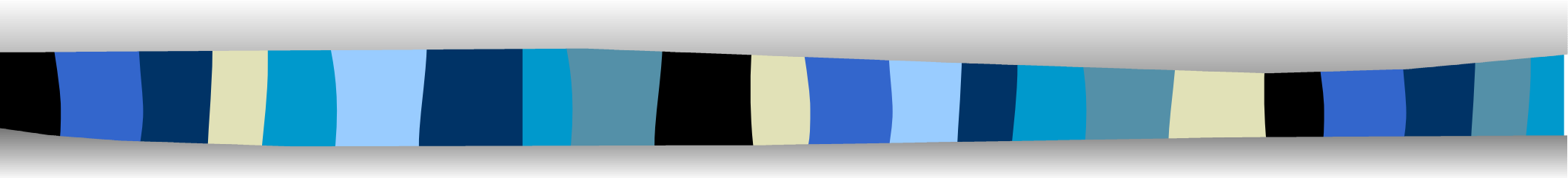
Some good development practices (not only in NLP)



Zdeněk Žabokrtský, Jan Štěpánek

Contents:

- testing
- bug reporting
- benchmarking
- profiling
- code reviewing

Testing

- *AHHHHHHH!!!! NOT TESTING! Anything but testing! Beat me, whip me, send me to Detroit, but don't make me write tests!* (CPAN, Test::Tutorial)

- most developers hate testing...
- ... but the better tests, the less need for debugging
- debugging
  - twice as hard as writing the code
  - usually more painful than writing tests in advance

Testing, cont.

- Ideally, you should write the test cases first.
- There should be tests for each module
- Automatize your tests
- Standardize your tests

Testing in Perl

- Read Test::Tutorial at CPAN, use Test::More
- Example:

```
use Test::More tests => 3;
use MyTagger;
my @words = qw(John loves Mary);
ok (MyTagger::tag() == 0, "survives empty sentence");
ok(
    scalar(MyTagger::tag(@words)) == scalar(@words),
    "one tag per word"
);
ok(
    join(" ",MyTagger::tag(@words)) eq "NNP VBZ NNP",
    "simple sentence tagged correctly"
);
```

Bug reporting

- As a programmer, sooner or later you start sending and receiving bug reports.
- Try to avoid the following scenario:
  - Module user: "*Hi, your module ABC does not work. Jim*"
  - Module author: "*Grrrrr!*"

Bad bug reports

- *I just clicked on ABC and it crashes.*
- *ABC completely fails.*
- *ABC is really slow.*
- *ABC used to work.*
- *ABC happens sometimes.*
- *The error message is stupid.*

Bug reproducibility

- The main aim of your bug report: it should allow the programmer to **reproduce the failure** and to see it with his/her own eyes
- Before sending the bug report, make sure you can reproduce it several times
- Try to **isolate** the bug to minimize the requirements needed for reproducing the bug (i.e., find the minimal failing test case)

Writing bug reports

- be precise, be clear, be specific
- describe steps to reproduce the failure (ideally on a fresh system)
- provide details: complete error logs, test case, test data, module versions, platform, OS...
- try to diagnose the failure yourself (but clearly distinguish your speculations from the observations)
- if you find a solution, offer a patch
- be polite

- Read more e.g. at http://www.chiark.greenend.org.uk/~sgtatham/bugs.html

Bug reports in Perl

- provide the version of your Perl
  - perl -v
- perlbug

Benchmarking

- benchmarking (in CS) = performance evaluation
- "Premature optimization is the root of all evil"
- unless you are familiar with Perl internals, your intuitions about the relative performance of two solutions might be unreliable
- --> don't optimize code – benchmark it!
- Rough benchmarking on command line: use time

```
time perl -e '@a=map{$_**2}(1..1000000)'
time perl -e 'for(1..1000000){push @a,$_**2}'
```

Benchmarking in Perl

```
use Benchmark qw(:all);

my @myarray;
my %myhash;

my $size = 100000;

foreach my $i (0..$size-1) {
    $myarray[$i] = $i;
    $myhash{$i} = $i;
}

my $count = 10000000;

cmpthese($count, {
    'hash write' => '$myarray[int(rand($size))] = 10',
    'array write' => '$myhash{int(rand($size))} = 10',
});
```

Benchmarking in Perl, learn more

- http://www252.pair.com/~comdog/Talks/benchmarking_perl.pdf
- http://www252.pair.com/comdog/mastering_perl/Chapters/06.benchmarking.html

Profiling

- *My program is slow. What should I focus on to make it faster?*
- Don't speculate – measure!!!
- profiling = analysis of a program's behavior using information gathered as the program executes
- profiler = a performance analysis tool that measures the frequency and duration of function calls (or other characteristics)

Profiling in Perl

- use Devel::DProf module
  - gather the runtime info:

    ```
    perl -d:DProf mytestscript.pl
    ```

  - view it

    ```
    dprofpp tmon.out
    ```

- use Devel::NYTProf module
  - gather the runtime info:

    ```
    perl -d:NYTProf some_perl.pl
    ```

  - convert it to html

    ```
    nytprofhtml
    ```

  - view it (by any browser)

    ```
    konqueror nytprof/index.html
    ```

Code reviewing

- code review = systematic examination of a source code written by someone else
- both formal and informal
- code reviewing
  - improves code quality
  - improves your own programming skills! (learn from masterpieces to become a master)
- learn to criticize constructively, learn to accept (and profit from) the criticism
- "*Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*" (Martin Golding)

Code reviewing

- You should look at
  - functionality (does it work as expected)
  - design quality (modularity, balanced APIs, algorithmization)
  - maintainability (coding style, readability)
  - coverage by tests
  - documentation coverage