Announcements

- Weekly reports
- Last day to discuss design document with your project coordinator
  - Friday, February 10
- Sprint 1 planning document due
  - Monday, February 13
- Also when Sprint 1 begins

Lecture 13

- Black-box and white-box testing
- Common defects
- Formal test cases
- Testing strategies

Black-box testing

- Testers provide the system with inputs and observe outputs
- Cannot see:
  - Source code
  - Internal data
  - Design documentation

Equivalence classes

- Brute force, every input permutation testing is usually impossible
  - It’s also pointless
- Divide possible inputs into groups that are treated similarly by an algorithm
  - Called equivalence classes
- Tester only needs to run one test per equivalence class

Examples

- Input a month (1-12)
  - Equivalence classes are:
    - \([-\infty, 0]\), \([1..12]\), \([13..\infty]\)
- Valid input is one of ten strings representing a type of fuel
  - Eleven classes - one for each string, one representing all other strings
Combinations
- Combinatorial explosion usually means you cannot test every possible input value
  - E.g., 4 inputs, 5 values → $5^4 = 625$ combinations
- Try to at least run one test per equivalence class
- Test combinations where an input is likely to affect the interpretation of another

Boundaries
- A lot of errors in software occur at the boundaries of equivalence classes
- Test at the extremes of each class
- Example: input a month (1-12)
  - Test 0, 1, 12, and 13

White-box testing
- Also called “glass-box” or “structural” testing
- Testers have access to system internals
  - Design documents
  - Code
  - Run time data and tracing
- Individual coders often informally employ white-box testing

Flow graph
- Each branch in the code (e.g., if and while statements) creates a node in the graph
- Testing strategy has to reach a targeted coverage of statements and branches
  - Cover all possible paths (usually infeasible)
  - Cover all possible edges (most efficient)
  - Cover all possible nodes (simpler)

Common defects
- Incorrect logical conditions
- Wrongly formulated conditions governing loops or if-then-else statements
- Testing
  - Use equivalence classes and boundary testing
  - Consider each variable used as an input
- Infinite loop or recursion
  - Analyze what causes a repetitive action to be stopped
- Testing
  - Try to identify test cases that might be handled incorrectly

- Incorrect preconditions
  - Preconditions state what must be true before an algorithm should be executed
- Testing
  - Create test cases in which each precondition is unsatisfied

- Infinite loop or recursion
- Incorrect preconditions

- Testing
  - Try to identify test cases that might be handled incorrectly

- Incorrect handling of NULL
  - Programs dereference something without checking what it points to
- Testing
  - Develop tests that result in NULL conditions
  - Search for abnormalities

- Incorrect handling of NULL
- Testing
  - Develop tests that result in NULL conditions
  - Search for abnormalities

- Incorrect handling of NULL
- Testing
  - Develop tests that result in NULL conditions
  - Search for abnormalities

- Not handling singleton or non-singleton conditions
  - A singleton condition occurs when there is normally one of something
  - What happens if there is more?
  - Non-singleton condition is the inverse
- Testing
  - Make sure unusual cases are properly handled
  - Brainstorm to determine unusual conditions, create and run tests

- Not handling singleton or non-singleton conditions
- Testing
  - Make sure unusual cases are properly handled
  - Brainstorm to determine unusual conditions, create and run tests

- Off-by-one errors
  - Program inappropriately adds or subtracts one
  - Loops one too many or too few times
  - Incorrect array indices
  - Particularly common defect
- Testing
  - Try to exercise boundary conditions
  - Develop tests that verify the program computes correct numerical answers, number of iterations

- Off-by-one errors
- Testing
  - Try to exercise boundary conditions
  - Develop tests that verify the program computes correct numerical answers, number of iterations

- Defects in timing and coordination
  - Deadlock: two or more threads are stopped, waiting on the other to do something
  - Remember dining philosophers?
  - Livelock: similar, except threads continue executing without making any forward progress
Deadlock and livelock often occur in a non-deterministic fashion
- Difficult to reproduce
- Often most effective approach is to use inspection to detect defects
- Test cases:
  - Vary timing
  - Cause a large number of threads to be created
  - Deliberately deny resources

Critical races
- One thread experiences a failure because another interferes with the "normal" sequence of events
- Also difficult to test for, especially with a black-box approach
- Deliberately slowing down threads is one strategy
- Inspection

Semaphores and synchronization
- Races can be prevented through proper synchronization
  - Lock data that is manipulated by multiple threads
  - Semaphore
  - In Java, the synchronized keyword can be used
  - Ensures no other thread can access an object until the synchronized method terminates

Stress and unusual situations
- Insufficient throughput or response time on minimal configurations
- Perform testing using minimally configured platforms

- Incompatibility with specific configurations of hardware or software
- Testing
  - Deploy and execute system on a variety of platforms with different configurations

- Defects handling peak load or resource exhaustion
- System does not gracefully handle resource shortage
  - Memory, disk space, network bandwidth, permissions
- Testing
  - Limit resources, run many copies at the same time
- Incorrect resource management
  - Program acquires and fails to release resources
  - Testing
    - Run the program for a long time, in a way that requires high resource usage

- Crash recovery
  - How does the system deal with hardware failure
    - What if power is lost
    - What if some part of the OS fails?
  - Testing
    - Kill programs at various times during execution
    - Power off the system

### Formal test cases and test plans
- A test case is an explicit set of instructions designed to detect a particular class of defect in a software system
- Test cases can give rise to many individual tests
- Each test is a particular running of the test case on a particular version of the system

### Test plans
- A test plan is a document that contains a complete set of test cases for a system
  - Along with other information about the testing process
  - Standard piece of documentation in software engineering

### No test plan?
- Testing will be done ad-hoc
- Poor quality software will result
- Should be written long before testing starts
- Can begin developing a test plan once requirements have been written

### Formal test case
- Identification and classification
  - Each test case should have a number and descriptive title
  - System, subsystem, or module should be clearly indicated
  - Importance of test case should be indicated
- Instructions
  - Tell the tester exactly what to do
  - Should not have to refer to any other documentation
- Expected result
  - What should the system’s response be?
  - Tester reports failure if not met

- Levels of importance
  - Level 1 – Critical
    - Designed to verify the system runs and is safe
    - Testing is generally terminated upon failure
  - Level 2 – General
    - Verify day-to-day/common functions are correct
    - Testing of other system aspects may continue if failure

- Level 3 – detailed
  - Test requirements of lesser importance
  - System functions most of the time, but has not yet met quality objectives

---

Sample test cases

A. Identification and classification
Test Case 2001
System: SimpleChat  Phase: 2
Server startup check with default args
Severity: 1

B. Instructions
1. At the console, enter: java EchoServer

C. Expected Result:
1. Server reports that it is active and listening, displaying:
   Server listening for clients on port 5555
2. Console waits for input

A. Identification and classification
Test Case 2002
System: SimpleChat  Phase: 2
Client startup check without login
Severity: 1

B. Instructions
1. At the console, enter: java ClientConsole

C. Expected Result:
1. Client reports cannot connect without login, message:
   ERROR – No login ID specified. Connection aborted.
2. Client terminates

A. Identification and classification
Test Case 2007
System: SimpleChat  Phase: 2
Server termination command check
Severity: 2

B. Instructions
1. Start a server (see Test Case 2001, instruction 1) using default args
2. Type #quit into server console

C. Expected Result:
1. Server quits and terminates
### Enumerating attributes

- Test cases should test every aspect of the requirements
- Each detail in the requirements is called an attribute
- Good first step: enumerate the attributes
  - Circle all of the important points in the requirements document (product backlog)
- Often many implicit attributes

### Big Bang - All at once

- Each module is, hopefully, first tested in isolation
- The entire system is then assembled and tested as a unit
- These people dream a lot and also buy lots of lottery tickets

### Big Bang disadvantages

- Very low probability of working
- Don’t know if the interfaces are correct until late in the testing process
- Stubs and drivers are both needed to test the modules in isolation
  - If it even happens
- Very difficult to isolate defects

### Big Bang advantages

- Maybe you’ll win the lottery
- Or get hit by lightning...

### Software testing strategies

- Big Bang – All at once
- Top Down
- Bottom Up
- Sandwich

### Top down testing

- Only the top module is tested in isolation
- Program modules are merged from the top to the bottom
- As each new module is integrated, the entire system is tested
**Stubs**
- Stubs are needed to simulate missing lower level modules
- Pieces of code that have the same interface as the lower level module(s)
- Do not perform real computations or manipulate real data
- Writing stubs can be expensive in itself

**Top down improvement**
- Test each module in isolation with both drivers and stubs before integrating them
- Prioritize the integration
- Build a working skeleton from top to bottom, then add the “flesh”
- Closer to integration testing

**Top down advantages**
- User interface is tested early
- Opportunity to involve the customer in early testing of the product
- If done right, an early working prototype can be made available for product validation
- Is it the right product?

**Top down disadvantages**
- Bottom levels are rarely tested enough
- Testing time starts slowly and then grows at a rapid rate near the end
- Usually occurs at about the same time as the project’s money and time are running out

**Bottom up testing**
- Modules are merged and tested from the bottom to the top
- Only modules tested in isolation are the terminal modules
- Call no other modules
- Higher level modules are added and tested in combination with previously tested lower level modules

**Drivers**
- Drivers are required to simulate the missing calls from the higher level modules
- Simple programs designed specifically for testing that make calls to the lower layers
- Similar role to stubs in top down testing
- Can also be time consuming to write
**Bottom up improvement**

- Test each module in isolation with drivers and stubs before integrating
- Prioritize integration of the modules
  - Build a working skeleton from bottom to top, then add flesh.
  - Stub out missing modules

**Bottom up advantages**

- Lower level modules are fairly well tested
- Testing can proceed early
  - And in parallel
  - And on separate parts of the system
- Good for real-time systems where many of the high priority, high risk modules are terminal

**Bottom up disadvantages**

- Top level UI is tested last
  - Cost of fixing it may kill the project
- Top level is rarely tested enough
  - Product is shipped, then the user gets to complete the testing

**Sandwich testing**

- Test the top level module and the terminal modules in isolation
  - Work both ways, top down and bottom up
- Use a combination of stubs and drivers to simulate missing parts

**Sandwich improvement**

- Test each module in isolation before integrating it into the system

**Sandwich advantages**

- Tests the user interface and terminal modules fairly well
- Gets users and engineers involved early
  - Tends to reduce cost associated with product changes
Sandwich disadvantages

- I can’t think of any I could really justify
- Can you?

Integration tests

- Initially all programs to be integrated are first stubbed out
- Each routine is then unit tested
- In a predetermined order, a series of test programs are generated with the unit tested routines replacing the stubs

Unit tests

- Independent, short code fragments that test an interface or individual method
- Generally written by the person coding the module
- Should still be reviewed by at least one other team member
- Or a formal testing team

Regression testing

- Set or subset of all tests that are re-run after any change or commit
- Cover as much of the system as possible
- Prevent regressions – system reverting to an earlier, incorrect state

Testing and product phases

- Alpha
  - Early, barely complete version of a product
  - Small number of trusted users
    - Understand that there are (potentially major) bugs
- Beta
  - Product is complete, but not thoroughly tested
  - Regular users are recruited to test in a normal work environment
  - Understand software is still "low-quality"
- Release candidate (RC)
  - "Going silver"
  - Beta version that may be the final product, unless a significant bug is found
  - Feature freeze
- Production release
  - "Gold"
  - Stable release
  - Ready for general use