Testing software

Harry Powell MRC-LMB

Testing has different purposes:

- do existing functions still work?
- do new functions work?
- does it give the customer what they want?
- is it an improvement over what existed before?
- performance testing

and different phases:

- alpha testing (core developers)
- beta testing (other developers + trusted users)
- release (the wider community)

Alpha testing is performed by the core developer(s) before any users see anything

- Uses standard input to give standard output
- tests new features (makes sure they don't break existing ones)
- Should get rid of all obvious bugs

Beta testing should be carried out by a small group of *intelligent* users:

- Once alpha testing is complete and no obvious bugs exist
- Makes sure features behave as expected in a non-sterile environment
- Reliable experienced users who will give full reports of failures (log files, circumstances, etc.)

Release is to the world at large ("real" users) and provides the most brutal testing

- you will enter the wonderful world of user support
- expect to get bug reports like "I pressed a button and it broke"
- users find "odd" bugs which arise from abuse of your carefully crafted software
- naïve users will find innovative ways of running (ruining?) your program

Reasons for testing existing software:

1. Want an identical result
2. Will accept a similar result
3. Want a different (better) result
4. Portability across platforms
5. Checking installation & performance

Types of testing:

1. background/batch/command-line
2. effect of different input
3. interactive *via* a GUI

You may want an *identical* result if:

- you have fixed an “unrelated” bug and don’t want to change the outcome
- you have tidied up the code (e.g. rewritten a routine)
- you have changed the optimization in compilation (*e.g.* from “`-O0 g3`” to “`-O5 -funroll-loops`”)
- you have used a different compiler (*e.g.* xlf instead of g77)
- you’ve *only* changed OS (e.g. rebooted from Linux to MS-Windows on the same box)
- you are running the program in different modes with the same input (*e.g.* batch mode or through a GUI)

You may accept a *similar* result if:

- you have just ported to a different chip (e.g. from PowerPC to i686)
- you are using a “random” seed
- your input is different
- you are using a different compiler

e.g. SGI Octane vs HP Alpha (autoindexing tetragonal lysozyme):

```
alf1_harry> diff alpha.lp irix_6.5_64.lp
186c186
<   20 306     tI   110.12   115.94    36.83    71.7  90.0  90.2
---
>   20 306     tI   110.12   115.95    36.83    71.7  90.0  90.2
188c188
<   18 204     oI    36.83   110.12   115.94    89.8  71.7  90.0
---
>   18 204     oI    36.83   110.12   115.95    89.8  71.7  90.0
223c223
< Initial cell (before refinement) is   77.8506   77.8506   36.8255
    90.000  90.000  90.000
---
> Initial cell (before refinement) is   77.8507   77.8507   36.8255
    90.000  90.000  90.000
```

You may want a *different (better)* result if:

- you have just spent six months improving an underlying algorithm
- you've implemented something new
- there are better traps for bad input
- you’ve been bug fixing

Batch testing (once set up) is easier, more reliable and less tedious than running a GUI

set up a shell script to
- run the program(s)
- check the output against a standard
- do the work while you do something more interesting

then expand the functionality as you realize you need it

Using a GUI usually means that you have to sit at a terminal and work through sets of examples and compare the answers (but with a scripted GUI (*e.g.* written in Tcl or Python) this can also be automated to some extent)...

Use a shell (csh or bash) or a scripting language?

csh (or tcsh) is the most common shell used by protein crystallographers
bash is most commonly used by computer scientists
zsh is a new tcsh-like shell which is becoming popular.
small molecule & powder crystallographers are often less familiar with shells

Largely a matter of personal choice, but bash’s syntax is a little more flexible and internal counters can be larger (but csh mutates less between platforms, and bash is missing on many SGIs).

For small scripts, a shell language is suitable, but for rigorous testing a proper scripting language may make further development easier (but remember James Holton's Elves - 63,000 lines of csh).

```
#!/bin/bash -f
export IPMOSFLM=/Users/harry/mosflm625/bin/ipmosflm
export LOGFILE=mosflm625_osx_august_01.log
if [ ! -e $IPMOSFLM ]
then
    echo $IPMOSFLM doesn\'t exist - exiting now\!\!
    exit
fi
#
echo Executable $IPMOSFLM | tee $LOGFILE
echo Running test on `date` >> $LOGFILE
#
I=1
while [ $I -le 10 ]
    do
        TIME_USED=$( (time ${IPMOSFLM} < test_$I > mosflm.lp) 2>&1 > /dev/null )
        echo  Run \#$I: cpu time: `echo $TIME_USED | awk '{print $4}'` >> $LOGFILE
        mv mosflm.lp mosflm_$I.lp
        mv lys_fine_002.mtz $I.mtz
        mv SUMMARY summary.$I
        /bin/rm -f GENFILE
    let I=I+1
    done
#
I=1
while [ $I -le 10 ]
    do
        wc -l mosflm_$I.lp IPMOSFLM_$I.lp
    let I=I+1
    done
echo finished test at `date` >> $LOGFILE
```

```
! DO NOT ADD or REMOVE STUFF FROM THIS FILE
! It is intended to test mosflm in a background job with the
! following sequence of processes:
!
! (1) Autoindex from two images
! (2) estimate mosaicity from the first
! (3) postrefinement
! (4) integration
!
BEAM  149.79  150.87
GAIN 1.80
ADCOFFSET   6
DISTANCE  195.132
TEMPLATE lys_fine_###.pck
NEWMAT postref2.mat
MOSAIC ESTIMATE
AUTOINDEX DPS IMAGE 2 PHI 0 0.2 IMAGE 51 PHI 9.8 10.0
GO
POSTREF MULTI SEGMENT 2
PROCESS 2 TO 5 START 0 ANGLE 0.2
GO
PROCESS 47 TO 50 START 9 ANGLE 0.2
GO
#
POSTREF MULTI NOSE FIX ALL
PROCESS 2 TO 51 START 0 ANGLE 0.2
GO
```

```
[g4-15:~/test/lys_fine] harry% ./testit.sh
Executable /Users/harry/mosflm625/bin/ipmosflm
   19010 mosflm_1.lp
   19191 IPMOSFLM_1.lp
   38201 total            (22313 lines different)
   19366 mosflm_2.lp
   19229 IPMOSFLM_2.lp
   38595 total             23106   "       "
   20329 mosflm_3.lp
   19229 IPMOSFLM_3.lp
   39558 total             25123   "       "
          .
          .
          .

[g4-15:~/test/lys_fine] harry% more mosflm625_osx_august_01.log
Executable /Users/harry/mosflm625/bin/ipmosflm
Running test on Mon Aug 1 15:18:58 BST 2005
Run #1: cpu time: 0m46.886s
Run #2: cpu time: 0m50.305s
Run #3: cpu time: 1m0.763s
Run #4: cpu time: 0m58.744s
Run #5: cpu time: 0m56.463s
Run #6: cpu time: 0m54.628s
Run #7: cpu time: 0m51.343s
Run #8: cpu time: 1m2.693s
Run #9: cpu time: 0m47.770s
Run #10: cpu time: 0m56.151s
finished test at Mon Aug 1 15:29:01 BST 2005
```

```
[macf3c-3:~/test/lys_fine] harry% diff mosflm_2.lp IPMOSFLM_2.lp | wc -l
   23106
[macf3c-3:~/test/lys_fine] harry% diff mosflm_2.lp IPMOSFLM_2.lp | more
1,3c1
<
<
<  ************ Version 6.2.5 for Image plate and CCD data 9th August 2005
     ***********
---
>  ************ Version 6.2.5 for Image plate and CCD data 30th June 2004
     ***********
                                        .
                                        .
                                        .
511c487
<  149.74  150.90  1.0014  195.40  1.0002    14    20  0.070 -0.311  0.000  0.000
---
>  149.74  150.90  1.0014  195.40  1.0002    14    19  0.070 -0.311  0.000  0.000
651a628,631
>
>  Detector distortion refinement using 50 SPOTS
>  Starting residual=0.163mm; Weighted residual  0.87
>  Residual after 1 CYCLE=0.115mm; Weighted residual  0.49
654c634
<  149.72  151.02  1.0010  195.33  0.9996     4     9  0.116 -0.322  0.000  0.000
---
>  149.72  151.02  1.0010  195.33  0.9996     4     8  0.116 -0.322  0.000  0.000
683a664,667
>
>  Detector distortion refinement using 24 SPOTS
>  Starting residual=0.090mm; Weighted residual  0.47
```

Is it an improvement over what existed before?

- produces results where it (or other software) didn't before
- faster (more streamlined code, better compilation, removal of bottlenecks)
- more accurate results (lower Rs, nicer peaks in maps)
- easier to use
- runs on a new platform

Performance testing:

Can inform choice of hardware/OS/compiler/flags e.g. for the batch test series earlier:

	clock time
Linux, Pentium, 3.2GHz, g77 3.4, -O2:	8m 37s
Linux, Pentium, 1.5GHz, g77 3.2, -O1:	16m 51s
Linux, Pentium, 1.5GHz, ifc, -O3:	25m 53s
OS X, Mac, 1.67GHz, g77, -O0:	17m 33s
OS X, Mac, 1.67GHz, g77, -O2:	10m 22s
OS X, Mac, 1.67GHz, g77, -O5 -funroll-loops:	10m 09s
OS X, Mac, 2.0 GHz, XLF -O2:	4m 55s
Tru64, Alpha, 500MHz, f77 -O2:	5m 41s
Irix 6.5, SGI, 400MHz, f77 -O2:	24m 09s

Can highlight particular problems or indicate improvements:

e.g. for Linux, NFS mounted disks can cause severe performance problems - caused by local/remote handshaking *every* time a read or write is performed.

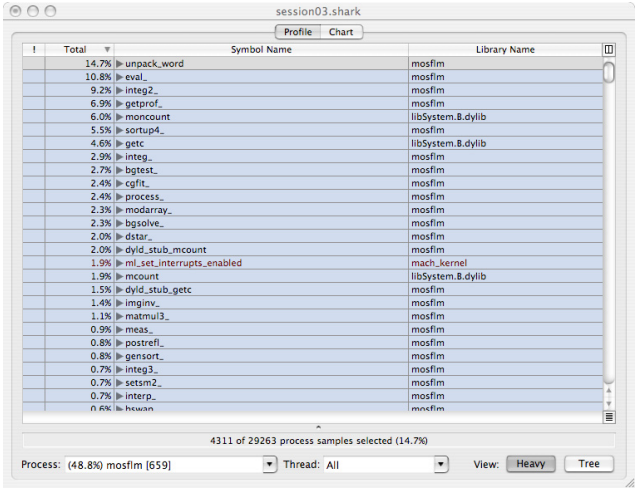
cure: (a) only use local disks
(b) buffer i/o to reduce the number of transfers

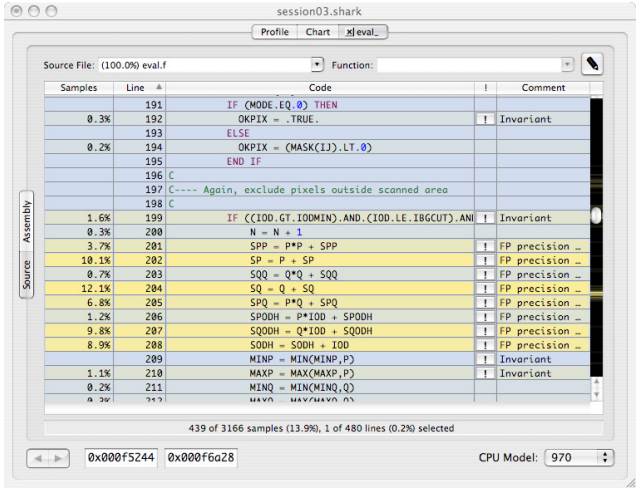
Profiling: use an external program to locate bottlenecks

e.g. Shark in OSX, gprof under other UNIXes; compile & link with flag "-pg", run the program and then

```
$ gprof <progname> gmon.out
                                .
                                .
                                .
granularity: each sample hit covers 4 byte(s) for 0.03% of 39.25 seconds

 %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
12.4      4.87     4.87   115121     0.04     0.08  _eval_ [4]
11.7      9.48     4.61    43166     0.11     0.19  _integ2_ [6]
10.1     13.44     3.96    43166     0.09     0.09  _getprof_ [11]
 8.5     16.76     3.32                             _moncount (7093)
 8.4     20.06     3.30       24   137.50   137.50  _rotate_clock90 [12]
 6.5     22.61     2.55   115148     0.02     0.02  _sortup4_ [14]
 5.7     24.84     2.23       24    92.92    92.92  _unpack_wordmar [16]
 4.0     26.42     1.58    49069     0.03     0.11  _integ_ [10]
 3.4     27.76     1.34       22    60.91   657.99  _process_ [3]
 3.4     29.08     1.32                             mcount (152)
```





Finally - test at all stages of development.

Modern OOP methodology recommends producing test classes for all important methods to check they work with model data before inclusion into your main program. Having the test class available makes it easier to investigate when someone breaks your program with unexpected input.