [3]

- **Literature**
  - Perl cookbook (Christiansen & Torkington)
  - 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)
  - Refactoring: Improving the Design of Existing Code (Fowler et al.)
- **Experiences**
  - User groups (Perl Mongers, Perl Monks)
  - Conferences (Perl Workshops, YAPC)
  - Articles (Perl Journal, perl.com)

Programmer's triptych



At the beginning...

- **Build correctly your own module in the 1st place**
  - A Perl module is a precise file tree structure
  - Easy to forget one of its numerous files (see José's guide)
  - Hard to remember the syntax of every file
- **Use a dedicated CPAN module**
  - For example Module::Starter (or even Module::Starter::PBP)
  - Creates correct-by-construction templates to fill out
  - Dedicated tests will check if they have been tampered with
- **Use an OO class builder (if applicable)**
  - Like Class::Generate, Class::MethodMaker, Class::Accessor, etc.
  - Or even Class::Std for an inside-out implementation

Testing for dummies [1]

- **Test = confront *intention* & *implementation***
  - Using *techniques* (“directed” or “constrained random” tests)
  - And a *reference* model (OK ~ no ≠ vs reference)
- **Intention**
  - Written in a specification, a test plan, etc.
  - When these documents are available indeed!
  - *Not-so-formal* stuff for they are meant to be read by humans
  - So obviously prone to *interpretation*
- **Implementation**
  - Code (+ documentation)
  - Split by units (modules = ∑ functions)

Testing for dummies [2]

- **Test-driven development (TDD)**
  - Unit tests (might be xUnit-compliant or not)
  - Acceptance tests (i.e., what the customer has paid for)
- **Test suite ≈ executable specification**
  - Somehow more formal (or somehow less informal ;^)
  - *"Old tests don't die, they just become non-regression tests!"*
    – chromatic & Michael G Schwern
- **But a passed test does not mean a lot!**
  - It should even be frustrating ("*OK? So what?*") for a tester!
- **To put it (again) the blunt way...**
  - *"Program testing can be used very efficiently to show the presence of bugs, but never to show their absence."*
    – Edsger Dijkstra (EWD288)

Testing for dummies [3]

- **Tester should ask 2 fundamental questions**
  - *"Is this correct?"*
  - *"Am I finished?"*
- ***"Is this correct?"***
  - Are *all* suite tests *100%* OK?
  - TAP protocol's role via Test::More + Test::Harness modules
  - With SKIP/TODO TAP concepts, it's a closed-answer Q (i.e., 100%)
- ***"Am I finished?"***
  - Did my tests actually stressed all my lines of code?
  - Code coverage concept (associated with metrics)
  - Falls into Devel::Cover module's domain
  - But... do my code lines *really* implement *meant* functionality?

Testing for dummies [4]

- **Code coverage ≠ functional coverage**
  - It's actually *very* tempting to pretend latter is equivalent to former
- **Code coverage**
  - A given code might be 100% covered, yet...
  - It could miss the part that does implement meant functionality!
- **Functional coverage**
  - A better answer to the "*am I done?*" question
  - Linked to all possible input combinations of a function (cf CRT)
- **Damned! How do I measure FC in Perl?**
  - It is possible with recent HDVL like SystemVerilog
  - It is listed in TODOes of Test::LectroTest module

Testing for dummies [5]

- **Code coverage ≠ functional coverage**
  - The trivial following counter-example
  -
```
=head2 foo

  Returns 'foo' to 'bar' and 'gabuzomeu' to 'baz'. Returns undef else.

=cut

sub foo {
  my $s = shift;

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

  undef;
}

use Test::More tests => 2;

is ( foo( 'baz' ), 'gabuzomeu', "returns 'gabuzomeu' if 'baz'"   );
is ( foo( 'foo' ), undef,       "returns undef if unknown input" );
```
  - Reaches 100% CC... but does not implement 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
---------------------------- ------ ------ ------ ------ ------ ------ ------
```

Unit tests

```
use warnings;

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;

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

$f()$

reference model

TAP protocol [1]

- **“Test Anything Protocol”**
  - Separation between result flow and results interpreter
  - Designed by Perl folks but actually language-agnostic
- **The function to test is seen as a black-box**
  - Test program only has to supply the interpreter with a TAP flow
  - Using ad'hoc toolbox (a module of course)
  - For example, Test::More module (`plan()`, `use_ok()`, `is()`, etc.)
- **Number of tests to perform is declared first**
  - Test “plan” (a bit shallow IMHO vs – overkill? – IEEE Std 829)
  - Test is failed if M *actual* tests ≠ N *expected* tests
  - Since any item may crash the whole test sequence before the end

TAP protocol [2]

- **Test "plan"**
  - `1..N (todo X Y)?`
- **Tests**
  - `ok X - description`
  - `not ok X - description`
  - `ok Y # SKIP why`
  - `not ok Y # TODO why`
- **SKIP**
  - Skip test because of an external factor (missing module, OS, etc.)
- **TODO**
  - Not yet implemented functionality (might nevertheless be OK)

TAP protocol [3]

- **Cf Test::Tutorial talk**
  - TAP and much, much more by chromatic & Michael G Schwern
- **A few TAP interpreters**
  - Test::Harness
  - Test::TAP::Model (IM built upon TAP flow -> Test::TAP::HTMLMatrix)
- **More about this topic...**
  - Specification within TAP module
  - Wikipedia entry: Test_Anything_Protocol
  - Curtis “Ovid” Poe's talk: “TAP::Parser Will Be Test::Harness 3.0”
  - chromatic's article: “*An Introduction to Testing*”
  - Web site: http://testanything.org/

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

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

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

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

Traceability

Functional tests

POD tests

Synthesis

TAP matrix [1]

- **A synthetic representation of test suite results**
  - Handy since test population is most likely to grow a lot, isn't it?
- **Through a dedicated interpreter**
  - Test::TAP::Model::Visual module
  - This interpreter analyzes TAP flow to build a TTM IM
  - TTM IM is in turn translated into HTML (Test::TAP::HTMLMatrix)
- **Very easy to use**
  -
```
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;
```

TAP matrix [2]

- **Our previous `make test` now looks like**

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%

TAP matrix [3]

- **Yet another example: data transformation**

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%

Code coverage [1]

- **Simply load Devel::Cover module during test**
  - Beware! Increases CPU load!
  -
```
% 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

Code coverage [2]

- **Statements**
  - Were all instructions executed?
- **Branches**
  - Checks conditional branches alternatives (`if, ?:`)
- **Conditions**
  - Checks logical expressions possibilities
- **Subroutines**
  - Were all functions called?
- **POD**
  - Usage of POD::Coverage module

Documentation

- **Tested & covered code is not so bad, but...**
  - Documented code is way better!
- **Documentation is written in POD ("Plain Old Doc")**
  - Its syntax should be checked with Test::POD module
  - Via `t/pod.t` test (created by Module::Starter module)
- **POD coverage**
  - Measured by Test::POD::Coverage module
  - Verifies that every function has an associated POD documentation
  - Practically speaking, "only" checks for
    - `=item foo ... =cut`
    - `=head foo ... =cut`
  - Via `t/pod-coverage.t` test (created by Module::Starter module)

Kwalitee

- **For a more exhaustive definition see CPANTS site**
  - "CPAN Testing Service" – http://cpants.perl.org/kwalitee.html
  - Defines Kwalitee metrics ("*ALPHA – Hic sunt dracones!*")
- **Reckon Kwalitee metrics w/ Test::Kwalitee module**
  - Simply add `t/kwalitee.t` test to test suite:
  - ```
    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

- **Describe working hypothesis of a function**
  - Its limits, what it does not know how to handle at all
  - Better crash the whole program as soon as possible...
  - Rather than let it wildly go into an unpredicted direction!
  - A crash because of an assertion is often easier to solve
  - *"Dead programs tell no lies!"*
    – Hunt & Thomas in The Pragmatic Programmer
- **Assertions from Carp::Assert::More module**
  - Simple..... `assert_ + (is, isnt, like, defined, nonblank)`
  - Numerical..... `assert_ + (integer, nonzero, positive, ...)`
  - Reference...`assert_ + (isa, nonempty, nonref, hashref, listref)`
  - Array/hash.....`assert_ + (in, exists, lacks)`

Test::LectroTest [1]

- **Traditional tests are so-called “directed”**
  - Sequences of stimuli and comparisons to expected values
  - But we cannot possibly think about everything (# of combinations)
- **An alternative is “Constrained Random Testing”**
  - Let the machine do the dirty job instead, (pseudo-)randomly
- **Using Test::LectroTest module**
  - Stick a *type* to each function parameter (argh! *types* in Perl?)
  - Add constraints to parameters (~ restrain to sub-ensembles)
  - Do N iterations, measure FC, tweak constraints, goto 0
- **Not yet used in production but it's cooking!**
  - Alter-ego in hardware verification world is (coded in SystemVerilog)

Test::LectroTest 2

- **The following code**

```
use Test::LectroTest::Compat; # ::Compat allows for Test::More interfacing
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 => 'every possible foo input' ;

holds( $property, trials => 100 ); # proves that property "holds" over 100 random inputs
```

Reference model

Constraints on inputs

Comparison to reference

- **Proves that foo ('bar') ≠ 'foo'**

```
#   Failed test 'property 'every possible foo input' falsified in 4 attempts'
#   in lt_foo.t at line 30.
#          got: undef
#     expected: 'foo'
#     Counterexample:
#     $s = "bar";
```

- **FC ≈ statistics over sets of input values**

Refactoring 101

- **Refactor early, refactor often**
  - Restlessly fight ever-growing entropy
  - Due to bug fixes, new features or... “*clever tricks*”!
  - Test suite *should* ensure nothing got broken in the process (see FC)
- **Beware! Only on development branch!**
  - Production branch should remain untouched unless proven buggy
- **Beware! Neither add any feature nor fix any bug!**
  - Only make the code more concise/readable/testable... *elegant*!
  - KISS principle: “*Simplicity is pre-requisite for reliability.*” – EWD498
  - Commit by small changeset increments (easier to trace/rollback)
- **Refactoring 102...**
  - See Michael G Schwern's talk: “*Tales Of Refactoring!*”

Non-executive summary...

- **A priori**
  - Read, learn and do not hesitate to ask questions (then *evolve* ;^)
  - Use field-proven tools (SCM, RT, editors, etc.)
  - Have “good” practices
  - Do not reinvent the wheel
  - Write tests (and even documentation if you dare!) first
- **A posteriori**
  - Use TAP test protocol and one of it's dedicated interpreters
  - Analyze code coverage (≠ FC!) ratios and POD coverage ratios
  - Insert assertions into your code
  - Generate synthetic test reports
  - Even let the machine do the dirty work instead with CRT!

Social engineering

- **Like a lot of human activities...**
  - There is *technique* **and** there is *commitment*
- **Technique**
  - I've been ranting about this for 40 (darn long isn't it?) minutes
  - And this is far from being exhaustive!
- **Commitment**
  - Without *motivation*, no Kwalitee!
  - This is a *path* (to follow) rather than a *destination* (to reach)
- **One last quote as a conclusion...**
  - *"At that time (1909) the chief engineer was almost always the chief test pilot as well. That had the fortunate result of eliminating poor engineering early in aviation."* – Igor Sikorsky

Questions?

On the web...

- **Hoplites just want to have fun...**
  - 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