Lecture 25: Embedded Software Test

Seyed-Hosein Attarzadeh-Niaki

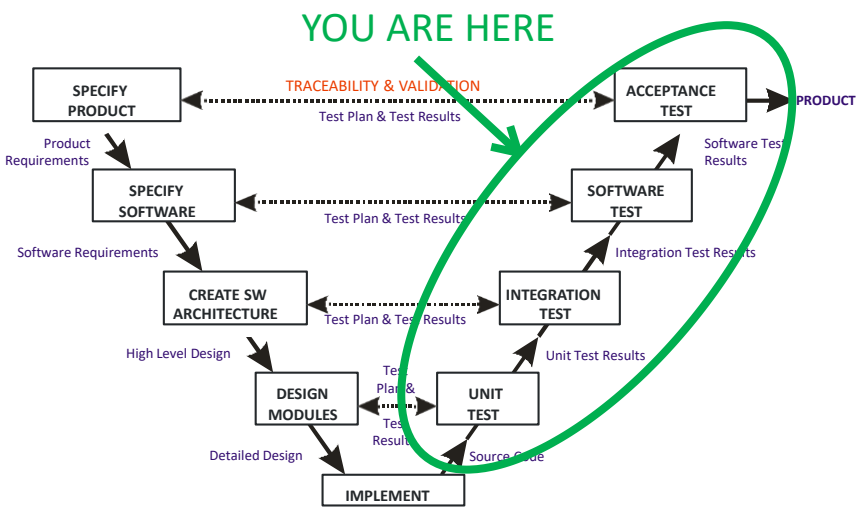
Based on slides by Philip Koopman

Review

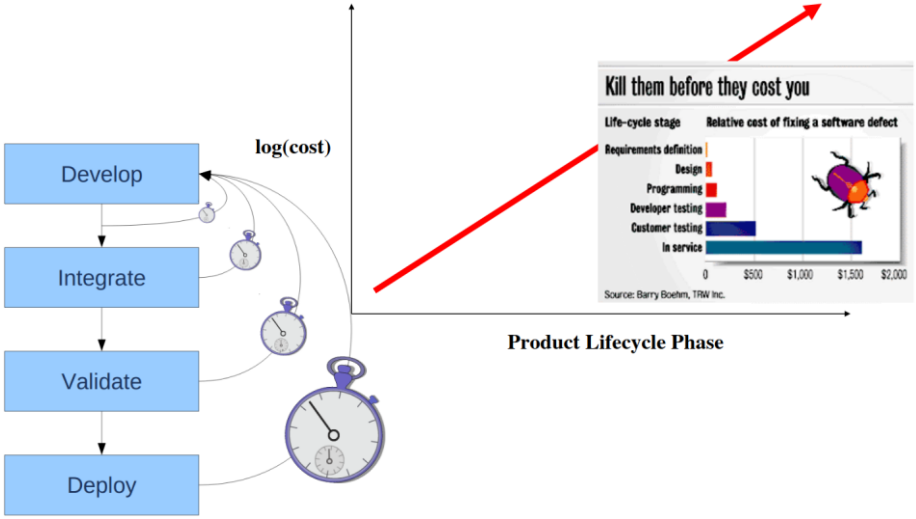
- Compilers for embedded systems
  - Energy-aware compilation
  - Memory-architecture aware compilation
  - Reconciling compilers and timing analysis

Outline

- Test for embedded software
- Types of testing
  - Testing styles
  - Testing situations
- Test coverage
- Model-based testing



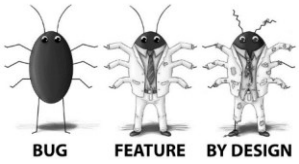
The Cost of Finding & Fixing A Defect In Different Stages



Definition Of Testing

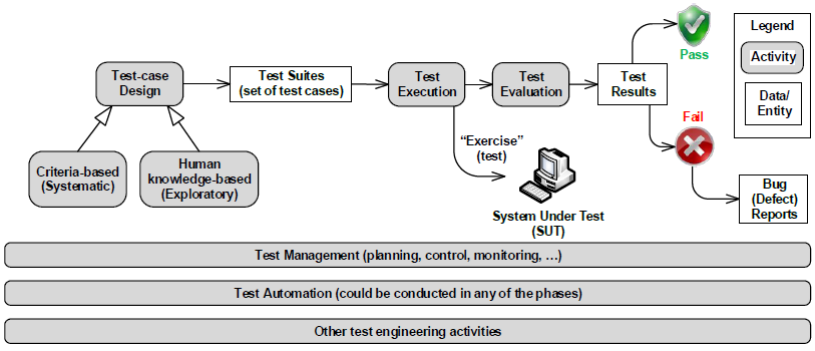
- Testing is performing *all* of the following.
  - Providing software with **inputs** (a "workload")
  - **Executing** a piece of software
  - **Monitoring** software state and/or outputs for expected properties, such as
    - Conformance to requirements
    - Preservation of invariants (e.g., never applies brakes and throttle together)
    - Match to expected output values
    - Lack of "surprises" such as system crashes or unspecified behaviors
- General idea is *attempting to find "bugs" by executing a program*
- The following are potentially useful techniques, but are **not** testing.
  - Verification methods such as model checking
  - Static analysis (lint; compiler error checking)
  - Design reviews of code
  - Traceability analysis

What's A Bug?



- Simplistic answer:
  - A "bug" is a software defect = incorrect software
  - A software defect is an instance in which the software violates the specification
- More realistic answer: a "bug" can be one or more of the following.
  - Failure to provide required behavior
  - Providing an incorrect behavior
  - Providing an undocumented behavior or behavior that is not required
  - Failure to conform to a design constraint (e.g., timing, safety invariant)
  - Omission or defect in requirements/specification
  - Instance in which software performs as designed, but it's the "wrong" outcome
  - Any "reasonable" complaint from a customer
  - … other variations on this theme …
- The goal of most testing is to attempt to find bugs in this expanded sense

A Generic Test Framework



Challenges in Embedded Software Test

- Embedded/cyber-physical systems integrated into a physical environment may be safety-critical
  - Need to test extra-functional aspects (timing, etc.) also
  - Testing in real environment may be dangerous
- Many functionalities initiated/controlled by machine-interfaces (and not human-interfaces)
  - Need to emulate particular electrical patterns
- High level of hardware dependence
  - *Availability* of hardware in the test phase
  - The *range* of the hardware unit types
  - Target with *different architecture* than host (PC)
    - Endianness, Memory model, Hardware accelerators
  - Target has *limited resources* (Memory, Disk, Speed, etc.)
- Harder to reproduce defects
- Updates may be essential

What to Test?

- Testing Functional Properties
  - Test depends on the physical environment. E.g.:

```
int x, y, buffer[128];

buffer = read_accelerometer(); //read accelerometer

if (f(buffer))
  Code A; //non-buggy code fragment
else
  y = x/buffer[0]; //buggy code fragment
```

- Testing Extra-functional (Non-functional) Properties
  - Timing constraints
  - Energy constraints
  - Reliability constraints

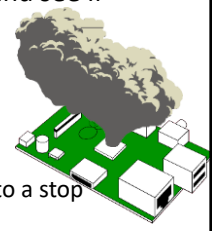
Types Of Testing

	Testing styles	Testing situations
Unstructured	• Smoke testing	• Unit test
Unstructured	• Exploratory testing	• Subsystem test
Methodical	• Black box testing	• System integration test
Methodical	• Grey box testing	• Regression test
Methodical	• White box testing	• Acceptance test
		• Beta test

- Most testing styles have some role to play in each testing situation

Smoke Testing

- Quick test to see if software is operational
  - Idea comes from hardware realm – turn power on and see if smoke pours out
  - Generally simple and easy to administer
  - Makes no attempt or claim of completeness
  - Smoke test for car: turn on ignition and check…
    - Engine idles without stalling
    - Can put into forward gear and move 5 feet, then brake to a stop
    - Wheels turn left and right while stopped



- Good for catching catastrophic errors
  - Especially after a new build or major change
  - Exercises any built-in internal diagnosis mechanisms
- But, *not usually a thorough test*
  - More a check that many software components are “alive”

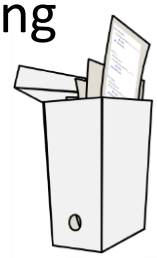
Exploratory Testing

- A person exercises the system, looking for unexpected results
  - Might or might **not** be using *documented system behavior* as a guide
  - Is especially looking for "strange" behaviors that are not specifically required nor prohibited by the requirements
- Advantages
  - An experienced, thoughtful tester can find many defects this way
  - Often, the defects found are ones that would have been missed by more rigid testing methods
- Disadvantages
  - Usually no documented measurement of coverage
  - Can leave big holes in coverage due to tester bias/blind spots
  - An inexperienced, non-thoughtful tester probably won't find the important bugs

Black Box Testing

- Tests designed with knowledge of behavior (Requirement)
  - But without knowledge of implementation
  - Often called "functional" testing
- Idea is to test **what** software does, but **not how** function is implemented
  - Example: cruise control black box test
    - Test operation at various speeds
    - Test operation at various underspeed/overspeed amounts
    - BUT, no knowledge of whether lookup table or control equation is used
- Advantages
  - Tests the *final behavior* of the software
  - Can be written *independent of software design*
    - Less likely to overlook same problems as design
  - Can be used to *test different implementations* with minimal changes
- Disadvantages
  - Doesn't necessarily know the *boundary cases*
    - For example, won't know to exercise every lookup table entry
  - Can be *difficult to cover* all portions of software implementation

White Box (Clear Box) Testing

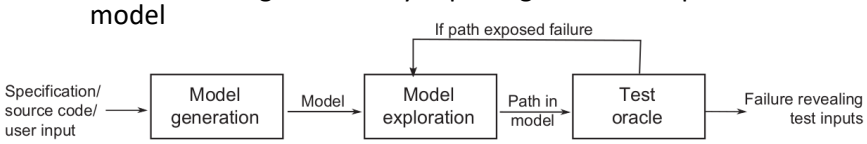


- Tests designed with knowledge of software design (Code)
  - Often called "structural" testing
- Idea is to exercise software, knowing how it is designed
  - Example: cruise control white box test
    - Test operation at every point in control loop lookup table
    - Tests that exercise both paths of every conditional branch statement

- Advantages
  - Usually helps getting *good coverage* (tests are specifically designed for coverage)
  - Good for ensuring *boundary cases* and *special cases* get tested
- Disadvantages
  - 100% coverage tests might not be good at assessing functionality for "surprise" behaviors and other testing goals
  - Tests based on design might *miss bigger picture* system problems
  - Tests need to be *changed if implementation/algorithm changes*
  - *Hard to test code that isn't there* (missing functionality) with white box testing

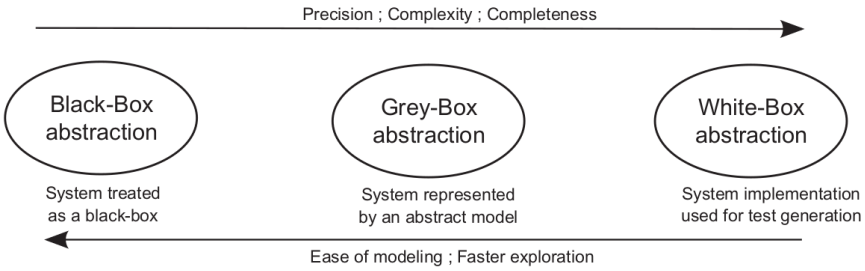
Grey Box Testing

- Tests designed based on a model of software
  - Abstract models capture only information related to properties of interest
  - Test cases are generated by exploring the search space of the model



- Examples of models
  - Timed State Machines
  - Markov Decision Process
  - Unified Modeling Language
  - Event Flow Graph

Comparison



Testing Coverage

- "Coverage" is a notion how completely testing has been done
  - Usually a percentage (e.g., "97% branch coverage")
- White box testing coverage (this is the usual use of the word "coverage")
  - Percent of conditional branches where both sides of branch have been tested
  - Percent of lookup table entries used in computations
  - *Doesn't test missing code* (e.g., missing exception handlers)
- Black box testing coverage
  - Percent of requirements tested
  - Percent of documented exceptions exercised by tests
    - But, must relate to externally visible behavior or environment, not code
  - Doesn't test extra behavior that isn't supposed to be there
- Important note: 100% coverage is not "100% tested"
  - Each coverage aspect is narrow; good coverage is necessary, but not sufficient to achieve good testing

Testing Is Typically Incomplete

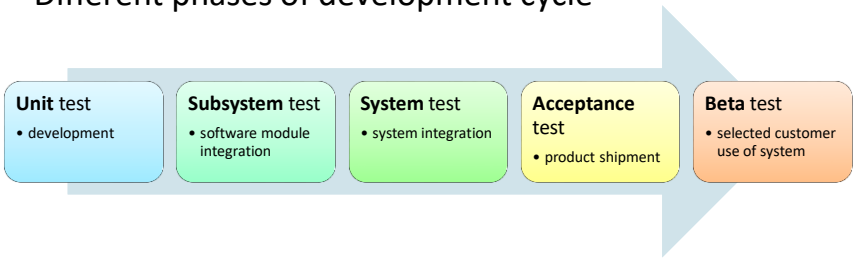
- It is impracticable to test everything at the system level
  - Too many possible operating conditions, timing sequences, system states
  - Too many possible faults, which might be intermittent
- Even unit testing is typically incomplete
  - How many test cases to completely test the following: `myfunction(int a, int b, int c)`
  - Assume 32-bit integers…
    - $2^{32} * 2^{32} * 2^{32} = 2^{96} = 7.92 * 10^{28}$ => at 1 billion tests/second Testing all combinations takes 2,510,588,971,096 years
- Even if you could test "everything," there is more to "everything" than just all possible input values
  - Kaner has published a list of more than 100 types of coverage https://kaner.com/pdfs/negligence_and_testing_coverage.pdf

81. Recovery from every potential type of equipment failure. Full coverage includes each type of equipment, each driver, and each error state. For example, test the program's ability to recover from full disk errors on writable disks. Include floppies, hard drives, cartridge drives, optical drives, etc. Include the various connections to the drive, such as IDE, SCSI, MFM, parallel port, and serial connections, because these will probably involve different drivers.
82. Function equivalence. For each mathematical function, check the output against a known good implementation of the function in a different program. Complete coverage involves equivalence testing of all testable functions across all possible input values.
83. Zero handling. For each mathematical function, test when every input value, intermediate variable, or output variable is zero or near-zero. Look for severe rounding errors or divide-by-zero errors.
84. Accuracy of every graph, across the full range of graphable values. Include values that force shifts in the scale.
85. Accuracy of every report. Look at the correctness of every value, the formatting of every page, and the correctness of the selection of records used in each report.
86. Accuracy of every message.
87. Accuracy of every screen.
88. Accuracy of every word and illustration in the manual.
89. Accuracy of every fact or statement in every data file provided with the product.
90. Accuracy of every word and illustration in the on-line help.
91. Every jump, search term, or other means of navigation through the on-line help.
92. Check for every type of virus / worm that could ship with the program.
93. Every possible kind of security violation of the program, or of the system while using the program.
94. Check for copyright permissions for every statement, picture, sound clip, or other creation provided …
95. Verification of the program against every program requirement and published specification.
96. Verification of the program against user scenarios. Use the program to do real tasks that are challenging and well-specified. For example, create key reports, pictures, page layouts, or other documents events to match ones that have been featured by competitive programs as interesting output or applications.
97. Verification against every regulation (IRS, SEC, FDA, etc.) that applies to the data or procedures of the program.

Absence of proof is not proof of absence.
– William Cowper

When Do We Test?

- Different phases of development cycle

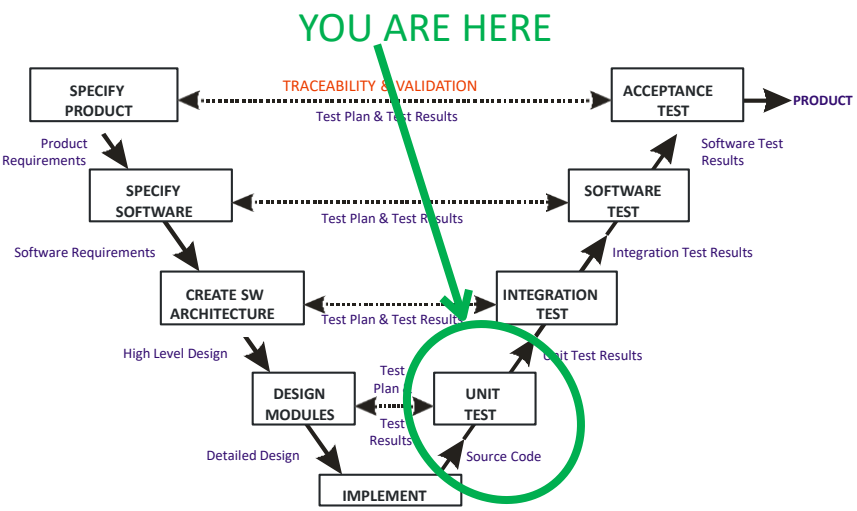


- Different people play roles of tester
  - Programmer often does own testing for unit test
  - Independent testers are often involved in subsystem, system & acceptance tests
  - Customers are often involved in acceptance & beta tests

Testing Across Development Cycle

- Testing within the V process
  - **Unit test**: single executable procedure
  - **Integration test**: do modules play nice?
  - **Software test**: exercise system or subsystem
  - **Acceptance test**: whole-product test with customer use scenarios
- Other types of testing
  - **Beta test**: see if representative users discover defects
  - **Regression test**: test to see if a previously fixed bug comes back
  - **Performance testing**: measure capacity and identify bottlenecks
  - **Robustness testing**: what happens with invalid inputs?
  - **Security testing**: can an attacker penetrate the system?
  - **Fault injection**: what happens when a component fails?

23



Embedded Real-Time Systems

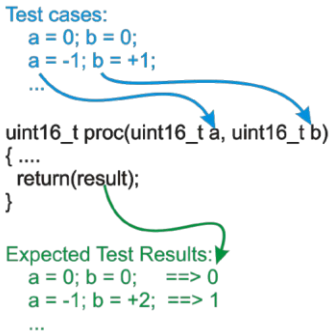
24

Unit Test

- A "unit" is a few lines of code to a small program
  - Usually created by a single developer
  - Usually tested by the programmer who created it
- **Purpose** of unit test
  - Try to find all the "obvious" defects
  - Can be done before and/or after code review
- **Approaches** (mostly exploratory & white box)
  - *Exploratory* testing makes a lot of sense
    - Helps programmer build intuition and understand code
  - *White box* testing to ensure all portions of code exercised
    - Often useful to ensure 100% arc and state coverage for statecharts
  - Some *black box* testing as a "sanity check"

Unit Testing

- Anti-Patterns
  - Only system testing
  - Testing only "happy paths"
  - Forgetting to test "missing" code

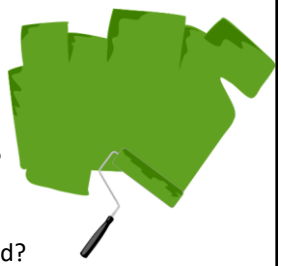


- Unit testing
  - Test a single subroutine/procedure/method
    - Use low level interface ("unit" = "code module")
  - Test both based on structure and on functionality
    - White box structural testing + Black box functional testing
  - This is the best way to catch corner-case bugs
    - If you can't exercise them here, you won't see them in system testing

Unit Testing Coverage

Coverage is a metric for how thorough testing is

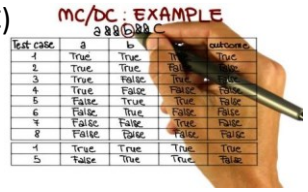
- Function coverage
  - What fraction of functions have been tested?
- Statement coverage
  - What fraction of code statements have been tested?
    - (Have you executed each line of code at least once?)
- Branch coverage (also Path Coverage)
  - Have both true and false branch paths been exercised?
  - Includes, e.g., testing the false path for if (x) { … }
- MCDC coverage (next slide)



- Getting to 100% coverage can be tricky
  - Error handlers for errors that aren't supposed to happen
  - Dead (unused) code that should be removed from source

MCDC Coverage as White Box Example

- Modified Condition/Decision Coverage (MC/DC)
  - Used by DO-178 for critical aviation software testing
  - Exercise all ways to reach all the code
    - Each entry and exit point is invoked
    - Each decision tries every possible outcome
    - Each condition in a decision generates all outcomes
    - Each condition in a decision is shown to independently affect the outcome of the decision



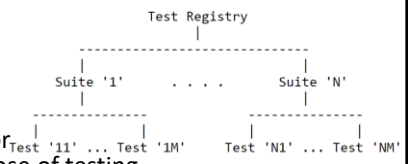
https://www.youtube.com/watch?v=DivaWCNohdw

  - For example: "if (A == 3 || B == 4)" ➔ you need to test at least
    - A == 3 ; B != 4 (A causes branch, not masked by B)
    - A !=3 ; B == 4 (B causes branch, not masked by A)
    - A !=3 ; B != 4 (Fall-through case)
    - A == 3 ; B == 4 is NOT tested because it's redundant (no new information gained)
  - Might need trial & error test creation to generate 100% MCDC coverage

Unit Testing Strategies

- Boundary tests
  - At borders of behavioral changes
  - At borders of min & max values, counter rollover
  - Time crossings: hours, days, years, …
- Exceptional values
  - NULL, NaN, Inf, null string, …
  - Undefined inputs, invalid inputs
  - Unusual events: leap year, DST change, …
- Justify your level of coverage
  - Trace to unit design
  - Get high code coverage
  - Define strategy for boundary & exception coverage

Unit Testing Frameworks



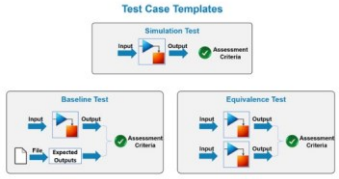
- Cunit as an example framework
  - Test Suite: set of related test cases
  - Test Case: A procedure that runs one or more executions of a module for purpose of testing
  - Assertion: A statement that determines if a test has passed or failed
- Test case example:

```
int maxi(int i1, int i2)
{ return (i1 > i2) ? i1 : i2; }
…
void test_maxi(void)
{ CU_ASSERT(maxi(0,2) == 2);// both a test case + assertion
CU_ASSERT(maxi(0,-2) == 0);
CU_ASSERT(maxi(2,2) == 2); }
```

Simulink Test

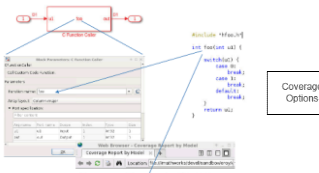
Provides tools for
- authoring,
- managing, and
- executing

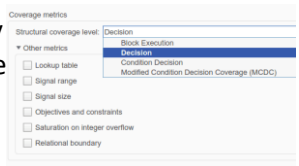
systematic, simulation-based tests of
- Models
- Generated code
- Simulated or physical hardware.

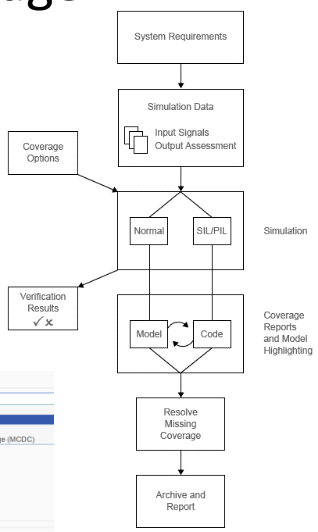


Simulink Coverage

- Model Coverage
- Code Coverage
- Coverage Metrics
  - Execution Coverage
  - Decision Coverage
  - Condition Coverage
  - Modified Condition/Decision Coverage (MCDC)
  - Cyclomatic Complexity
  - Lookup Table Coverage
  - Signal Range Coverage
  - …





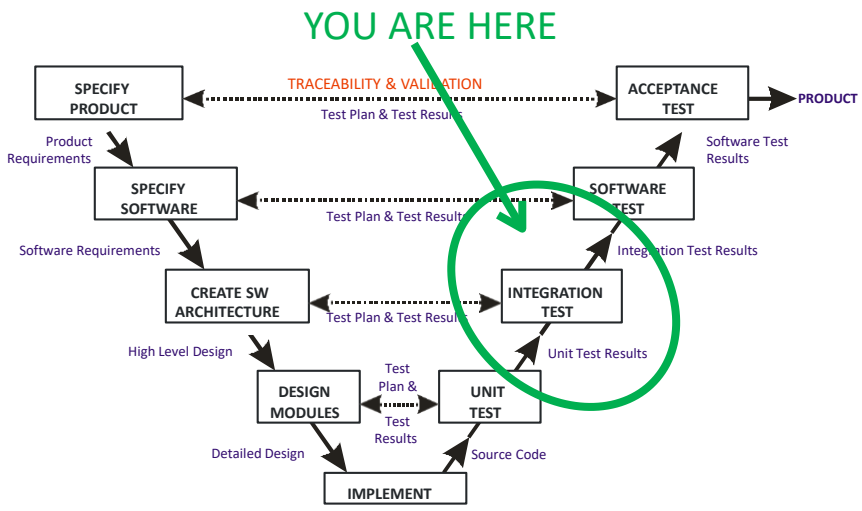


Best Practices For Unit Testing

- **Unit Test every module**
  - Use a unit testing framework
    - Don't let test case complexity get too high
  - Use combination of white box & black box
    - Get *good coverage*, ideally 100% coverage
  - Get good coverage of data values
    - Especially, validate all lookup table entries
- **Unit Testing Pitfalls**
  - Creating test cases is a development effort
    - Code quality for test cases matters; test cases can have bugs!
  - Difficult to test code can lead to dysfunctional "unit test" strategies
    - Breakpoint debugging is not an effective unit test strategy
    - Using Cunit/Unity/… to accomplish subsystem testing is not really unit testing
  - Pure white box testing doesn't test "missing" code

Subsystem Test

- A "subsystem" is a relatively complete software component (e.g., engine controller software)
  - Usually created by a team of developers
  - Usually tested by a combination of programmers and independent testers
- **Purpose** of subsystem test
  - Try to find all the "obvious" defects
  - Can be done before and/or after code review
- **Approaches** (mostly white box; some black box)
  - *White box* testing is key to ensuring good coverage
  - *Black box* testing should at a minimum check interface behaviors against specification to avoid system integration surprises
    - *Exploratory testing* can be helpful, but shouldn't find a lot of problems
    - *Smoke test* is helpful in making sure a change in one part of subsystem doesn't completely break the entire subsystem

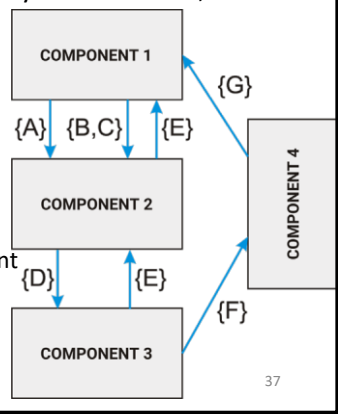


System Integration Test

- A "system" is a complete multi-component system (e.g., a car)
  - Often created by multiple teams organized in different groups
  - Usually tested by independent test organizations
- **Purpose** of system integration test
  - Assume that components are mostly correct; ensure system behaves correctly
  - Find problems/holes/gaps in interface specifications that cause system problems
  - Find unexpected behavior in system
- **Approaches** (mostly black box)
  - Tends to be mostly *black box* testing to ensure system meets requirements
  - Want *white box* techniques to ensure all aspects of interfaces are tested
  - *Exploratory testing* can help look for strange component interactions
  - *Smoke tests* are often used to find version mismatches and other problems in independent components

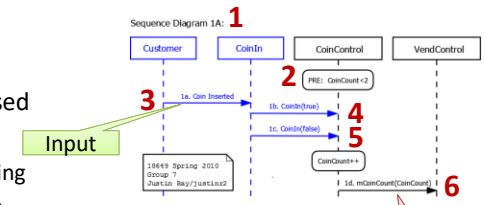
Integration Testing

- Anti-Patterns
  - Skipping unit test to do system test
  - No traceability from integration test to High Level Design
  - Integration test "pass" criterion based on system function, not interfaces
- Testing component integration
  - Exercise all component interfaces
    - Correct responses to input sequences?
    - Handle all types of data on interfaces?
  - Ensure modules match HLD and SDs
    - Assume unit test has vetted each component
    - Concentrate on component interactions



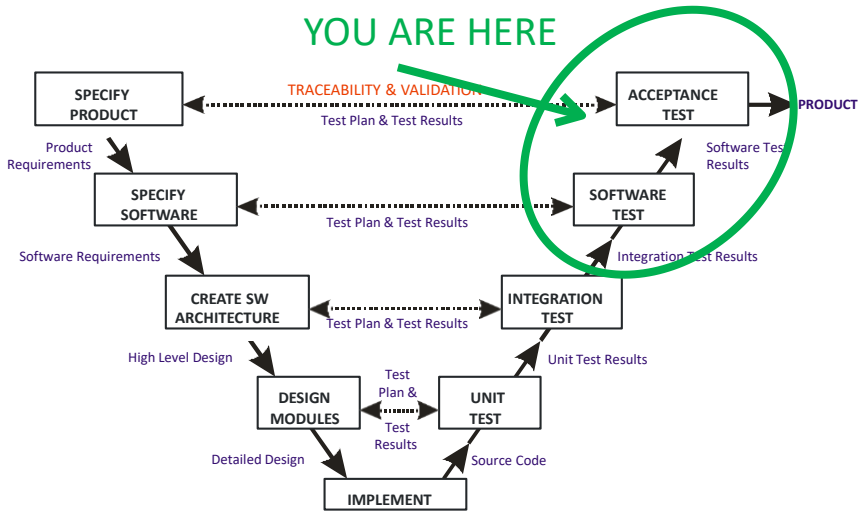
Integration Test Approaches

- Exercise all interfaces
  - All inputs result in correct outputs
  - Every component interface exercised
    - With all relevant values
    - With all relevant timing & sequencing
  - Use SDs and HLD info drive testing
    - Pass/fail: does it match SD?
- Integration test coverage:
  - All arcs on all SDs exercised?
  - Off-nominal behaviors tested?
    - Invalid sequencing and extraneous inputs?



Integration Test IT-1a:

1. Initialize modules
2. Test setup: CoinCount to zero
3. Insert coin (1a)
4. Observe CoinIn(true) (1b)
5. Observe CoinIn(false (1c)
6. Observe mCoinCount == 1 (1d)



System Level Testing

- Anti-Patterns
  - Excessive defect “escapes” to field testing
  - Majority of testing effort is ad hoc exploratory testing
  - Acceptance testing is the only testing done on system
- System test is last line of defense against shipping bugs
  - System-level “acceptance test” emphasizes customer-type usage
  - Software test emphasizes aspects not visible to customer
    - E.g., is the watchdog timer turned on and working?

Acceptance Test

- Acceptance tests ensure system provides all advertised functions
  - Testing performed by a customer or surrogate
  - Might also involve a certification authority or independent test observer
- Purpose of acceptance test
  - Does the system meet all requirements to be used by a customer?
  - Usually the last checkpoint before shipping a system
  - Might be performed on all system components to check for hardware defects/manufacturing defects, not just software design problems
  - In a mature software process, it is a quality check on the entire process
    - OUGHT to **find few or no** significant bugs if software has high quality
    - If bugs are found, it means you have a *quality problem*
- Approaches
  - Usually *black box* testing of system vs. 100% of high level product requirements

Regression Test

A. Regression tests ensure that a fixed bug stays fixed
   - Often based on test that was used to reproduce the bug before the fix

B. Regression tests ensure functionality not broken after a change
   - Often a subset of unit and integration tests (e.g., nightly build & test cycle)

- **Purpose** of regression testing
  - We made a change to the software; did we break anything?
  - In the case of iterated development, ensure there is a working system at the end of every periodic change/build/test cycle
  - It is a cheaper-to-run test than an acceptance test
- **Approaches**
  - Usually a *combination* of all types of tests that are sized to get decent coverage and reasonably fast execution speed.
  - Often concentrates on areas in which bugs have previously been found

Beta Test

- A "beta version" is complete software that is "close" to done
  - Theoretically all defects have been corrected or are at least known to developers
  - Idea is to give it to sample users and see if a huge problem emerges
- Purpose of beta test:
  - See if software is good enough for a friendly, small user community (limit risk)
  - Find out if "representative" users are likely to stimulate failure modes missed by testing
- Approaches
  - This is almost all *exploratory testing*
    - Assumption is that different users have different usage profiles
    - Hope is that if small user community doesn't find problems, there won't be many important problems that slip through into full production

➢ Defect in beta test means you have a significant hole somewhere
  - If you have a good process, it is likely a requirements issue
  - If it is "just a bug" then why did you miss it in reviews & testing???

Development vs. Test Effort

- Tester to Developer ratio varies depending on situation
- **Web development**: 1 tester per 5-10 developers
- **Microsoft**: 1 tester per 1 developer
- **Safety critical software**: up to 4-5 testers per 1 developer
  - "test hours" to "developer hours."
- This is in addition to
  - External acceptance testing (customer performs test)
  - Beta test
- How much does testing cost?
  - Total validation & verification (including testing) is often 50% of software cost
  - For "good" embedded systems, total test cost is about 60% of project
    - This includes time that developers spend on unit test & peer reviews
  - In critical systems, it can be 80% of total software cost!
    - (Take a look at the ratios above; this checks with them.)

Software Defect Issues

Pesticide paradox

- Tests will get rid of the bugs you test for, but not other bugs
- Thus, a program that passes 100% of automated tests is NOT bug-free!!!

Fixing a bug causes new bugs

- Fixing 1 bug will cause X other bugs
  - (X is "fault reinjection rate")
- Sometimes, fixing bugs is more expensive than writing new code, so
  - If X is medium to high (say, 0.2 to 1.0), then may be cheaper to write new code
  - If X is greater than 1, it is probably too risky to fix any more bugs

Bugs tend to congregate

- The more bugs you find in a piece of software, the more bugs are left
- It is common that some pieces of code are "bug farms" due to
  - Overly complex requirement for one piece of software
  - Poor quality implementation

Spaghetti Code

- Anti-Patterns
  - Deeply *nested conditionals*
  - "Switch" nesting
  - High Cyclomatic Complexity (too many paths through the code)

Hacker DICTIONARY

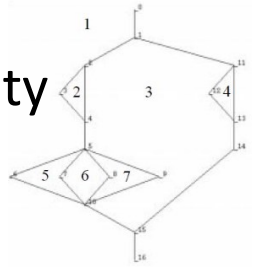
spaghetti code

n. Code with a complex and tangled control structure, esp. one using many GOTOs, exceptions, or other `unstructured' branching constructs. Pejorative. The synonym `kangaroo code' has been reported, doubtless because such code has many jumps in it.

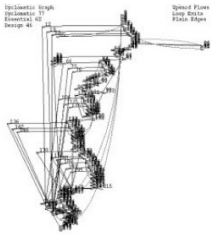
- Unstructured code leads to bugs
  - Unstructured code is generally hard to understand, test, and review
    - But, even structured code can be problematic if it is too complex
  - Want to limit complexity within each unit (e.g., subroutine, method)
    - Complex code is difficult to review – you will miss bugs during review
    - Complex code can be difficult or impossible to test

Measuring Complexity

- McCabe Cyclomatic Complexity (MCC)
  - Measure each module (subroutine, method, …)
  - Draw a control flow graph
    - Graph has an arc for each path through the module
  - MCC is # of "holes" in graph + 1
    - Worst case number of unit tests to cover all paths
    - Might need more tests – it's a guideline
- Strict Cyclomatic Complexity (SCC)
  - For complex "if" tests, each condition counts
  - "if ((x == 0) || (y == 0)) …"
    counts as +2, because need to test x!=0 and y!=0
  - MCDC testing requires this type of coverage



Complexity=7



Complexity=77

(NIST 500-235, 1996)

As the number of branches in the module or program rises, the cyclomatic complexity metric rises too. Empirically, numbers less than ten imply reasonable structure, numbers higher than 30 are of questionable structure. Very high cyclomatic numbers of more than 50 imply the application cannot be tested, while even higher numbers of more than 75 imply that every change may trigger a "bad fix". This metric is widely used for Quality Assurance and test planning purposes.

(RAC 1996, p.124)

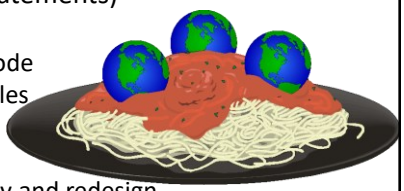
Global Variables Are Evil!

- Anti-Patterns
  - More than a few read/write globals
  - Globals shared between tasks/threads
  - Variables have larger scope than needed
- Global variables are visible everywhere
  - Use of globals indicates poor modularity
    - Globals are prone to tricky bugs and race conditions
  - Local static variables are best if you need persistence
    - File static variables can be OK if used properly
    - Don't make procedures globally visible if not needed

Spaghetti Factor (SF) Metric

SF = SCC + (Globals*5) + (SLOC/20)

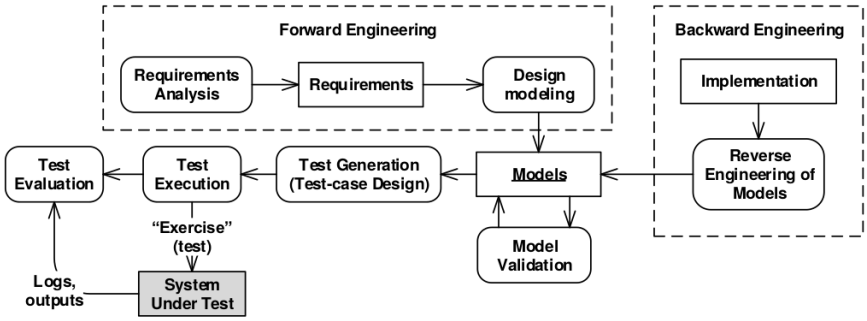
- SCC = Strict Cyclomatic Complexity
- Globals = # of read/write global variables referenced
- SLOC = # source lines of code (e.g., C statements)
- Scoring:
  - 5-10 - This is the sweet spot for most code
  - 15 - Don't go above this for most modules
  - 20 - Look closely; possibly refactor
  - 30 - Refactor the design
  - 50 - Untestable; throw the module away and redesign
  - 75 - Unmaintainable; throw the module and its design away; start over
  - 100 - Nightmare; throw it out and re-architect



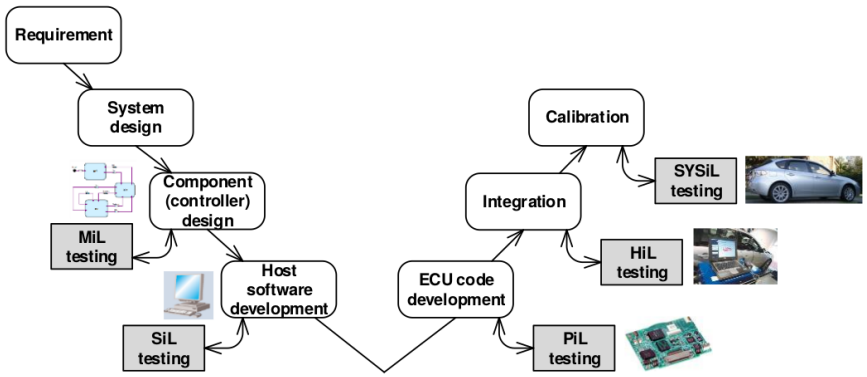
Bug Farms: Concentrations of Buggy Code

- 90/10 rule applied to bug farms
  - 90% of the bugs are in 10% of the modules
  - Those are the most complex modules
- Bug farms can be more than just bad code
  - Bad design that makes it tough to write code
  - Too complex to understand and test
  - Poorly defined, confusing interfaces
- Fixing bug farms
  - Refactor the module, redesign the interface
  - Often, smart to throw away and redesign that piece

Model-Based Testing



X-in-The-Loop Testing



X-in-The-Loop Testing

Phase/level	Entity under test	Testing interfaces
Model-in-the-Loop (MiL) testing	System model	Messages and events of the model
Software-in-the-Loop (SiL) testing	Control software (e.g., C or Java code)	Methods, procedures, parameters, and variables of the software
Processor-in-the-Loop (PiL) testing	Binary code on a host machine emulating the behavior of the target	Register values and memory contents of the emulator
Hardware-in-the-Loop (HiL) testing	Binary code on the target architecture	I/O pins of the target microcontroller or board
System-in-the-Loop (SYSiL) testing	Actual physical embedded system	Physical interfaces, buttons, switches, displays, etc.

Top 10 Risks of Poor Embedded Software Quality

10. Your module fails unit test (Tie with #9)
9. A bug is found in peer review (Tie with #10)
8. The system fails integration or software testing
7. The system fails acceptance testing
6. You get a field problem report
5. Your boss wakes you up at 2 AM because a Big Customer is off-line
4. You get an airplane ticket to a war zone to install a software update
3. You hear about the bug on social media
2. Your corporate lawyers ask you to testify in the lawsuits

And, the Number One Worst Way To Find A Bug:

1. The reporters camped outside your house ask you to comment on it