Software Testing for Sysadmin Programs

http://menlo.com/lisa-2015/s7/

apology & explanation

This is a new class and I'm trying to incorporate the current best practice of “active learning”; that is, rather than stand up and tell you things, I hope to engage you in discussion, Q&A, and exercises. As such, organizing the class materials is tricky because I'm not yet sure how I'll use them. The conference production schedule requires me to submit my files almost two full months in advance; rather than give you nothing, I've provided the latest materials I have, organized as best I can for now. By the time you get to the conference and receive this USB key, I will have greatly expanded and reorganized these notes; you'll receive a copy of them with the pre-class download (when you get the VM with the exercises loaded on it).

latest slides

- you can get the latest notes here:
  - http://menlo.com/lisa-2015/s7/

schedule

- 1:30 - 3:00 class
- 3:00 - 3:30 break
- 3:30 - 5:00 class

agenda / objectives

- interactive: if possible, more discussion than lecture
- hands-on: exercises
  - work individually on in groups
- explanation: detailed walk-through of some exercises
  - more in the beginning than the end
  - maybe a full walk-through of the final solution

agenda / objectives

- definitions, background, benefits
- exercises
- q&a: throughout the day

what is testing

- want to be clear because it affects everything that follows
- “system under test”: your program
- tests: separate code that proved the SUT is correct

what is testing

- SUT may (should?) include “defensive” code
  - check for errors
  - deal with unexpected results
  - handle variations
- but how do you know that code is correct?
- with testing (of course)

what is testing

- first test: DTRT with simple, most common, expected results all other tests: verify each and every variation, alternate code path (“code coverage”), error condition, most common unexpected results

testing terminology

- unit, integration, system, smoke, regression, etc.
- the distinction isn't really important
- better: small, medium, large
  - small: fast, no resources
  - medium: slower, some resources
  - large: very slow, lots of resources
- concentrate on small tests and lots of them
- maybe one medium or large test
  - if needed
  - maybe implement as --noop (if possible)
- even just small tests can help

why write tests

- s/w testing . . .
  - IS NOT PERFECT
  - DOES NOT produce bug-free code
  - WILL NOT find errors of intent
  - DOES NOT make you a better programmer

why write tests

- perfect is the enemy of good
  - IT DOESN'T HAVE TO BE PERFECT
  - Pareto Principle (80/20 rule)
  - avoiding even one failure may pay off
    - especially if it's a big failure

why write tests

- so what **IS** testing good for?
  - helps you write better software
    - more thought goes into it
    - often makes you re-examine the design
    - may help find bad assumptions
  - reduces risk when making changes
    - verify it works
    - make changes
    - see that it still works

why write tests

- it still won't be perfect
  - adding regression tests helps
  - adding more tests helps

why this class

- too many sysadmins think testing is hard
- it's not
- well, it may be
  - some (many?) sysadmin programs are written in shell (bash, ksh)
  - tests for shell are harder than
- this class shows the least painful way to test shell programs I've ever found

shell v. <language>

- you can apply this technique to other languages
  - perl, python, ruby, go, java, groovy
- **PLEASE DON'T**
- all those languages can do OO

OO is NOT inherently bad

- yes, it can be (and often is) overused (abused?)
- used in moderation it's a big win
  - especially in making it easier to test programs
- so please, if you're writing perl/python/ruby/go, take the time to learn just enough OO to write “proper” tests
  - it really will pay off in the end
- that said, everything today can be applied to programs in any language
  - but again, please use only for shell :-)

shell v. <language>

- if enough people are writing in “not shell” I can show basic OO and testing

why write tests first?

- or, if not first, at least at the same time as the code
- so why?
- forces you to think more about the design
- forces you to design testable software
  - more likely to introduce errors if you have to retrofit testing
  - less likely to add tests after software is written

TDD (**Sssssh**!)

- “Test-Driven Development”
- I'm not calling it TDD to avoid the hype/fervor/religion
- I believe the basic idea is sound
  - but I don't take it literally
- writing code and tests at the same time is a good compromise

the 50-cent tour

write skeleton

```
#!/bin/bash
exit 0
```

write first test

```
myprogram
ret=$?
if [ $ret -ne 0 ] ; then
      printf "expected 0, got $ret\n"
      exit 1
fi
```

- run test, everything should pass

write second test

```
myprogram no
ret=$?
if [ $ret -ne 1 ] ; then
    printf "expected 1, got $ret\n"
    exit 1
fi
```

- run tests, see #2 fails

add first feature

```
#!/bin/bash
if [ $1 = yes ] ; then
      exit 0
else
     exit 1
fi
```

- run tests, everything should pass

SEE THE PROBLEM?

- `myprogram no`
- `myprogram maybe`
- tests all pass

what's wrong?

- test #2 doesn't correctly check desired behavior
- is the assumption wrong?
  1. "no" returns 1, everything else 0?
  2. or "yes" returns 0, everything else 1?
  3. or even "yes" returns 0, "no" returns 1, everything else returns 2?

change tests and/or program

behavior 1

```
# (myprogram)
if [ $1 = no ] ; then
      exit 1
else
      exit 0
fi
```

behavior 2

```
# (test)
myprogram yes
ret=$?
if [ $ret -ne 0 ] ; then
      printf "expected 0, got $ret\n"
      exit 1
fi
```

behavior 2

```
# (more test)
myprogram no
ret=$?
if [ $ret -ne 1 ] ; then
     printf "expected 1, got $ret\n"
     exit 1
fi
```

behavior 3

```
# (test)
myprogram yes
ret=$?
if [ $ret -ne 0 ] ; then
     printf "expected 0, got $ret\n"
     exit 1
fi
```

behavior 3

```
# (more test)
myprogram no
ret=$?
if [ $ret -ne 1 ] ; then
     printf "expected 1, got $ret\n"
     exit 1
fi
```

behavior 3

```
# (more test)
myprogram maybe
ret=$?
if [ $ret -ne 2 ] ; then
     printf "expected 2, got $ret\n"
     exit 1
fi
```

behavior 3

```
# (myprogram)
case "$1" in
yes)
     exit 0
     ;;
no)
     exit 1
     ;;
```

behavior 3

```
# (myprogram cont’d)
*)
     exit 2
     ;;
esac
```

what's missing from #3?

```
# test for no argument
myprogram
ret=$?
if [ $ret -ne 2 ] ; then
     printf "expected 2, got $ret\n"
     exit 1
fi
```

just examples

- won't use shell for the tests
- didn't want to complicate things (yet)

my background

- because I want to let you know “where I'm coming from”

my background

- programmer (before we were called “developers”)
- programmer + “the guys who took care of the machine”
- full-time sysadmin
- technical trainer
  - programming, sysadmin
    - in particular, best practices
- sysadmin + programmer
  - writing programs to manage the systems
- developer
  - still writing programs related to sysadmin

my background

- one job was “serious” development in ksh & awk
- “software engineering” was important
- but it was still ksh & awk
- testing was hard
  - so we mostly didn't do it
  - relied mainly on inspection and review

my background

- joined MathWorks in 2013
- VERY strong focus on s/w testing
- I work for a group that builds test infrastructure
- we try to hold ourselves up as an example to others

what's the point?

- if sysadmin programs fail, Very Bad Things happen
  - `newfs` the wrong partition
  - mount the wrong partition in the wrong place
  - `rm -rf /`
- sysadmins should be writing software tests
- some (many?) sysadmin programs are written in shell
- testing shell programs is hard
  - or, at least, harder than other