

# Day 10: Correctness

Suggested Reading:

*Programming Perl* (3rd Ed.)

Chapter 20: The Perl Debugger

<http://perldoc.perl.org/> — Modules  
Test::Simple, Test::More, Test::Harness

# Homework Review

# Correctness

**I wrote a script!**

**\o/**

**Did I do it right?**

**>\_<**

## What Does “Do It Right” Mean?

|                        |   |
|------------------------|---|
| <b>functionality</b>   | Does it work correctly?                 |
| <b>reliability</b>     | Does it work every time?                |
| <b>usability</b>       | Is it easy to use?                      |
| <b>efficiency</b>      | Is it fast? Low memory, disk, I/O, ...? |
| <b>maintainability</b> | Is it easy to change?                   |
| <b>portability</b>     | Does it work well everywhere?           |

(adapted from ISO 9126-1)

## What About ... Not So Right?

### Failure

**An *event*: Something went wrong; unexpected behavior**

Hardware

Network

Data

User

Software

### Defect

**A *mistake*: Something is likely or certain to cause a failure**

Cracked solder joint

Flaky router

Data-entry errors

Hangover

Bugs!

## How to Test / Debug

- “Try it out”
- **print()** statements / logging
  - see, e.g., Log::Log4perl
- Debugger
- Automated testing
  
- Code review
- Formal analysis

# Manual Testing



## Manual Debugging: print()

```
my $DEBUG = 1;
sub convert {
    my ($from, $to, $value) = @_;
    print "from = $from\n" if $DEBUG;
    print "to = $to\n" if $DEBUG;
    my $meters = $value * $UNITS{$from};
    print "meters = $meters\n" if $DEBUG;
    my $result = $meters / $UNITS{$to};
    print "result = $result\n" if $DEBUG;
    return $result;
}
```

# Debuggers

## Common debugger features:

- View code
- Step through (running) code, line-by-line
- Examine variables
- Run and stop at breakpoints
- Stack traces
- Watch points

# The Perl Debugger

```

List/search source lines:
  l [ln|sub]  List source code
  - or .     List previous/current line
  v [line]   View around line
  f filename View source in file
  /pattern/ ?patt? Search forw/backw
  M         Show module versions

Debugger controls:
  o [...]   Set debugger options
  <[<|{|{|>|>] [cmd] Do pre/post-prompt
  ! [N|pat] Redo a previous command
  H [-num]  Display last num commands
  = [a val] Define/list an alias
  h [db_cmd] Get help on command
  h h      Complete help page
  |[[]db_cmd Send output to pager
  q or ^D  Quit

Control script execution:
  T         Stack trace
  s [expr]  Single step [in expr]
  n [expr]  Next, steps over subs
  <CR/Enter> Repeat last n or s
  r         Return from subroutine
  c [ln|sub] Continue until position
  L         List break/watch/actions
  t [expr]  Toggle trace [trace expr]
  b [ln|event|sub] [cmd] Set breakpoint
  B ln|*   Delete a/all breakpoints
  a [ln] cmd Do cmd before line
  A ln|*   Delete a/all actions
  w expr   Add a watch expression
  W expr|* Delete a/all watch exprs
  ![!] syscmd Run cmd in a subprocess
  R         Attempt a restart

Data Examination:
  expr      Execute perl code, also see: s,n,t expr
  x|m expr  Evals expr in list context, dumps the result or lists methods.
  p expr    Print expression (uses script's current package).
  S [![]pat] List subroutine names [not] matching pattern
  V [Pk [Vars]] List Variables in Package. Vars can be ~pattern or !pattern.
  X [Vars]  Same as "V current_package [Vars]". i class inheritance tree.
  y [n [Vars]] List lexicals in higher scope <n>. Vars same as V.
  e        Display thread id      E Display all thread ids.

For more help, type h cmd_letter, or run man perldebug for all docs.

```

# Automated Testing

## Automated Testing

- Write software to test (other) software
- Humans vs. machines
- Types of automated tests
  - Unit tests
  - Functional tests
  - Performance tests

## Test::Simple

Run a `.t` file to test a module

```
#!/usr/bin/perl
use strict; use warnings;

use SomeModule; # to be tested

use Test::Simple tests => 2;

ok(is_doing_ok(), 'doing ok');
ok(the_result() == 7, 'value ok');
```

## Test::More

```
use Test::More tests => 6;

ok(is_ok(), 'is OK');
is($the_answer, 42, 'answer ok');
isnt(exit_status(), 1, 'syscall');
like($name, qr/tim/i, 'good name');
unlike($result, qr/error/i, 'test');
diag("current value of name = $name");
SKIP: {
    skip 'not available', 1 if ...;
    ok(...);
};
```

## Testing a Standalone Script

```
use Getopt::Long;
GetOptions('test' => \&run_tests);

# main script & subroutines here

sub run_tests {
    require Test::More;
    Test::More->import;
    plan(tests => nnn);
    # test subroutines here
    exit 0;
}
```



## Unit Testing Tips

- Test logical chunks of code — usually subroutines
- Aim for reasonable coverage
- Run often!
  - After every (significant) change
  - Before you use, hand in, commit, ...
- Encode past failures as tests

# Test-First Development

- Radical idea: Write tests **FIRST**
- Then write code until tests pass
- Clarifies and documents design
- And of course... is useful for testing!

<http://junit.sourceforge.net/doc/testinfected/testing.htm>

# Wrap Up

## Other Scripting Languages

- “Try it out” and printing/logging always work
- Most have debuggers and/or interactive modes
- Unit testing:
  - Most others are based on jUnit
  - Expect similar and richer assertions
  - Introspection rocks!

## Homework

- Write a couple of simple but non-trivial functions
- Write unit tests against them
- Make script work in “test” and normal modes