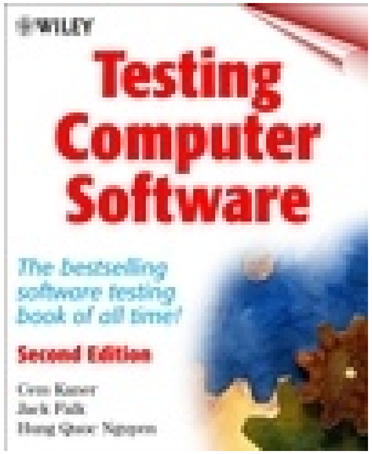
Software Testing

Suggested Reading

Testing Computer Software, Cem Kaner, Jack Falk, Hung Quoc Nguyen

- Used as framework for much of this lecture



Software Engineering: A Practitioner's Approach, Robert Pressman

- Chapters 17 & 18

The Art of Designing Embedded Systems, Jack Ganssle

- Chapter 2: Disciplined Development
- Chapter 3: Stop Writing Big Programs

The Mythical Man-Month, Frederick P. Brooks, Jr.

The Practice of Programming, Brian Kernighan & Rob Pike

Why Does Software Cost So Much? and Other Puzzles of the Information Age, Tom DeMarco

Overview

Big Picture

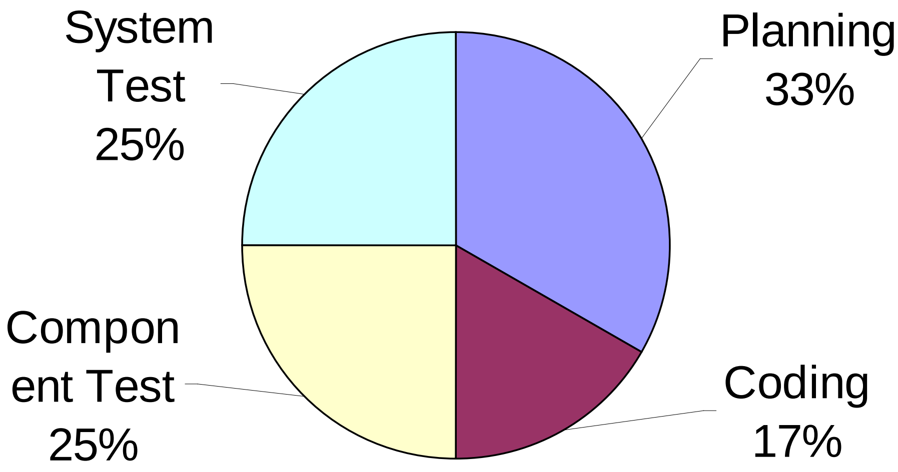
- What testing is and isn't
- When to test in the project development schedule
- Incremental vs. Big Bang

How to test

- Clear box vs. black box
- Writing test harnesses
  - Software only
  - Software and hardware
- Selecting test cases
  - What code to test
  - What data to provide

Testing

Brooks (MMM): Preferred time distribution – mostly planning and testing



The sooner you start coding, the longer it will take to finish the program

Philosophy of Testing

Common misconceptions

- "A program can be tested completely"
- "With this complete testing, we can ensure the program is correct"
- "Our mission as testers is to ensure the program is correct using complete testing"

Questions to be answered

- What is the point of testing?
- What distinguishes good testing from bad testing?
- How much testing is enough?
- How can you tell when you have done enough?

Clearing up the Misconceptions

Complete testing is impossible

- There are too many possible inputs
  - Valid inputs
  - Invalid inputs
  - Different timing on inputs
- There are too many possible control flow paths in the program
  - Conditionals, loops, switches, interrupts…
  - Combinatorial explosion
  - And you would need to retest after every bug fix
- Some design errors can't be found through testing
  - Specifications may be wrong
- You can't prove programs correct using logic
  - If the program completely matches the specification, the spec may still be wrong
- User interface (and design) issues are too complex

What is the Objective of Testing?

Testing IS NOT "the process of verifying the program works correctly"

- You can't verify the program works correctly
- The program doesn't work correctly (in all cases), and probably won't ever
  - Professional programmers have 1-3 bugs per 100 lines of code after it is "done"
- Testers shouldn't try to prove the program works
  - If you want and expect your program to work, you'll unconsciously miss failures
  - Human beings are inherently biased

The purpose of testing is to find problems

- Find as many problems as possible

The purpose of finding problems is to fix them

- Then fix the most important problems, as there isn't enough time to fix all of them
- The *Pareto Principle* defines "the vital few, the trivial many"
  - Bugs are uneven in frequency – a vital few contribute the majority of the program failures. Fix these first.

Software Development Stages and Testing

1. Planning
   - System goals: what it will do and why
   - Requirements: what must be done
   - Functional definition: list of features and functionality
   - *Testing during Planning: do these make sense?*
2. Design
   - External design: user's view of the system
     - User interface inputs and outputs; System behavior given inputs
   - Internal design: how the system will be implemented
     - Structural design: how work is divided among pieces of code
     - Data design: what data the code will work with (data structures)
     - Logic design: how the code will work (algorithms)
   - *Testing during Design*
     - *Does the design meet requirements?*
     - *Is the design complete? Does it specify how data is passed between modules, what to do in exceptional circumstances, and what starting states should be?*
     - *How well does the design support error handling? Are all remotely plausible errors handled? Are errors handled at the appropriate level in the design?*

Software Development Stages

3. Coding and Documentation
   - Good practices interleave documentation and testing with coding
     - Document the function as you write it, or once you finish it
     - Test the function as you build it. More on this later
4. Black Box Testing and Fixing
   - After coding is “finished” the testing group beats on the code, sends bug reports to developers. Repeat.
5. Post-Release Maintenance and Enhancement
   - 42% of total software development budget spent on user-requested enhancements
   - 25% adapting program to work with new hardware or other programs
   - 20% fixing errors
   - 6% fixing documentation
   - 4% improving performance

Development and Testing Approach: Incremental vs. Big Bang Testing

Incremental Testing

- Code a function and then test it (*module/unit/element testing*)
- Then test a few working functions together (*integration testing)*
  - Continue enlarging the scope of tests as you write new functions
- Incremental testing requires extra code for the *test harness*
  - A *driver* function calls the function to be tested
  - A *stub* function might be needed to simulate a function called by the function under test, and which returns or modifies data.
  - The test harness can *automate* the testing of individual functions to detect later bugs

Big Bang Testing

- Code up all of the functions to create the system
- Test the complete system
  - Plug and pray

Why Test Incrementally?

Finding out what failed is much easier

- With BB, since no function has been thoroughly tested, most probably have bugs
- Now the question is “Which bug in which module causes the failure I see?”
- Errors in one module can make it difficult to test another module
  - If the round-robin scheduler ISR doesn’t always run tasks when it should, it will be hard to debug your tasks!

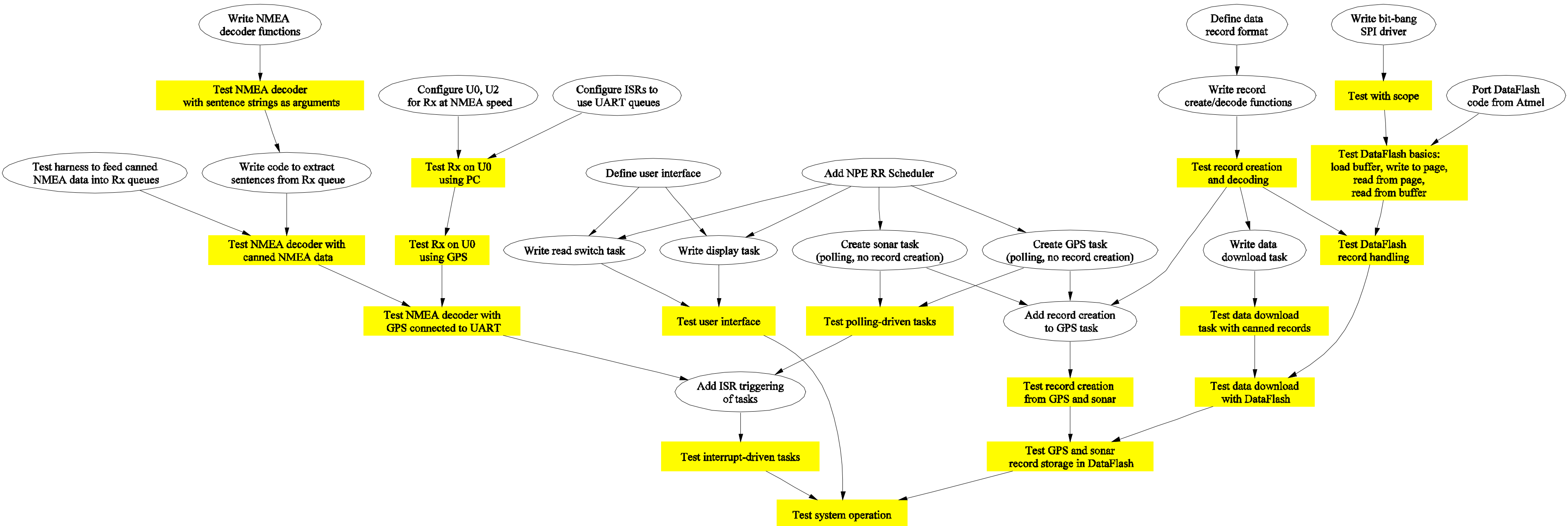
Less finger pointing = happier team

- It’s clear who made the mistake, and it’s clear who needs to fix it

Better automation

- Drivers and stubs initially require time to develop, but save time for future testing

Development Tasks



Mapper Development Plan

Development = Σ(coding + testing)

Task dependency graph shows an overview of the sequence of

- What software must be written
- When and how it is tested

Nodes represent work

- Ellipse = code, Box = test

Arrows indicate order

Overview

- Big Picture
  - What testing is and isn't
  - When to test in the project development schedule
  - Incremental vs. Big Bang
- How to test
  - Bug reports
  - Clear box vs. black box testing
  - Writing test harnesses
    - Software only
    - Software and hardware
  - Test plan and selecting test cases
    - What code to test
    - What data to provide

Bug Report

Goal: provide information to get bug fixed

- Explain how to reproduce the problem
- Analyze the error so it can be described in as few steps as possible
- Write report which is complete, easy to understand, and non-antagonistic

Sections

- Program version number
- Date of bug discovery
- Bug number
- Type: coding error, design issue, suggestion, documentation conflict, hardware problem, query
- Severity of bug: minor, serious, fatal
- Can you reproduce the bug?
- If so, describe how to reproduce it
- Optional suggested fix
- Problem summary (one or two lines)

Clear Box (White Box) Testing

How?

- Exercise code based on *knowledge of how program is written*
- Performed during Coding stage

Subcategories

- Condition Testing
  - Test a variation of each condition in a function
    - True/False condition requires two tests
    - Comparison condition requires three tests
      - » A>B? A < B, A == B, A > B
  - Compound conditions
    - E.g. (n>3) && (n != 343)
- Loop Testing
  - Ensure code works regardless of number of loop iterations
  - Design test cases so loop executes 0, 1 or maximum number of times
  - Loop nesting or dependence requires more cases

Black Box Testing

Complement to white box testing

Goal is to find

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavioral or performance errors
- Initialization and termination errors

Want each test to

- Reduce the number of additional tests needed for reasonable testing
- Tell us about presence or absence of a class of errors

Comparing Clear Box and Black Box Testing

Clear box

- We know what is inside the box, so we test to find internal components misbehaving
- Large number of possible paths through program makes it impossible to test every path
- Easiest to do *during development*

Black box, behavioral testing

- We know what output the box should provide based on given inputs, so we test for these outputs
- Performed *later in test process*

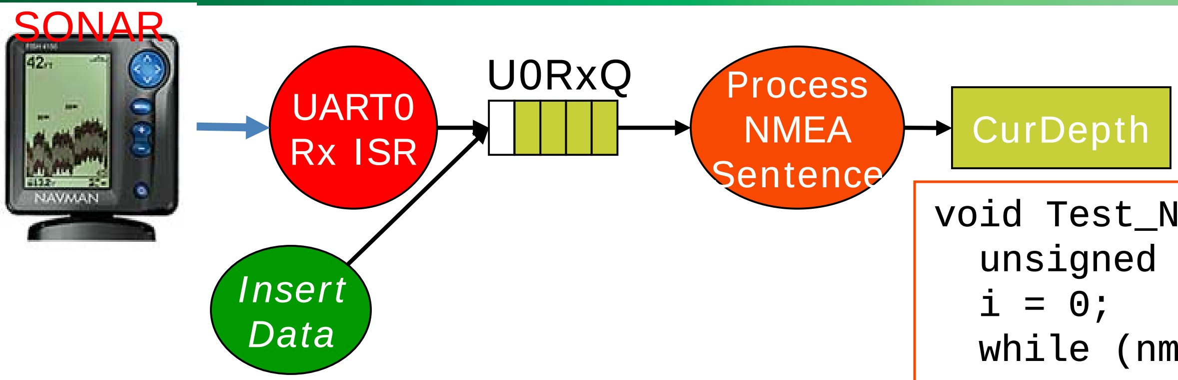
Test Harness

- Components
  - Driver: provide data to function under test
  - Stub: simulate an as-of-yet-unwritten function
    - May need stub functions to simulate hardware
- Conditional compilation
- Automation

```
#define TESTING 1
#define MIN_VAL (10)
#define MAX_VAL (205)

#if TESTING
  #define ADC_VAL ADC_Stub()
#else
  #define ADC_VAL adc2
#endif
```

```
int ADC_Stub(void) {
  static float i=0.0;
  i += 0.04;
  return 50*sin(i);
}

void Test_ADC_Clip(int num_tests){
  int n;
  while (num_tests--) {
    n = ADC_Clip();
    // verify result is valid
    if ((n<MIN_VAL)||(n>MAX_VAL))
       Signal_Test_Failure();
  }
}

int ADC_Clip(void) {
// read value from ADC ch 2 and
// clip it to be within range
  int v = ADC_VAL;
  v = (v>MAX_VAL)? MAX_VAL : v;
  v = (v<MIN_VAL)? MIN_VAL : v;
  return v;
}
```

Passing Input Data to Functions



Code gets data from…
- Arguments – easy to handle
- Global variables (including global data structures) – require some "glue" code to configure/preload

Example: Testing decoding of recorded NMEA sentences from sonar
- Don't have sonar connected to board
- Instead load U0RxQ with NMEA sentences

```
void Test_NMEA_Decoding(void) {
  unsigned int i;
  i = 0;
  while (nmea_sonar[i][0]) {
    Q_Enqueue_String(&SONAR_RX_Q,
      nmea_sonar[i]); /*  add string
      to queue */
    sonar_sentence_avail = 1;
    TASK_Process_NMEA_Sentence();
    i++;
  }
}
```

```
_far const char nmea_sonar[9][] = {
"$YXXDR,R,0.0,l,PORT FUEL,R,0.0,l,STARBOARD FUEL,U,12.4,V,BATTERY
*4F\r\n",
"$SDDBT,0.0,f,0.0,M,0.0,F*06\r\n",      "$SDDPT,0.0,0.0,2.0*57\r\n",
"$PTTKV,0.0,,,49.6,49.6,72.3,F*11\r\n", "$PTTKD,0.0,,B*1F\r\n",
"$VWVHW,,,,,0.0,N,0.0,K*4D\r\n",        "$VWMTW,22.4,C*16\r\n",
"$VWVLW,49.6,N,49.6,N*4C\r\n",          ""};}
```

Class Exercise: black box, driver testing

Test the function:

Function: ctof

Prototype: int ctof(int);

Works: Input a valid integer Celsius temperature, output will be a valid Fahrenheit temperature.

Assignment: Write a driver to test the function using black box testing

Your solution for black box testing

```
void main(void) {
   printf("input %d, output %d\n", -30000,
     ctof(-30000));
   printf("input %d, output %d\n", -274,
     ctof(-274));
   printf("input %d, output %d\n", -273,
     ctof(-273));
   printf("input %d, output %d\n", -40,
     ctof(-40));
   printf("input %d, output %d\n", 10000,
     ctof(10000));
   printf("input %d, output %d\n", 23,
     ctof(23));
}
```

Now imagine this code for ctof

```
int ctof (int tempin){
   if (tempin < -273) return (-32768);
   if (tempin > 18185) return (-32768);
   return ((tempin*9/5) +32);
}
```

Class Exercise: white box, driver testing

Test the function:

Function: ctof

Prototype: int ctof(int);

Works: Input a valid integer Celsius temperature, output will be a valid Fahrenheit temperature.

Assignment: Write a driver to test the function using white box testing

Sample solution for white box testing

```
void main(void) {
   printf("input %d, output %d\n", -32768,
     ctof(-32768)); //result -32768
   printf("input %d, output %d\n", -274,
     ctof(-274));   //result -32768
   printf("input %d, output %d\n", -273,
     ctof(-273));   //result -460
   printf("input %d, output %d\n", -40,
     ctof(-40));    //result -40
   printf("input %d, output %d\n", 18185,
     ctof(18185));  //result 32767
   printf("input %d, output %d\n", 18186,
     ctof(18185));  //result -32768
   printf("input %d, output %d\n", 32767,
     ctof(32767));  //result -32768
}
```

Code for ctof – it has a code error!!!!

```
return ((tempin*9/5) +32);
```

`tempin*9` could result in an integer (16-bit) overflow!

Can you instead divide by 5 first?

  Test with the number 15003:

  15003/5 = 3000, * 9 = 27000, +32 = 27032

But if you enter in 15003 it should yield a correct answer 27037.

Solution:

```
return (int(((long)tempin*9/5) +32));
```

Would your test have found the error?

Test Plans

A test plan is a general document describing the general test philosophy and procedure of testing. It will include:

Hardware/software dependencies

Test environments

Description of test phases and functionality tested each phase

List of test cases to be executed

Test success/failure criteria of the test phase

Personnel

Regression activities

Test Cases

A test case is a specific procedure of testing a particular requirement. It will include:

Identification of specific requirement tested

Test case success/failure criteria

Specific steps to execute test

Test Case Example

Test Case L04-007:

Objective: Tested Lab 4 requirement 007.

Passing Criteria: All characters typed are displayed on LCD and HyperTerminal window.

Materials needed: Standard Lab 4 setup (see test plan).

1. Attach RS-232c cable between the SKP board and a PC.
2. Start HyperTerminal on PC at 300 baud, 8 data bits, 2 stop bits, even parity.
3. Type “a” key on PC. Ensure it is displayed on SKP board LCD, and in the PC HyperTerminal window.
4. Test the following characters: CR, A, a, Z, z, !, \, 0, 9

A Good Test…

Has a high probability of finding an error

- Tester must have mental model of how software might fail
- Should test classes of failure

Is not redundant

- Testing time and resources are limited
- Each test should have a different purpose

Should be “best of breed”

- Within a set of possible tests, the test with the highest likelihood of finding a class of errors should be used

Should be neither too simple nor too complex

- Reduces possibility of one error masking another

Should test rarely used as well as common code

- Code which is not executed often is more likely to have bugs
- Tests for the common cases (e.g. everything normal) do not exercise error-handling code
- We want to ensure we test rare cases as well

Equivalence Partitioning

Divide input domain into data classes

Derive test cases from each class

Guidelines for class formation based on input condition

- Range: define one valid and two invalid equivalence classes
  - `if ((a>7) && (a<30))…`
  - Valid Equivalence Class: 7<x<30
  - Invalid Equivalence Class 1: x <= 7
  - Invalid Equivalence Class 2: x >= 30
- Specific value: one valid and two invalid equivalence classes
  - `if (a==20))…`
  - Valid Equivalence Class: x == 20
  - Invalid Equivalence Class 1: x < 20
  - Invalid Equivalence Class 2: x > 20
- Member of a set: one valid and one invalid equivalence classes
- Boolean: one valid and one invalid equivalence classes

Examples of Building Input Domains

Character strings representing integers

- Valid: *optional '–' followed by one or more decimal digits*
  - 5, 39, -13451235
- Invalid: *strings not matching description above*
  - 61-, 3-1, Five, 6 3, 65.1

Character strings representing floating point numbers

- Valid: *optional '–' followed by one or more decimal digits, optional '.' followed by one or more decimal digits*
  - 9.9, -3.14159265, 41
- Invalid: *strings not matching above description*
  - 3.8E14, frew, 11/41

Character strings representing latitude

- Valid:
  - *Degrees: integer string >= -180 and <= 180 followed by °*
  - *Minutes: floating point string >= 0.0 and < 60.0 followed by '*
  - *31° 15.90', 31° 15.90'*
- Invalid: *strings not matching description*
  - *310° 15.90', 1° -15', 30° 65.90'*

Regression Tests

A set of tests which the program has failed in the past

When we fix a bug, sometimes we'll fix it wrong or break something else

- Regression testing makes sure the rest of the program still works

Test sources

- Preplanned (e.g. equivalence class) tests
- Tests which revealed bugs
- Customer-reported bugs
- Lots of randomly generated data

Testability- How Easily Can A Program Be Tested?

How we design the software affects testability

- Operability – *The better it works, the more efficiently it can be tested.*
  - Bugs add overhead of analysis and reporting to testing.
  - No bugs block the execution of the tests.
  - The product evolves in functional stages (allowing concurrent testing)
- Observability – *What you see is what you test.*
  - A distinct output is generated for each input
  - System state and variables should be visible or queriable during execution (past states and variables too)
  - Incorrect output is easily identified
  - Internal errors are detected through self-testing, and are automatically reported
  - Source code is accessible

More Characteristics of Testability

- Controllability – *The better we can control the software, the more testing can be automated and optimized.*
  - All possible outputs can be generated through some combination of inputs
  - All code is executable through some combination of input
  - Software and hardware states can be controlled directly by the test engineer
  - Input and output formats are consistent and structured
  - Tests can be conveniently specified, automated and reproduced
- Decomposability – *By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting*
  - Software is built from independent modules
  - Modules can be tested independently
- Simplicity – *The less there is to test, the more quickly we can test it.*
  - Functional simplicity – no extra features beyond requirements
  - Structural simplicity – partition architecture to minimize the propagation of faults
  - Code simplicity – a coding standard is followed for ease of inspection and maintenance

More Characteristics of Testability

- Stability – *The fewer the changes, the fewer the disruptions to testing.*
  - Changes to software are infrequent and controlled
  - Changes to software do not invalidate existing tests
  - Software recovers well from failures
- Understandability – *The more information we have, the smarter we will test*
  - The design is well understood
  - Dependencies among components are well understood
  - Technical documentation is
    - Instantly accessible
    - Well organized
    - Specific, detailed and accurate