Software Testing for Sysadmin Programs

Downloads:
VMs, exercises, and slides

Schedule

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

Before we start

• one definition
• one assumption

Definition

• “Testing” is ...

• You have a program
• You want to see if works
• So you write another separate program to test the first program

• The second program is the test
Assumption

• You agree that testing is a good thing
• Or, you want to learn how to do testing before deciding if it’s good

• I’m not going to spend any time convincing you that testing is good
  • But I will cover what it can and can’t do
  • and what it is and isn’t good for

• If you want to argue about testing you’re in the wrong class
Ask yourself

• “Is this the class I was hoping it would be?”

• If not, you can (and probably should) switch to another class

• I won’t be offended

• But I hope you’ll stay
Agenda and 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

• Exercises are optional but strongly encouraged
Agenda and Objectives

• Definitions, background, benefits
• Exercises
• Q&A: throughout the day
Agenda and Objectives

• I have to jump around a bit at the beginning
• A few “building blocks” to get started
• Then I’ll settle into a more logical progression
• Please bear with me
What is Testing

• I want to be clear because it affects everything that follows
• “System Under Test”: your program
  • a/k/a “SUT”
• 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)
My Background

• Because I want to let you know “where I'm coming from”
My Background

• Programmer (before we were called “developers”)
• Programmer + “the guy who took care of the machine”
• Full-time sysadmin
• Technical trainer
  • Programming, sysadmin
• 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
  • We relied mainly on inspection and review
My Background

• I joined MathWorks in 2013
• The company has a 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
Why Write Wests?

• Software testing . . .
  • IS NOT PERFECT
  • DOES NOT produce bug-free code
  • WILL NOT find errors of intent
  • DOES NOT make you a better programmer
    • OK, maybe it helps a little bit
Why Write Wests?

• 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 Wests?

• 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 Wests?

• 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
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
Testing Philosophy: “Standard”

• 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
Testing Philosophy: The Sysadmin Version

• It’s easiest to write “final outcome” tests
  • Did the program do what I expected it to do?
  • Did the program fail the way I expected it to fail?
  • Does the program correctly defend against errors?

• Because unit tests in shell are hard
  • Really hard
Testing Philosophy: The Sysadmin Version

• Other tests may be possible

• In general, more tests are better
  • No matter how you classify them
Testing Philosophy: Getting Started

• 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
Why Write Tests First?

• Or, if not first, at least at the same time as the code
• So why?
• It forces you to think more about the design
• It 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
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
The 50-Cent Tour
What Tests Do I Need?

• What does the specification say the program should do?

• Spec?

• Bwahahahahaha!
OK, let’s get real

• write just enough of a spec:
  1. “myprog yes” should exit with 0
  2. “myprog no” should exit with 1
  3. any other arg or no args -> exit with 2
Translate the Spec Into Tests

1. myprog yes, check $?, fail if not 0
2. myprog no, check $?, fail if not 1
3. myprog, check $?, fail if not 2
4. myprog maybe, check $?, fail if not 2

• Wait, where did #4 come from?

• Some requirements need multiple tests
Introducing ...

- Maven
- Spock
- Groovy

- On second thought, let’s write some code!
Exercise 1

• cd $HOME/lisa2015s7/exercises/ex01
• ls -lR *
  • pom.xml
  • src/main/bash/myprog
  • src/test/groovy/Test_myprog.groovy

• cat src/main/bash/myprog

• Boring!
Exercise 1

• Edit src/test/groovy/Test_myprog.groovy
  • vi, vim, nano, emacs, pico, ...

• Look for test #1
  • Ignore everything above and below
  • we’ll cover it later

• I’m betting you can figure out what this does with me having to explain it
Exercise 1

• Don’t worry, I will explain it

• My point is even though you’ve never seen any of this before, it’s pretty simple

• It will get more complicated but not much more

• We’ll have lots of time for explanations and questions
Exercises: How-To

• You write the tests
• I’ll write the program being tested
  • Well, I wrote a program to write the program being tested
  • It lets you pretend you’re writing the program being tested
  • But it lets you concentrate on the tests themselves

• Here’s how it works ...
Exercises: How-To

• Each exercise starts at “step 0”
  • An “empty” bash script
  • It will always exit with 99

• `step -e # -s #`
  • `-e #` is the exercise number
  • `-s #` is the step number
Exercises: How-To

• You’re at step 0
• mvn test
  • should fail
• Write your tests
• step –e 1 –s 1
• mvn test
  • Eventually this will pass
• Repeat edit (the tests) then (mvn) test until everything passes
Exercises: How-To

• When everything passes, start writing the next test
• `step -e # -s #+1`
• `mvn test`
• Repeat edit then test until it passes

• Repeat edit / step / test until no more steps
Exercise 1

• All the tests are written for you
• Delete the “@Ignore” line to enable the next test
• There are four steps but 5 tests
  • do 4 and 4a together

• For exercise 2 you’ll actually write the tests
  • The file and test skeleton already exists
Exercise 1

- `mvn test`
- `step -e 1 -s 1`
- `mvn test`
- `edit .../Test_myprog.groovy`
  - delete the @Ignore for test #2
- `mvn test`
- `step -e 1 -s 2`
- `mvn test`
Exercise 1

- edit .../Test_myprog.groovy
  - delete the @Ignore for test #3
- mvn test
- step -e 1 -s 3
- mvn test
- edit .../Test_myprog.groovy
  - delete the @Ignore for test #4 and #4a
- mvn test
Exercise 1

• `step -e 1 -s 4`
• `mvn test`

• Ta-dah!
Introductions

• OK, now let’s meet the players
• Maven
  • A dependency-based build system
• Spock
  • A testing framework
  • Uses Groovy
• Groovy
  • A scripting language
Choice of Weapons

• There are lots of test frameworks
  • Too many to count
  • Some work better than others
  • Most of them get the job done
Choice of Weapons

- My project at work was written in Groovy
- Maven is our preferred build system
- Spock was an easy choice
  - some other people were already using it
- It works for me
Choice of Weapons

• If you like it, great
  • if not, pick a framework you like more
  • any framework is probably better than no framework

• The principles still apply
  • Write a requirements “spec”
  • Turn that into tests
  • Write tests before writing code
    • or, at least, write them at the same time
Introducing Maven

- Driven by pom.xml
  - just copy the one I’ve provided
- export M2_HOME=/path/to/maven/install
- export M2=$M2_HOME/bin
- mvn test
  - To run tests
- mvn clean
  - To clean up
- That’s all!
Introducing Spock

• In `pom.xml`:

  ```xml
  <dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-core</artifactId>
    <version>1.0-groovy-2.4</version>
    <scope>test</scope>
  </dependency>
  ```
import spock.lang.*
class Test_example extends Specification {
    def "test number 1"() {
        setup:
            // set-up
        when:
            // stimulus
More Spock

def “test number 1”() {
    
    then:
        // response
    cleanup:
        // runs whether test passes or fails
}
}
More Spock

• Setup block: stuff to do before the stimulus block starts
  • Could be in stimulus but tradition uses setup instead
• Stimulus block: perform these actions
• Response block: test that these conditions are true
• Cleanup block: stuff to do after the test is complete
  • Runs whether test passes or fails
  • Have to be careful about errors in this block
• All are written in Groovy
More Spock

• That’s as much Spock as you need to know
  • There’s plenty more
  • We don’t need it here
  • You will probably never need it
  • But it’s easy enough to learn if you want to know more
Introducing Groovy

• The illegitimate love child of Java and Perl
• Most Java runs without change
• Has many Perl-like features and syntax
• Runs in the JVM
• Close enough to Perl you should be able to pick it up today
One Last Spock Thing

class Test_example extends Specification {
    def here = System.getProperty("user.dir")
    def prog = "${here}/src/main/bash/example"

• Adds portability, readability
Stimulus Block

• Usually looks like this:

```python
when:
    def p = "${prog} arg ...".execute()
p.waitFor()
```

• “Run $prog with arg(s) and wait for it to finish”
“def”

- Groovy supports strict typing
  - Process
  - String
  - List<String>
  - Map<String, String>

- Also allows dynamic typing
  - “def”
“def”

• I prefer strict typing
  • I know that’s not “the Groovy way”

• We’ll use “def” to keep things simple
Response Block

- \( p\text{.exitValue()} == 0 \)
- \( p\text{.exitValue()} != 0 \)
- \( p\text{.exitValue()} == 1 \)
- etc.
class Test_example extends Specification {
    def here = System.getProperty("user.dir")
    def prog = "${here}/src/main/bash/example"

    def "test for exit 0"() {
        when:
            def p = "${prog}".execute()
            p.waitFor()
        then:
            p.exitValue() == 0
    }
}
exercise 2

• Let’s extend the specification we wrote for exercise 1
exercise 2

1. “myprog yes” should exit with 0
   • should print “hello, world” on stdout

2. “myprog no” should exit with 1
   • should print “goodbye, cruel world” on stdout

3. any other arg or no args -> exit with 2
   • should print “what a maroon” on stderr
exercise 2

• Three (3) steps

• Use your tests from exercise 1 as an example
  • or even copy the file and just change the package name
Response Block

• p.text
  • stdout as a single string

• p.err.text
  • stderr as a single string

• Multiple lines saved as a single string with embedded newlines ("\n")
Response Block

• Save string before using them
  • Weird behavior if you don’t

• def err = p.err.text
• def out = p.text

• Save p.err.text first!
  • Again, weird behavior
Response Block

• `def out = p.text`

• `out == "hello, world"`
  • fails!

• `out == "hello, world\n"`

• That’s annoying
exercise 2 (again)

• You’ve got enough to start

• Work for a while then I’ll show you some tricks to make things easier

• Or read the next bunch of slides before you start 😊
Response Block

- Regular expressions to the rescue

- out =~ /hello, world/

- out =~ /^hello/
- out =~ /world$/
Response Block

• “Perl-ish” regular expressions

• Syntax is a little different

• Documented in java.util.regex.Pattern

• Mostly you can pretend it’s Perl
Response Block

• Sloppy way to match multiple strings
  • def out = p.text
  • out =~ /line 1/
  • out =~ /line 3/
  • out =~ /line 2/

• Notice that order doesn’t matter
Response Block

• What if order does matter?
  • def out = p.text.split(\n/)
  • out == [
    •   "line 1",
    •   "line 2",
    •   "line 3",
    • ]

• Checks order and number of lines
Response Block

• Check just the number of lines?
  • def out = p.text
  • out.split(/\n/).size() == 3
Response Block

• Test for no output
  
  • def err = p.err.text
  • def out = p.text

  • out == ""
  • err == ""

• Testing for “null” doesn’t work
Response Block

- Simple file tests
  - new File("some-file").exists() == true
  - new File("other-file").exists() == false
exercise 3

• Just a small variation on exercise 2

3. any other arg or no args -> exit with 2
   • should print “what a maroon” on stderr
   • write everything on the command line to the file “myprog.log”

• Just one step
  • Because obviously you’ll copy the code from exercise 2
Maven and files

• Where does Maven run?
  • The “project directory”

• Where’s that?
  • “here”
  • Where `pom.xml` is
  • BUT ...
Maven and files

• By agreement, tests are supposed to create all files in “the project directory”

• Where’s that?
  • Unless you change it, ./target

• That means we have to work a little bit to keep files in the right place
Maven and files

- If your program changes its working directory, all bets are off
  - That won’t be an issue today
Groovy and files

• It’s not complicated
  • But you will have to put up with OO syntax

• new File("path/to/file").exists() == true

• new File("path/to/file").exists() == false
Groovy and files

• Remember, everything belongs in target

• `new File("target/...").exists() == true`

• `new File("target/...").exists() == false`
Groovy and files

• Create a file object
  • Unless all you’re doing is testing if the file exists

• File f = new File("target/…")
• f.exists() == true
Groovy and file size

• It’s complicated
  • There’s no “how many lines in the file”

• `File f = new File("target/...")`
• `f.size()`
  • Shows the number of bytes
  • Unless you’re testing for an empty file, this is A Bad Idea
Groovy and file size

• You have to read the lines into a list (array) then count the number of elements in the list
Groovy and file size

File f = new File("target/…")

List<String> got = []
f.eachLine { line ->
    got << line
}

got.size() == 3
Groovy and file size

- `got.size() == 3`
- `3? Um, no`

- Create a list with the lines that should be in the file
- Compare the size of the two lists
Groovy and file size

List<String> expected = [
    "Line 1",
    "Line two",
    "Line 3 3 3",
]

got.size() == expected.size()
File contents

• `got == expected`

• Really, that’s all there is to it
  • Almost makes up for the hassle with size and number of lines
What’s the Big Deal?

• So far, just simple tests
  • Exit value, stdout & stderr, files & file contents

• And very simple programs
  • “Sysadmin programs” are more complicated

• How does any of this help me?
“Sysadmin Program”? 

• Written to perform an operation related to the maintenance and operation of the computing infrastructure

• Typically parses the output of a “system program”
  • mount, df, ifconfig, du, etc.

• Performs an operation based on the results of the parsing
Sysadmin Programs

• Difficult or impossible to control the output of the system programs
  • May require running on multiple machines

• Operations may not be idempotent
• Worse, operations may disrupt service
  • Especially if not done correctly

• All this makes testing harder
  • So what’s a test-driven sysadmin to do?
“You mock me, sir”

- “Mocks,” also known as “stubs”

- A mechanism to return a pre-determined result
  - Possibly in response to input

- A mechanism for accepting and validating input
  - But performs no other actions
Mocks

• The idea is simple:

• Running /bin/mount or /bin/umount multiple times during a test will probably cause your system to crash

• So run $TESTBIN/mount and $TESTBIN/umount instead

• We’ll have to write those
Mock

• Let’s assume the simple case of just wanting particular output
  • No testing the input
  • No actions to be performed
Mocks

• It’s this easy

```bash
#!/bin/bash
cat $MOCK_MOUNT_FILE
exit $MOCK_MOUNT_EXIT
```
Mocks

• Put that in $TESTBIN/mount, chmod a+x
• Set $MOCK_MOUNT_FILE and $MOCK_MOUNT_EXIT
• Run your program

• Wait, where did $TESTBIN come from, and how do we get our program to run $TESTBIN/mount instead of /bin/mount?
Mocks

- In your program

    MOUNT=${TESTBIN:-/bin}/mount

- and

    $MOUNT args ...
Mocks

• What goes in your Spock/Groovy code

• Look at $HOME/lisa2015s7/mock.example
Mocks

• Lines 6 – 10: define some variables to make things easier to read

• Lines 14 – 21: create a wrapper that sets the necessary environment variables then runs the real program ("myprog")

• Line 31: delete the wrapper
Mocks

• In one test file, repeat lines 14 – 21, *mutatis mutandis*, for each test that needs a wrapper

• Change `$MOCK_MOUNT_FILE` and `$MOCK_MOUNT_EXIT` as needed
Mocks

• The mock mount command goes in src/test/resources/bin

• The data files have already been created for you

• They’re in src/test/resources/data
Exercise 4

STEP 1:
The program will parse the output of mount(1) and emit one or more starting points for back-ups. If the system uses only a single partition, no attempt to prune directories such as /tmp; if the system is built on multiple partitions, certain directories will be excluded from the list.

With no flags or if "-d" is given it will emit block device name (for example, /dev/sda7); the exit value will be zero (0).
Exercise 4

STEP 1 (cont’d):

Other flags will cause an error message of "myprog: unexpected arg '<char>'' and the exit value will be two (2).

No arguments are allowed; if any are given, an error message of "myprog: expected 0 args, got #" will be printed and the exit value will be two (2).
Exercise 4

• Tests to write:
  1. “myprog foo” exits with 2, gives correct error message
  2. “myprog –z” exits with 2, gives correct error message
  3. Using “mount.single”, “myprog” emits “/dev/sda1” and exits with 0
  4. Same as #3 except “myprog –d”
  5. Using “mount.separate”, “myprog” emits (on separate lines) “/dev/sda1”, “/dev/sda9”, “/dev/sda5”, and “/dev/sda6” (and exits with 0)
  6. Same as #5 except “myprog –d”
Exercise 4

• Data files are in ex04/src/test/resources/data
• There’s a mock mount command in ex04/src/test/resources/bin
• Don’t forget to create a wrapper script
  • Hint: read the mock mount command
Exercise 4

STEP 2:
If "-m" is given, the program will emit mount points (/, /usr, etc.) instead of block device names.

If both "-d" and "-m" are given, the system will emit the block device name then the mount point (one pair per line).

In either case, if no error occurred, the exit value will be zero (0).
Exercise 4

• New tests are similar to tests 3, 4, 5, and 6 (from Step 1) except they need to verify “myprog –m” and “myprog –d –m” with each of the two data files
That’s All, Folks!

• I think that’s everything

• If we have more time I’ll show you stuff on the fly

• Or we can go over anything you want

• And Q & A, of course
That’s All, Folks!

• Please fill out the evaluations
  • They really help me improve this class!

• Or send anonymous email with comments

• Come talk to me
  • Today or any time this week

• Send me email: asm2lisa2015s7@menlo.com
That’s All, Folks!

• Thanks for coming!

• I really hope you feel like you got your money’s worth

• Have a good conference

• Shirt made by kog 😊