Testing Practice

CS195
Lecture 2

Software Development Today
Why do we have this structure?

Typical Scenario (1)

Typical Scenario (2)

Typical Scenario (3)

Typical Scenario (4)
Typical Scenario (5)

- Decision Maker
- Programmer
- Tester

"Oops, the world has changed. Here's the new spec."

"I'm done."

"Yes, it's done!"

Software Development Today

Why do we have this structure?

- Decision Maker
- Programmer
- Tester

Key Assumptions

- Development and testing must be independent
- Specifications must be explicit
- Specifications are always evolving
- All resources (including time) are finite
- Human organizations need decision makers
- Examine each of these separately

Independent Testing and Development

- Testing is basic to every engineering discipline
  - Design a drug
  - Manufacture an airplane
  - Etc.

- Why?
  - Because our ability to predict how our creations will behave is imperfect
  - We need to check our work, because we will make mistakes

Independent Testing and Development of Software

- In what way is software different?

- Folklore:
  - "Programmers are optimists"
  - The implication is that programmers make poor testers
  - Economics: "Programming costs more than testing"
  - The implication is that programming is a higher-skill profession

- How valid is the folklore, and how much is due to the current state of the art in testing?

Explicit Specifications

- Software involves multiple people
  - At least a programmer and a user
  - But usually multiple programmers, testers, etc.

- Any team effort requires mutual understanding of the goal
  - A specification
  - Otherwise, team members inevitably have different goals in mind
Specifications Change

- Why?

- Many software systems are truly "new"
  - Differ from all that went before in some way
  - Initial specification will change as problems are discovered and solved

- The world is changing
  - What people want
  - The components you build on (e.g., the OS version)

Software Specifications

- Software specifications are usually
  - In prose
  - Imprecise
  - Out of date

- Current state of specification is not conducive to automation
  - Not consumable by tools
  - Without a spec, there is nothing to check

Finite Resources

- Organizations make trade-offs
  - Not all goals can be achieved
  - Bounded resources are finite

- $S$ express relative costs among goals
  - Goals that are hard to quantify pose a problem
  - E.g., correctness, completeness

*We have 2 months, 5 programmers, and 2 testers. Here is a priority list of features. A feature is finished when it passed all of the tests for that feature. A programmer does not move on to a new feature until all higher priority features are finished or assigned to other programmers. We start now and ship whatever features are finished in 60 days.*

Reality

- Many proposals for improving software quality

- But the world tests
  - $\rightarrow$ 50% of the cost of software development

- Conclusion: Testing is important

The Rest of the Lecture

- Goal: Understand current state of practice
  - Boring
  - But necessary

Topics

- Purpose of testing
- Impact on design

- Widely-used practices
  - Manual testing
  - Automated testing
  - Regression testing
  - Nightly build
  - Code coverage
  - Bug trends
  - Stress testing

Testing frameworks
The Purpose of Testing

- Two purposes:
- Find bugs
  - Find important bugs
- Elucidate the specification

Example

- Test case
  - Add a child to Mary Brown's record

  - Version 1
    - Check that Ms. Brown's # of children is one more
  - Version 2
    - Also check Mr. Brown's # of children
  - Version 3
    - Check that no one else's child counts changed

Specifications

- Good testers clarify the specification
  - This is creative, hard work
- There is no hope tools will automate this
  - This part will stay hard work
- An extreme example
  - Warning: cheap laughs
  - "AOL spilled my coffee"

Manual Testing

- Test cases are lists of instructions
  - "test scripts"
- Someone manually executes the script
  - Do each action, step-by-step
    - Click an "login"
    - Enter username and password
    - Click "OK"
    - ...
  - And manually records results
- Low-tech, simple to implement

Manual Testing

- Those are the best reasons
- There are also not-so-good reasons
  - Not-so-good because innovation could remove them
  - Testers aren't skilled enough to handle automation
  - Automation tools are too hard to use
  - The cost of automating a test is 10x doing a manual test
Automated Testing

- Idea:
  - Record manual test
  - Play back on demand

- This doesn't work as well as expected
  - E.g., Some tests can't/shouldn't be automated

Fragility

- Test recording is usually very fragile
  - Breaks if environment changes anything
  - E.g., location, background color of textbox

- More generally, automation tools cannot generalize
  - They literally record exactly what happened
  - If anything changes, the test breaks

- A hidden strength of manual testing
  - Because people are doing the tests, ability to adapt tests to slightly modified situations is built-in

Breaking Tests

- When code evolves, tests break
  - E.g., change the name of a dialog box
  - Any test that depends on the name of that box breaks

- Maintaining tests is a lot of work
  - Broken tests must be fixed: this is expensive
  - Cost is proportional to the number of tests
  - Implies that more tests is not necessarily better

Improved Automated Testing

- Recorded tests are too low level
  - E.g., every test contains the name of the dialog box

- Need to abstract tests
  - Replace dialog box string by variable name X
  - Variable name X is maintained in one place
  - So that when the dialog box name changes, only X needs to be updated and all the tests work again

- This is just structured programming
  - Just as hard as any other system design
  - Really, a way of making the specification more concise

Data Driven Testing (for Web Applications)

- Build a database of event tuples
  < Document, Component, Action, Input, Result >

- E.g.,
  < LoginPage, Password, InputText, $password, "OK" >

- Complete system will have many relations
  - As complicated as any large database

- A test is a series of such events chained together

Discussion

- Testers have two jobs
  - Clarify the specification
  - Find (important) bugs

- Only the latter is subject to automation

- Helps explain why there is so much manual testing
Regression Testing

- Idea
  - When you find a bug,
  - Write a test that exhibits the bug,
  - And always run that test when the code changes,
  - So that the bug doesn’t reappear
- Without regression testing, it is surprising how often old bugs reoccur

Regression Testing (Cont.)

- Regression testing ensures forward progress
  - We never go back to old bugs
- Regression testing can be manual or automatic
  - Ideally, run regressions after every change
  - To detect problems as quickly as possible
- But, regression testing is expensive
  - Limits how often it can be run in practice
  - Reducing cost is a long-standing research problem

Regression Testing (Cont.)

- Note other tests (besides bug tests) can be checked for regression
- Ideally, entire suite of tests is rerun on a regular basis to assure old tests still work

Nightly Build

- Build and test the system regularly
  - Every night
- Why? Because it is easier to fix problems earlier
  - Easier to find the code after one change than after 1000
  - Avoid new code from building on the buggy code
- Test is usually subset of full regression test
  - “Smoke test”
  - Just make sure there is nothing horribly wrong

A Problem

- So far we have:
  - Measure changes regularly (nightly build)
  - Make monotonic progress (regression)
- How do we know when we are done?
  - Could keep going forever
- But, testing can only find bugs
  - Cannot prove their absence
  - We need a proxy for the absence of bugs

Code Coverage

- Idea
  - Code that has never been executed likely has bugs
- This leads to the notion of code coverage
  - Divide a program into units (e.g., statements)
  - Define the coverage of a test suite to be
  $$\frac{\text{# of statements executed by suite}}{\text{# of statements}}$$
**Code Coverage (Cont.)**

- Code coverage has proven value
  - It's a real metric, though far from perfect
- But 100% coverage does not mean no bugs
  - E.g., a bug visible after loop executes 1,025 times
- And 100% coverage is almost never achieved
  - Ships happen with < 60% coverage
  - High coverage may not even be desirable
  - May be better to devote more time to tricky parts with good coverage

---

**Using Code Coverage**

- Code coverage helps identify weak test suites
- Tricky bits with low coverage are a danger sign
- Areas with low coverage suggest something is missing in the test suite

---

**Example**

```c
status = perform_operation();
if (status == FATAL_ERROR)
    exit(3);

// Coverage says the exit is never taken
// Straightforward to fix
// Add a case with a fatal error
// But are there other error conditions that are not checked in the code?
```

---

**The Lesson**

- Code coverage can't complain about missing code
  - The case not handled
- But coverage can hint at missing cases
  - Areas of poor coverage ⇒ areas where not enough thought has been given to specification

---

**Bug Trends**

- Idea: Measure rate at which new bugs are found
  - Rational: When this flattens out it means
    1. The cost/bug found is increasing dramatically
    2. There aren't many bugs left to find
- Assumes testing resources well-deployed
  - We aren't overlooking any part of the code
- Assumes bugs can be fixed
  - Cautionary tale of OS/360

---

**The Big Picture**

- Testing practice has grown by trial-and-error
  - Many, many errors
- Standard practice
  - Measure progress often (nightly builds)
  - Make forward progress (regression testing)
  - Stopping condition (coverage, bug trends)
**Stress Testing**

- Push system into extreme situations
  - And see if it still works...
- Stress
  - Performance
  - Feed data at very high, very low rates
  - Interfaces
  - Replace APIs with badly behaved stubs
  - Internal structures
  - Work for any size array? Try sizes 0 and 1.
  - Resources
  - Set memory artificially low.
  - Some for # of file descriptors, network connections, etc.

**Stress Testing (Cont.)**

- Stress testing will find many obscure bugs
  - Explores the corner cases of the design
- Some may not be worth fixing
  - Too unlikely in practice
- A corner case now is tomorrow's common case
  - Data rates, data sizes always increasing
  - Your software will be stressed

**Back to Design**

- Testing has a profound impact on design
  - Because some designs are easier to test
- Design software so it can be tested!
- Or at least avoid designing software that cannot be tested

**Principles of Testability**

- Avoid unpredictable results
  - No unnecessary non-deterministic behavior
- Each piece of functionality in one place only
  - If two pieces of code have overlapping functionality, one often ends up being the other
- Design in self-checking
  - At appropriate places have system check its own work
  - Asserts
  - May require adding some redundancy to the code

**Principles of Testability (Cont.)**

- Avoid system state
  - Retain nothing across units of work
  - A transaction, a session, etc.
  - System returned to well-known state after each task is complete
  - Easiest system to test
- Minimize interactions between features
  - Number of interactions can easily grow huge
  - Rich breeding ground for bugs
- Have a test interface

**Testing Frameworks**

- Key components of a test system are
  - Building the system
    - May build many different versions to test
  - Running the tests
  - Deciding whether tests passed/failed
    - Sometimes a non-trivial task
  - Reporting results
- Testing frameworks provide these functions
  - E.g., Tinderbox
Summary

- Testing requires a certain mindset
  - Want to break the code

- Good testing is hard work
  - Requires real insight into the nature of the system
  - Will help elucidate the spec