testing

UVic SEng 265

Daniel M. German
Department of Computer Science
University of Victoria

October 8, 2002 Version: 1.00

Testing

- "program testing can be used to show the presence of bugs, but never their absence" E. Dijkstra
- Debugging is what you do when you know that a program is broken.
- Testing is a determined, systematic approach to break a program that you think is working
- It is theoretically impossible to verify the correctness of any program

Types of Testing

- Two main approaches: **white-box** and **black-box** testing
- **Black-box**: testing without relying (or knowing) the design or implementation
- **White-box**: testing using the information provided by the design and implementation
- White-box testing verifies the implementation.
- Black-box testing tests what the program is supposed to do.
- Both strategies should be combined

Test as you write code

- The earlier a bug is found the better
- Think systematically about what you are writing as you write it
- Verify simple properties of your program as you write it
- You can start testing before it is compiled

Test Code at Its Boundaries

- As you write each small piece of code is written, test it by probing its natural boundaries
- This code is supposed to copy an array, and replace the last element with “END”

```
my @other = @s;
$other[$#other] = "END";
```

- What are the errors?

Test Code Boundaries

- Check:
  - When the array is empty
  - When the array is exactly full
  - When the array is almost full

Test Code at Its Boundaries...

- This testing is useful to find off-by-one errors.
- It can become second nature,
- By **thinking**, as we write code, we can eliminate errors before they happen

Test pre and post conditions

- Verify the properties of data before (pre-condition) and after (post-condition) some piece of code
- Function to compute the average of an array of n numbers:

```
sub Average
{
    my (@array) = @_;
    my $total = 0;
    foreach my $a (@array) {
        $total += $a;
    }
    return $total / scalar (@array);
}
```

Assert conditions

- Best way to verify pre and post conditions

```
sub Average
{
    my (@array) = @_;
    my $total = 0;
    die "Illegal parameter. Array should be non-empty\n"
        if scalar (@array) == 0;
    foreach my $a (@array) {
        $total += $a;
    }
    return $total / scalar (@array);
}
```

- Very useful to verify parameters
- They pinpoint inconstencies between caller and callee
  - If assertion fails, the blame is on the caller

Protect your program for invalid use or data

- Add code that handles inconsistencies in the input
- For example, in `jukebox_??`
  - What happens if the input file does not exist?
  - WHat happens if some of the input records are invalid?
- Sometimes it is a good idea to protect against corruption created by the program itself
  - Out of range suscriptors
  - Division by zero...

Check Return Values for Errors

- It is extremely bad habit not to check error codes from functions
- You cannot assume that a function works as expected
  - file operations (open, print, printf, fwrite...)
  - =~ operator
  - =~ s// operator

Test as you program.. epilogue

- When you write the code, you understand it better
- Why wait until it breaks? Then you will not remember it
- Minimal effort and huge pay off

Systematic Testing

- You should test your programs systematically
- Do it orderly, so you don't overlook anything
- Keep records of what you have done

Test Incrementally

- Create and run tests as you create your program
- Do not wait until you finish
- Write some code, test, write more code, test
- Test and retest the same features as you add new ones
- Assignment 2
  - You complete simplest features, test it
  - You complete more features, test it and retest to part 1
  - You complete even more features, test it and retest to part 1 and 2
- You comment your code at the end it, test it!

Test Simple Parts First

- Test the simplest and most common parts first
- Then move on
- That way you build confidence on some parts of your code
- Easy testing finds the easy bugs
- Each new bug found is usually harder to find that its predecessor, but fixing it might not be harder

Know what output to expect

- You **must** understand and know the right answer for your tests
- If you don't, you are wasting your time
- This depends on the application domain on the program

Use tools to compare your output

- We should use diff extensively in this course to verify the output of programs

Measure Test Coverage

- One goal of testing is to make sure that every statement of a program is executed
- Testing cannot be considered complete unless every line of the program has been executed at least once
- But this is difficult to achieve
- It is difficult to find normal inputs that force a program to go through every statement
- 'Profilers' help you understand what parts of your program get executed.

Test Automation

- Several kinds of test coverage
  - Statement testing: has the statement been executed?
  - Branch testing: have all branches through a single-/two-way/multi-way conditional been taken?
  - Path testing: has this sequence of instructions been executed?
  - Definition-use path testing: has a specific path from a variable definition to that variable's use been executed?
  - All-uses testing: have all paths from a variable's definitions to all variable uses been executed?

Automate Regresion Testing

- Testing by hand: tedious and unreliable
- Proper testing involves
  - lots of tests
  - lots of inputs
  - lots of output comparisons
- worth the time to prepare a script
  - don't have to worry about being tired or careless
  - the easier the test, the more often you'll run if in the form of a script
- try to make test-script preparation a habit

Automate Regression Testing

- automate regression testing
- regression tests: a sequence of tests that compare the new version of something with the previous version
- intent: ensure behavior has not changed in new program except in expected ways

```
foreach $a in (1 2 3 4 5) {  # loop over test data files
   print ‘./old_version < input.$i >old.$i‘;  # run old version
   print ‘./new_version < input.$i >new.$i‘;  # run the new
   print ‘diff old.$i new.$i > diff.$i || echo "BAD OUTPUT"‘;
}
```


Automate Regression Testing...

- Test scripts should usually run silently
  - Only produce output if something unexpected occurs
  - Error output should be brief
- however, at times we may be concerned about infinite loops
  - print name of file being tested
  - eliminate this, though, when tests are running properly
- Big assumption: previous version of program computes the right answer
- For this to work: versions must be carefully checked “at the beginning of time”
  - If an error sneaks into a previous version, everything following will be invalid

Regression Testing...

- Programs may require hundreds of little tests
- Implementing them in an automated script makes it easy to do extensive testing after any change
- If you discover an error:
  - If not found by current tests, add the new test
  - And verify test by running on broken code
  - This may suggest further tests (or even a whole new set of possible errors)
- Never throw away a test! and keep records of bugs, changes and fixes (will help identify old problems)

Stages of Testing

- Unit Testing
- Integration Testing
- Functionality Testing
- Performance Testing
- Acceptance
- Installation

Unit Testing

- Each module/class is tested in a controlled environment
  controlled == under programmer's control
- Test team provides predetermined inputs
- Test team observes the outputs and other actions
- Test team checks:
  - Internal data structures
  - Program logic
  - Boundary conditions for input and output
- In our discussion of test so far, this is the stage on which we have focused

Tips for Testing

- Test team verifies that the system components work together according to the design documents
- Design takes place after gathering requirements and producing specifications focusing on the interfaces amongst units:
  - Are the function/methods signatures correct?
  - Are error values returned when expected?
  - Are they not returned when not expected?
  - Are there any conflicts amongst names? use of resources?
  - are units as robust together as they are apart?

Function Testing

- Evaluate whether the functions described by the customer's requirements are actually performed by the system
- does the program behave correctly?
- does it work well at the systems level?
  - Complete piece of software
  - Inputs are like those eventual users will give
- Not as much control as in the previous two stages (if there is an error, then it is harder to fix)

Performance Testing

- *first get it right, then make it fast*
- compare the system with the developer's specification
  - response time (interval between making a request and having it serviced)
  - throughput (number of requests serviced per second/minute, etc.)
- stress testing (peak loads)
- volume testing
- security testing
- timing/speed tests
- fault tolerance, error recovery

Acceptance Testing

- Customer and developer run through customer's requirements together
- Verify system meets customer's expectations
- Pilot testing
- Alpha test
  - Small number of developers
  - Usually on developer's site
- Beta test
  - Larger number of users
  - Usually at target site

Installation Testing

- Product is installed in the environment where it will be used
- Re-tested to verify it still works as desired
- Small changes in environment (between developer and customer sites) could be crucial
- Murphy's Law
- Afterwards, system is now in use customer now becomes the tester finds and reports bugs

Summary

- The better code you write, the less bugs (and the better paid you are)
- Do systematic and regression testing
- The single most important rule of testing: **do it**
- And remember, your reputation is on the line