What is software testing?
- Running a program in order to find bugs (faults, defects, errors, flaws, etc.).

What is not software testing?
- Static analysis: examining the source code to find bugs without executing the code.
- This is useful but technically not testing.

Why do testing?
- You need some tests to pass the code review.
- Your boss requires it.
- The product must adhere to specific safety standards.
- You want to prove your code correct.

Why is testing so hard?
- The number of possible input combinations is astronomical.
- How do you choose the input combinations to test?
- It requires extra time and effort.
Stages of testing

- Unit testing: the first phase done by module developers.
- Integration testing: combines the modules (that have already been thoroughly tested at the unit level) and tests interactions.
- System testing: tests the entire program.

Unit Testing

- Unit testing is more than just testing I/O relations.
- Unit tests should check intermediate computations, execution paths, etc.
- Three key components to a unit testing framework are:
  - Harnesses.
  - Stubs.
  - Instrumentation.
- Many frameworks exist to aid in writing unit tests.

Harnesses

- A framework used to run tests and analyze their results.
- Are often automatically generated.
- Include a suite of tests to run.
- Automatically generated tests are a start but should not be the end as they tend to be too generic.

Stubs

- Code that replaces the actual implementation.
- Some code may run very slowly, so using a stub is efficient.
- Some code may not yet be written, so using a stub is necessary.
- Some code may be complex, so using a stub is impractical.
Instrumentation

- Adding debug information into your code.
- For example: timing, resource usage, etc.
- Should be easily compiled in or out.
- Often the information is written to log files.
- Automated tools can be very useful here.
- Profilers fall into this category.

Regression Testing

- Changes to the code can and often break the code (i.e. the software regresses).
- To help prevent this a set of tests are run on the code to find errors introduced by code changes.
- This regression suite is automatically run periodically (at check-in, every night, etc.).
- Developers are notified when the regression suite uncovers a flaw in their code.
- Selecting a small but effective set of tests for the regression is the hard part.

How do we know if a test fails?

- The program should not crash (seg fault).
- Assertions should not be violated.
- Use a golden reference (it is automatable).
- Hand inspection.

Black-box Testing

- Testing does not look at the source code or internal structure of the program.
- Send the program a stream of inputs then observe the outputs.
- Abstracts away the internals which is useful for high-level testing.
White-box Testing

- Use the source code or code structure to design test cases.
- This leads us to the ideas of coverage.

Coverage Metrics I

- Used to measure the quality of the test suite.
- Helps answer questions like:
  - Do I need more tests?
  - What areas of the system are well tested?
  - Which tests should I write next?
- Coverage metrics are not perfect, but they are a useful guide.

Coverage Metrics II

- Achieving 100% coverage is not the goal.
- Finding bugs in the goal.
- 100% coverage does not prove that the program cannot fail.

Code Coverage Metrics

- Line: Is every line of code executed?
- Branch: Is the positive and negative of every branch taken?
- Expression: Are all legal expression values executed?
- Path: Are all paths in the CFG executed?
Mutation Testing

- A mutation of a program is a version of the program with one or more random changes.
- Mutation testing is another way to measure test suite quality.
- The test suite is run on the mutations.
- If these mutations are not found by the test suite there should be concern.
- Of course, it is not trivial to generate good mutants.

Faults, Errors, and Failures

- Fault: a static flaw in a program.
- Error: a bad program state that results from a fault.
- Failure: an observable incorrect behavior of a program as a result of an error.

Exposing a fault

- Reachability: the test must actually reach and execute the location of the fault.
- Infection: the fault must actually corrupt the program state.
- Propagation: the error must persist and cause incorrect output.
Test-driven Development

- One way to design for testability is to write the test cases before the code (extreme programming and agile development).
- This forces observability and controllability.
- Reduces the temptation to adjust tests to the program behavior.

Simulation and Stubbing

- A key to controllable code is simulation and stubbing.
  - Simulation
    - Usually done for hardware interfaces.
    - Real hardware may be slow, scarce, and hard to control.
  - Stubbing
    - Usually done for other code.
    - Not all code is complete, fast, or easy to control.

Assertions

- Assertions can improve observability by making errors into failures.
- Assertions can also improve observability by making the error rather than the failure visible.

Static Analysis

- Identifies potential bugs without actually executing the code.
- For example:
  - uninitialized variables.
  - Stack/buffer overflow.
  - Out of bounds array accesses.
  - Resource leaks.
- Historically static analysis has been plagued with excessive false positives.
- Lint was developed in the late 1970s.
- The ratio of reported to actual defects was often 10:1 or greater.
- Tools are improving, but scalability is still a problem.
Formal methods can prove programs correct.
In reality, they find a different set of bugs.
Formal tools are currently used for very specific tasks.
SPIN - detects deadlock in concurrent programs.
SLAM - checks that Windows Device Drivers meet common specifications.