Testing

Kurt Schmidt

Dept. of Computer Science, Drexel University

December 5, 2018
A QA engineer walks into a bar:
A QA engineer runs into a bar:
A QA engineer skips into a bar:
- She orders 1 beer
- She orders -1 beers
- He orders 99,999 beers
- It orders sfojbeerkiw
- He orders ’0 ; select id, password from users ; drop beers ; commit’

Opening day a customer walks in, asks for the bathroom. The bar catches fire, everybody dies.
Testing

Kurt Schmidt

Intro

Testing

Assertions

Debugging

Intro
Testing and Debugging

Testing  “...a determined, systematic attempt to break a program that you think is working.”
- Running a program with the intent of finding bugs
- “...testing can demonstrate the presence of bugs, but not their absence.”

Debugging  Finding the cause of an error, and fixing it.
Goals of Testing

- Discover bugs, *not* to show that program works
- Reduce the risk of failure to an acceptable level
- Designing a test before writing code is a great way to reduce bugs
Complete Testing

- Complete testing coverage is generally not possible
  - Or, not practical
- E.g., consider a program that takes an input of 10 characters
  - $2^{80}$ distinct inputs
  - At $1 \mu s$/test, would take more than twice the age of the universe
Testing Caveats

- People naturally assume what they do is correct
- People by nature overlook minor deficiencies in their own work
- Easy to overlook or ignore bad results
- Easy to choose only test cases that show the program “works”
- It’s useful to get another’s help
Software Testing Myths

- Really good programmers don’t have bugs
- Testing implies an admission of failure
- Testing is a punishment for our errors
- Testing can be avoided if we
  - Concentrate
  - Use OO methods
  - Use a good programming language
Humans make mistakes
  - Especially when creating complex artifacts

Even good programs have 1-3 bugs per 100 lines of code

People who claim they write bug-free code likely haven’t coded much
Defensive Programming

- Anticipate potential problems
- Design, code the system so problems are accounted for, or detected as early as possible

Defensive design  – Minimise confusion due to complexity
Defensive coding  – Take steps to localise problems
Defensive Design

- Simplicity of design
- Encapsulation
- Design with error in mind
- Prototype, walk-through
- **Make all assumptions and conditions explicitly**
Encapsulation

- Minimise coupling (dependencies) between objects\(^1\)
- Provide a sufficient interface
  - Hide all data behind the interface
  - Have enough functionality so that the client needn’t access the data directly

\(^1\)Loosely; i.e., a collection of related data, not necessarily an instance of a class
Designing With Error in Mind

- Error handling is often added as an afterthought
- Should be part of the interface
  - Decided before/as you code
- Ask “What if?” often
Design Reviews

- Show the design to another programmer, or lead
- Discussions and critiques are an excellent way to learn
- It is much easier to see another’s mistakes, and assumptions
  - Another pair of eyeballs is always helpful
Pre- and Post-Conditions

Pre-conditions  Assumptions that are made by a function or method upon entry, of necessity or efficiency
  - Units or state of arguments
  - State of the object
  - State of globals

Post-condition  Anything a function guarantees upon exit (if pre-conditions were met
  - Return value
  - Any side-effects
Class Invariants

- Conditions on an object which are always true
  - For a Vector $V$ of length $n$, elements $V_0$ through $V_{n-1}$ are valid
  - A Fraction might always be in lowest terms
- Conditions public (interface) functions can assume upon entry
- Must guarantee are true upon exit
  - Note, this does not necessarily apply to helper functions
Evolutionary Programming

- Compile and test your code often
- Implementation is bottom-up
  - Write small pieces
  - Test them
  - Gather into bigger pieces
- Evolve (and test) your program by writing modules, using stubs
Testing
General Testing Guidelines

- A necessary part of a test case is the expected output/behavior
- Test border conditions
- Test error conditions
- Check that the program doesn’t do what it shouldn’t (harder to impossible)
- Never be the last to test your own code
- Testing is an art; your skills will develop
Functional vs. Structural Testing

Functional (black box) testing:
- Implementation details are invisible
- Code is tested against its specs – look at advertised behavior
  - Inputs, outputs
  - Side effects
  - Error codes and exceptions
- Cases can be written *before* the code is written

Structural (clear box) testing:
- Tester must be familiar with the code
- Exercise all control paths, loop boundaries
- Tools such as *gcov* might be helpful
Integration, Unit, System, and Acceptance Testing

- These labels often get in the way of talking about testing\(^1\)
- I don’t want to confuse this discussion this term
- “Black-box” does *not* imply “unit” testing
- “Clear-box” does *not* imply “integration” testing

---
\(^1\)Mike Kelly, [searchsoftwarequality.techtarget.com](http://searchsoftwarequality.techtarget.com)
The *assert* statement

- Used during development, testing and debugging to test programmer’s assumptions
  - *Not* to be used in released code

```c
#include <assert.h>
...
assert( x!=0 ) ;
```

- If condition fails
  - Print condition
  - Print a line number
  - Dump the program

- To turn it off:

```
$ gcc -DNDEBUG ...
```
Debugging
Types of Bugs

- Syntactic
- Design
- Logic
  - Interface
  - Memory
How to Find a Problem

- **Think**
  - I’ll spend a few minutes with print/debug statements
  - I’ll move to a debugger before too long
  - If you reach an impasse, sleep on it
  - If you reach an impasse, show it to someone else
Why Use a Debugger?

- Debuggers are very powerful
- Desk checking code can be tedious and error-prone
- Print/Debug statements might require re-compilation
  - Error-prone
  - Leaves a mess to clean up
Basic Common Debugger Functions

- **Breakpoints**
  - Set at line or function
  - Can be conditional

- **Run program**
  - Execute line at a time
  - Run to next breakpoint
  - Move execution pointer around
  - Set command-line arguments
  - See stack trace

- **Data**
  - Evaluate any variable, expression
  - Modify values in memory
Think Before Repairing Errors

- Be careful you fix the problem
  - Don’t simply address the symptom
- Remember, don’t fix bad code, rewrite it!