Testing software

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)

Harry Powell MRC-LMB

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

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

---

**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
```

---

**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)...
#!/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

[macf3c-3:~/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
39558 total25123   " ".
...
[macf3c-3:~/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

[p4-15:~/test/lys_fine] harry%./testit.sh
Executable /Users/harry/mosflm625/bin/ipmosflm
19010 mosflm_1.lp
19391 IPMOSFLM_1.lp
38901 total  38901 lines different
19366 mosflm_2.lp
19229 IPMOSFLM_2.lp
38595 total  38595 lines different
19229 mosflm_3.lp
19229 IPMOSFLM_3.lp
38588 total  38588 lines different
38588
...
[p4-15:~/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

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:

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Compiler</th>
<th>Flags</th>
<th>Clock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux, Pentium, 3.2GHz</td>
<td>g77</td>
<td>-O2</td>
<td>8m 37s</td>
</tr>
<tr>
<td>Linux, Pentium, 1.5GHz</td>
<td>g77</td>
<td>-O1</td>
<td>16m 51s</td>
</tr>
<tr>
<td>Linux, Pentium, 1.5GHz</td>
<td>ifc</td>
<td>-O3</td>
<td>16m 51s</td>
</tr>
<tr>
<td>OS X, Mac, 1.67GHz</td>
<td>g77</td>
<td>-O0</td>
<td>9m 33s</td>
</tr>
<tr>
<td>OS X, Mac, 1.67GHz</td>
<td>g77</td>
<td>-O2</td>
<td>10m 22s</td>
</tr>
<tr>
<td>OS X, Mac, 2.0 GHz</td>
<td>XLF</td>
<td>-O2</td>
<td>4m 55s</td>
</tr>
<tr>
<td>Tru64, Alpha, 500MHz</td>
<td>f77</td>
<td>-O2</td>
<td>6m 41s</td>
</tr>
<tr>
<td>Irix 6.5, SGU, 400MHz</td>
<td>f77</td>
<td>-O2</td>
<td>24m 09s</td>
</tr>
</tbody>
</table>
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 <prognme> 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  _mgetprof_ [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     2.95  115148     0.02     0.02  _srtup4_ [14]
5.7  24.84     2.23    24  92.92    92.92  _unpack_wordbar [16]
4.0  26.42     1.58  49069     0.03     0.11  _integ_ [10]
3.4  27.76     1.34                             _moncount (152)
3.4  29.08     1.32

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.