Best Programming Practices

Ashish Mahabral, Caltech
Ay/Bi 199b
31 Mar 2011
The zen of bug-free programming

- If debugging is the process of removing bugs, programming must be the process of introducing them.
  - Edsger W. Dijkstra (1930-2002)

Don’t program!
Obfuscated programming contests

- touch selfreproducingprogram.c
- makefile:
  - cp selfreproducingprogram.c a.out
  - chmod 755 a.out
- ./a.out
• Programming style
• Programming tools

My own experience/mistakes

The Pragmatic Programmer
By Andrew Hunt and David Thomas

Perl Best Practices
By Damian Conway
The scene keeps changing

- Drupal [http://drupal.org/node/287350](http://drupal.org/node/287350)

... and yet the basics stay the same
Coding by instinct

• Variable names (caps, underscores, ...)
• Types of loops (for, while, ...)
• Formatting
  – Indents, brackets, braces, semicolons
• Procedural versus object oriented approach

Conscious and consistent programming style
Necessary ingredients

- Robustness
- Efficiency
- Maintainability
Robustness

- Introducing errors
  - checking for existence (uniform style)

- Edge cases
  - 0? 1? last?

- Error handling
  - exceptions? Verifying terminal input

- Reporting failure
  - Traces? Errors don’t get quietly ignored
Efficiency

• Working with strength
• Proper data structures
• Avoiding weaknesses
• Dealing with version changes (backward compatibility)
Maintainability

• More time than writing
• You don’t understand your own code
• You yourself will maintain it
• Consistent practices
  – Braces, brackets, spaces
  – Semicolon (after last statement)
  – Trailing , in lists
  – Linelengths, tabs, blank lines
• cb, bcpp, perltidy, jacobe, Jxbeauty
• my @countries = ( USA, UK, UAE, Ukraine );

• my @countries = ( USA, UK, UAE, Ukraine, );
• every piece of code splits the universe in two possibilities.
• which future are you coding for?
• if your code is most consistent, you have least to fear.
Some simple recommendations

• Use underscores
  – $tax\_form$ rather than $taxForm$

• Don’t use abbrvs
  – don’t drop all vowels if you do

• Don’t use single letter variable names
  – Except perhaps in trivial small loops

• Don’t use too common words as variable names
  – e.g. no, yes, count, left, okay

• Empty strings: name and use them
  – my $empty\_string$ = “ “;

• Constants: use Readonly
  – my READONLY $\Pi$ = 3;
• easy development versus easy maintenance
  – projects live much longer than intended
  – adopt more complex and readable language
• check requirements
• design, implement, integrate
• validate
• Don’t trust the work of others
  – Validate data (numbers, chars etc.)
  – Put constraints (-90 <= dec <= 90)
  – Check consistency
• Don’t trust the work of others
  – Validate data
  – Put constraints
  – Check consistency

• Don’t trust yourself
  – Do all the above to your code too
Design by contract (Eiffel, Meyer ’97)

- Preconditions
- Postconditions
- Class invariants

Be strict in what you accept
Promise as little as possible
Be lazy

Inheritance and polymorphism result
• Crash early
  – Sqrt of negative numbers (require, ensure, NaN)
• Crash, don’t trash
  – Die
  – Croak (blaming the caller)
  – Confess (more details)
  – Try/catch (own error handlers e.g. HTML 404)
• Exceptions – when to raise them
  – should it have existed?
  – Don’t know?
sub locate_and_open {
    open my $fh, '<', "filename";
    return $fh;
}
sub load_header_from {
    TRY TO READ HEADER HERE
}
my $fh = locate_and_open($filename);
my $head = load_header_from($fh);
sub locate_and_open {
    open my $fh, '<', "filename" or croak "can't";
    return $fh;
}
my $fh = locate_and_open($filename);
my $head = load_header_from($fh);
If(my $fh = eval { locate_and_open($filename)})){
    my $head = load_header_from($fh);
}
else{
    carp "Couldn’t access $filename.\n";
}
• Tests
• Comments
• Arguments
• Debugging
Tests

• Test against contract
  – Sqrt: negative, zero, string
  – Testvalue(0,0)
  – Testvalue(4,2)
  – Testvalue(-4,0)
  – Testvalue(1.e12,1000000)
• Test harness
  – Standardize logs and errors
• Test templates
• Write tests that fail
All software will be tested

• If not by you, by other users!
  – perl Makefile.pl
  – make
  – make test
  – make install

• Don’t use code you do not understand
Source Code Control

- SVN
  - Checkin
  - Checkout
  - Comment
  - Merge

Git, google docs, wiki, trac

http://img.idealwebtools.com/blog/svn.gif
Modification cycle

• write test
• run and make sure it fails
• Checkout
• change, comment, edit readme etc.
• Compile
• run: make sure test passes
• checkin
Comments

• If it was difficult to write, it must be difficult to understand
• bad code requires more comments
• tying documentation and code
• use Euclid;
Documentation/comments in code

• List of functions exported
• Revision history
• List of other files used
• Name of the file
Documentation

• Algorithmic: # full line comments to explain the algorithm
• Elucidating: # end of line comments
• Defensive: # Has puzzled me before. Do this.
• Indicative: # This should rather be rewritten
• Discursive: # Details in POD
Arguments

• Don’t let your subroutines have too many arguments
  – universe(G,e,h,c,phi,nu)

• Look for missing arguments

• Set default argument values

• Use explicit return values
Needing/demanding arguments

- unless (@ARGV == 4) { exit; }
- my ($a, $b, $c, $d) = @ARGV;

use Getopt::Euclid;  # not just demands arguments

# but provides constraints
PROMPT> pq_images.pl
Missing required arguments:
   -r[a] [=] <RA>
   -d[ec] [=] <Dec>
(Try: pq_images.pl --help)
PROMPT>
PROMPT> pq_images.pl --help
Usage:
   pq_images.pl -r <RA> -d <Dec> [options]

Required arguments:
   -r[a] [=] <RA>
       Specify RA in degrees [0 <= RA <= 360]

   -d[ec] [=] <Dec>
       Specify Dec in degrees [PQ: -25 <= Dec <= 25]

Options:
   -i[d] [=] <id> [string]
       ID of the object
-c[leanup] [=] <cleanup>
  Level of cleanup after the program is done [default: 2] 0: Do not remove anything 1: Remove everything except individual mosaics (and final product) 2: Leave only final coadded image

-v
  --verbose
    Print all warnings

--version
--usage
--help
--man
  Print the usual program information
PROMPT>
PROMPT> pq_images.pl --man

AUTHOR
   Ashish Mahabal <aam@astro.caltech.edu>

BUGS
   There are undoubtedly serious bugs lurking somewhere in this code. Bug
   reports and other feedback are most welcome.

COPYRIGHT
   Copyright (c) 2007, Ashish Mahabal. All Rights Reserved. This module is
   free software. It may be used, redistributed and/or modified under the
   terms of the Perl Artistic License (see
   http://www.perl.com/perl/misc/Artistic.html)
use Getopt::Euclid;
...
=head1 REQUIRED ARGUMENTS
=over
=item -r[a] [=] <RA>
Specify RA in degrees [0 <= RA <= 360]
=for Euclid:
  RA.type: number >= 0
  RA.type: number <= 360
=item -d[ec] [=] <Dec>
Specify Dec in degrees [PQ: -25 <= Dec <= 25]
=for Euclid:
  Dec.type: number >= -25
  Dec.type: number <= 25
=back
Debugging

• there will be bugs!
• the only bugfree program is one that does not do anything
• tests: write unit tests first
• make sure the program compiles without warnings (perl -c)
• make bugs reproducible (with a single command)
• visualize the data
• ddd or perl -d
• Breakpoints
• use Smart::Comments;

http://www.gnu.org/software/ddd/plots.png
use Smart::Comments;

### seeing: $seeing
### calcmag: $cmag
### calcmag2: $cmag2;
When you find a bug ...

• check boundary conditions
  – first and last elements of lists
• describe the problem to someone else
• why wasn't it caught before
• could it be lurking elsewhere (orthogonality!)
• if tests ran fine, are the tests bad?
• (non)Duplication
• Orthogonality
• Refactoring
Duplication

- Don't repeat yourself
- Impatience
- Reinventing wheels
Orthogonality

- Decouple routines
- Make them independent
- Change in one should not affect the other
- Changes are localized
- Unit testing is easy
- Reuse is easy
- If requirements change for one function, how many modules should be affected? 1
- Configurable
sub line{
  my ($startpoint, $endpoint, $length);
  ...
}

Choose a template

Choose a custom look for your blog. You can easily change the template later, or even create your own custom template design once your blog is set up.

Minima
Created by: Douglas Bowman
preview template

Minima Black
Created by: Douglas Bowman
preview template

Sample Blog

CONTINUE
• if while entertaining libraries you need to write/handle special code, it is not good.
• avoid global data
• avoid similar functions
• even if you are coding for a particular flavor of a particular OS, be flexible
Refactoring

• Early and often
  – Duplication
  – Non-orthogonal design
  – Outdated knowledge
  – Performance
• Don’t add functionality at the same time
• Good tests
• Short deliberate steps
Portfolio building

• learn general tools, invest in different ones
  – plain text
    • easier to test (config files, for instance)
  – Shells
    • find, sed, awk, grep, locate
    • .tcshrc, .Xdefaults
  – learn different (types of) languages
  – Editor
    • if you know emacs, learn just a little bit of vi
    • Configurable, extensible, programmable (cheat sheet)
      – syntax highlighting
      – auto completion
      – auto indentation
      – Boilerplates
      – built-in help
• Text manipulation
  – perl and ruby are very powerful
Metaprogramming

• Configure
• Abstraction in code, details in metadata
  – Decode design
  – Pod files (plain old documentation)
• Code generators
  – make files, config files, shell scripts., ...
• Active code generator:
  – Skyalert (streams)
    • new transient
    • obtain new data
    • incorporate it
    • if certain conditions met,
      – run other programs
      – or raise alerts
      – drive other telescopes
      – and obtain feedback
Workflow

• Improving concurrency
• Unified Modeling Language (UML) diagrams
• Architecture
  – Action
  – Synchronization
  – Connect actions
PQ/SDSS coadds

new transient

various VO tools

get cutouts

ensure it is not artifact

get other info

subscribers in wait

see if alert condition is met

alert issued
Publish-subscribe rather than push

• Allow people to subscribe
• Let them subselect
• Allows separate view of model

Skyalert http://www.skyalert.org
Before the project

• Dig for requirements
• Document requirements
• Make use case diagrams
• Maintain a glossary
• document
• Don’t optimize code – benchmark it
• Don’t optimize data structures – measure them
• Cache data when you can – use Memoize
• Benchmark caching strategies
• Don’t optimize applications – profile them (find where they spend most time)
use Benchmark qw( cmpthese );

my @sqrt_of = map { sqrt $_ } 0..255;

cmpthese -30, {
    recompute => q{ for my $n (0..255) {
        my $res = sqrt $n
    } },
    look_up_array => q{ for my $n (0..255) {
        my $res = $sqrt_of[$n]
    } },
};
Summarizing ...

• Software entropy
  – Fix broken windows
• Know when to stop
  – Don’t overperfect
• Widen knowledge portfolio
  – Hotjava
  – Postscript
  – vi/emacs
• Languages/tools/OSes/editors
  – 99 bottles of beer
  – Programming shootout
  – Project Euler
    • Python
    • Perl
    • J
    • Haskell
Whats the lesson?

• Chain as weak as its weakest link
• Comment! For others and for yourself
• Tests!
• Orthogonality
• Don’t duplicate
• Designing by contract
• Know the features
• Review/balance
  – Public forums
    • Ask specific things
    • Check FAQs, webresults etc.
  – Maintain your own bookmarks
• Use wikis
• Use SVN, trac
• CHECK REPOSITORIES (like CPAN)
• Law 1: Every program can be optimized to be smaller.

• Law 2: There's always one more bug.

• Corollary: Every program can be reduced to a one-line bug.

From a Bug’s life