Lecture 18: Dynamic analysis and testing I

CS 5150, Spring 2022
Administrative announcements

• Final presentation scheduling is happening now
• Feedback on Report #3 to be released at end of week
  • If you have not produced some working code by this point, you are likely behind
• Peer evaluations due this evening
  • Summary scores will be posted Thursday
• Next week is spring break
  • No client meetings
• Report #4
  • Required sections related to testing (details Thursday)
Lecture goals

• Justify uniformity of coding conventions and style
• Advocate for portability
• Write reliable, maintainable tests of various styles, scopes, and sizes
• Leverage dynamic analysis tools to find bugs
Style guides
Activity

• Brainstorm advantages of uniform style, universal rules

• Any disadvantages?
Style automation

**Advantages**

• Zero human effort
• Uniform enforcement
• Prevent accidentally misleading style
• Can be applied after refactoring, synthesizing code
• Can update entire codebase when style rules change

**Disadvantages**

• Can't reproduce all reasonable style rules
• Special-case exceptions are awkward
• Reformatting pollutes blame history
Style guide examples

• **Google: C++**
• MISRA C/C++
• **Google: Java**
• **Google: Python**

• Don't blindly adopt someone else's style guide – some justifications may not apply externally
  • But good to inherit from

• Elements of good style guides
  • Justify choices
    • Avoid danger
    • Enforce best practice
    • Ensure consistency
  • Avoid details that can be automated
  • Get developer buy-in
Portability

• Advantages
  • Enlarges customer base
  • Futureproofing
    • e.g. Apple Silicon
  • Reduces implicit assumptions
  • Improves process robustness
  • Expands tooling options
    • Compilers
    • Analysis tools
  • Educates team

• Anecdote: Every time I build a project with a new compiler, I discover bugs
  • Sometimes those bugs are in the compiler... but most are in the application

• Drawbacks
  • Maintenance burden
Portability targets

- Architecture
  - x86, ARM, 32 vs. 64-bit
- Operating system
  - Linux (Red Hat, Debian), Windows, Mac OS
  - Android, iOS
- Form factor
  - Smartphone, tablet, laptop, desktop, dual monitors
- Web browser
  - Chrome, Safari, Firefox

- C/C++ compilers
  - GCC, Clang, MSVC, Intel, Solaris Studio, IBM XL, PGI, SGI/Open64/PathScale
- Java virtual machines
  - Oracle/OpenJDK, IBM/OpenJ9, Azul
- Python interpreters
  - CPython, PyPy, Jython
Poll: PollEv.com/cs5150
Techniques to improve portability

- Heterogeneous developer environments
- Automated cross-platform builds and tests
  - Cloud infrastructure available
  - Don't ignore errors
- Highlight in style guides, code review checklists
- Use cross-platform standards and abstraction layers
  - Avoid writing your own #ifdefs unless portability is a business case

- Common gotchas:
  - Integer sizes
  - Filesystems
  - Unsupported APIs, language features
  - Floating-point behavior
  - Performance characteristics
  - Assumptions about unspecified behavior
    - Hyrum's Law
Testing
Goals of testing

- Find and prevent bugs
- Improve maintainability (esp. refactoring)
- Clarify intended usage

To meet these goals, tests themselves should be:
  - Bug-free
  - Maintainable
  - Clearly documented and easy to read
Test coverage

• Ways to measure "how much code" was tested
  • Function coverage
  • Statement (line) coverage
  • Branch coverage
  • Condition/decision coverage
  • Loop coverage
  • Path coverage
  • ...

• Coverage analysis can reveal gaps in testing

• Example:
  ```java
  if (a>b && c!=25) { d++; }
  ```

• Required cases for condition/decision coverage:
  • a<=b
  • a>b && c==25
  • a>b && c!=25
double[] boxFilter(double[] x) {
    var y = new double[x.length];
    for (int i = 0; i < x.length; ++i) {
        var xl = x[i]; var xr = x[i];
        if (i > 0) { xl = x[i-1]; }
        if (i < x.length-1) { xr = x[i+1]; }
        y[i] = (xl + x[i] + xr)/3.0;
    }
    return y;
}
Coverage targets

• *Any statement not covered by a test is code you expect your client/users to run before you do*

  • By this philosophy, 100% line coverage would be a minimum target
    • But chasing coverage metrics with low-quality tests can be self-defeating
  • Tests take time to write, review, and run; must consider cost/benefit ratio
Activity: Brainstorm difficult testing scenarios
Difficult testing scenarios

- Error codes & exceptions from library and system calls
  - Out of memory
  - Out of disk space
  - Incomplete I/O
  - Transient I/O error (EAGAIN)
  - Timeouts
- Unbounded blocking
- Crash/power loss
  - Corrupted data
- Malicious intent

- Concurrency
  - High lock contention
  - Race conditions
  - Caching & memory ordering
  - True concurrency vs. multitasking
- Portability
  - Unsupported capabilities
  - Platform differences
- Performance
  - NUMA
  - Big.LITTLE
  - Disk I/O (bandwidth, latency)
  - Network I/O (bandwidth, latency)
Beyoncé rule

• "If you liked it, then you shoulda put a test on it"

• Manages responsibility during large-scale refactoring
  • *Infrastructure team* must ensure all tests pass before committing
  • If functionality breaks, *product team* must fix it (and add more tests)

• Aim for sufficient coverage so that you (and your teammates) would be okay being held responsible for a production breakage in uncovered code
Example: SQLite

- 640x more test code than application code
- 100% branch test coverage
- OOM, I/O errors, crashes
  - Use abstractions to wrap malloc, I/O operations
- Boundary values
- Regression tests
- Valgrind
- Fuzz testing

- [https://www.sqlite.org/testing.html](https://www.sqlite.org/testing.html)
Kinds of testing

• Styles
  • Exploratory
  • Smoke tests
  • Black box
  • Glass box
  • Fuzz testing
  • Dynamic analysis

• Scopes
  • Unit tests
  • Integration tests
  • End-to-end tests

Can synthesize with boundary value analysis, coverage feedback

• Sizes
  • Small: fast, deterministic (in-process)
  • Medium: multi-process, allow blocking calls (single machine)
  • Large: Multi-node

• Purpose
  • Prevent reoccurrence of bugs (regression tests)
  • Prepare for release (acceptance tests, beta testing)
  • Ensure operating health (self tests)
Example: aerospace testing

- Unit tests
  - Ensure thorough coverage
  - Verify independent implementations

- Smoke tests
  - Small-scale integration test
  - Ensure configs are valid

- Regression tests
  - Catch any change to behavior (ensure refactoring changes are non-functional)
  - Ensure control algorithms achieve mission objectives

- Checkpoint/restore tests

- Exploratory tests
  - Logged data posted to reviews

- Software-in-the-loop
  - Medium-scale integration test
  - Leverage virtualization, preloading, hardware simulation
  -Subsystem and end-to-end scope

- Hardware-in-the-loop
  - Large-scale integration test
  - Verify non-functional requirements

- Vehicle-in-the-loop
  - Large-scale integration test
  - Verify a particular "production unit"

- Formal test deliverables