

Testing and Testability

- Good Test Cases
- Black-Box Testing
- White-Box Testing
 - rationale and approach
 - white box testing exercise
- code coverage
- complexity
- Exercise: test case enumeration

Good Test Cases

- Fundamental Characteristics
 - dispositive determines correctness
 - valid that answer is correct
 - deterministic yields consistent results
- Usability Characteristics
 - isolated independent test cases
 - automated runs w/o assistance
 - self-contained brings what it needs

Black Box Testing

- based on specified functionality
 - not based on any design knowledge
- does it perform all specified functions
 - with all specified options
 - and perform each of them correctly
- does it reasonably handle obvious errors
 - invalid requests from users/callers
 - anticipatable failures of underlying services
- common for acceptance criteria

improving Black-Box tests

- Specifications
 - tell us what functions to test, not what values
- boundary value analysis tries to choose
 - parameters near the edges of their range
- orthogonal array testing tries to choose
 - well distributed combinations of parameters
- these are all parameter choice heuristics
 - why not use our knowledge of the code to choose more probative parameter values?

Beyond Black-Box Testing

- Black-box testing can be reasonable
 - when output is defined as a function of inputs
- White box testing has a much farther reach
 - identify equivalence partitions of input combinations
 - enabling better parameter choice for black-box testing
 - code poorly exercised by primary interfaces
 - state results from combinations of operations
 - crucial interactions with other components
 - state ill-captured by return values and output
 - component depends on large/complex internal state
 - functionality not described by requirements
 - mechanisms defined by implementation strategy
 - areas of high perceived risk

Test Cases

- **name**
 - unique identifier for this test case
- **purpose**
 - component and functional area it tests
 - brief prose description of what assertion it tests
- **set-up**
 - brief description of pre-conditions for test case
- **operation**
 - operations to be performed (and how)
- **results**
 - what results we will capture (and how)
 - how we will use them to determine correctness

Exercise: Test cases

	Black Box	White Box
insertNode	team #1	team #3
deleteNode	team #2	team #4

- Break into your teams
 - examine your routine and methodology
 - black box: specifications and boundary values
 - white box: look for things that could go wrong
 - identify the required test cases
 - make lists
 - be prepared to justify necessity/sufficiency
- You will compare your results

Ordered Doubly-Linked Lists

```

typedef struct { node *prev; node *next; int data } node;
typedef struct { node *prev; node *next; } dbllist;

// insertion into an ordered doubly linked list
void insertNode(node *newNode, dbllist *list) {
    node *curr, *prev;
    prev = NULL;

    // scan for correct place to insert
    for ( curr = list->next;
          curr && curr->data < newNode->data;
          curr = curr->next )
        prev = curr;

    // update next pointer in pre node
    newNode->next = curr;
    newNode->prev = prev;
    if (prev == NULL)
        list->next = newNode;
    else
        prev->next = newNode;

    // update prev pointer in next node
    if (curr == NULL)
        list->prev = newNode;
    else
        curr->prev = newNode;
}

node *deleteNode(unsigned long value, dbllist *list) {
    node *prev, *curr, *next;

    // scan for the desired node
    for ( curr = list->next;
          curr && curr->data < value;
          curr = curr->next );

    if (curr == NULL || curr->data != value)
        return(0); // value may not be there

    // update next pointer in prev node
    if ((prev = curr->prev) != NULL)
        prev->next = curr;
    else
        list->next = curr;

    // update prev pointer in next node
    if ((next = curr->next) != NULL)
        next->prev = prev;
    else
        list->prev = prev;

    // return the removed node
    return( curr );
}
    
```

White-Box or Black-Box?

- purpose of testing is to gain confidence
 - that we have found all of our defects
 - that the component will function properly
- the question is ...
 - what set of tests will best give us confidence
 - not “which testing philosophy is best”
- they aren’t competing religions
 - they are approaches to test case definition
 - each with its own strengths and weaknesses
 - they may be highly complementary

Code Coverage

- the (too) simple goal
 - to be certain we’ve tested “all” the code
- how to measure code coverage
 - statically – simply by analyzing the code
 - run-time - with automatic instrumentation
- the process
 - identify yet unexecuted code segments
 - define test cases to exercise them
 - run them, verify both coverage and result

100% code coverage?

- 100% branch coverage may be too little
 - need all combinations of decisions
 - including a wide range of loop iterations
- 100% path coverage may be impossible
 - impossible condition combinations
 - errors that should never happen
- Advice
 - higher coverage is always a good thing
 - large numbers of paths may hide problems
 - supplement coverage with reviews

Code Complexity

- complex code is a problem
 - it is harder to design and implement
 - it is more likely to contain bugs
 - it requires more test cases
- we should be able to quantify complexity
 - best known metric is cyclomatic complexity
 - number of independent code paths
 - static call fan-out and depth
 - number of interfaces and parameters
 - inter-component coupling

static complexity analysis

- valuable as a basis for comparison
 - module A is much more complex than B
- limited use for estimating test cases
 - branch & code paths != execution-paths
- it ignores major sources of complexity
 - asynchronous interactions
 - thread serialization
 - fallibility of called services
 - coupling through dynamic data

Test Cases and Testability

13

what makes code “testable”

- observability
 - all interesting program behavior is visible
- controllability
 - all interesting program behavior is triggerable
- logical isolate-ability of functionality
 - so we can exercise one function at a time
- incremental construction
 - able to build and test from the earliest stages
- these result from architecture and design

Test Cases and Testability

14

Observeability

- What kinds of output are produced?
 - can they all be externally observed?
- Many are easily observed
 - return values
 - output text and dialogs
 - files and their contents
- Others are more difficult to observe
 - messages sent to other components
 - requests for system services

Test Cases and Testability

15

Controllability

- Many factors determine the code path
 - can they all be driven externally?
- Many can be driven very simply
 - scripted commands
 - prepared input files and databases
- Some inputs can be simulated
 - messages from other components
 - errors that are not easily caused
 - these simulations must be realistic?

Test Cases and Testability

16

For the next lecture

- McConnell 8: Defensive Programming
- Kampe: High Availability Taxonomy
- Kampe: Software High Availability

Supplementary Slides

Class Design

17

Test Cases and Testability

18

automation is essential

- tests must be run regularly
 - on each new version of the component
- tests must be run repeatably
 - the exact same tests run every single time
- results must be checked mechanically
 - tired/bored eyes can miss minor errors
- results must be summarized and reported
 - to measure improvement/regression
- these are repetitive, mind-numbing tasks

defining test cases

- start with specification-based test cases
 - these exercise the basic functionality
 - including specified error handling
- do operations naturally come in sequences
 - if so, we will also need use-case scenarios
 - we may want random scenario generation
- are there hidden internal mechanisms
 - not yet thoroughly exercised by the above
 - if so, design appropriate white-box test cases
- when we become confident, we can stop

Internal State & Methods

- internal state (in-memory databases)
 - how can we initialize these?
 - how can we observe changes to them?
- internal methods (that act on internal state)
 - how can we trigger them?
 - how can we observe their behavior?
- diagnostic options and operations
 - set or display internal state
 - initiate specific internal actions
- test harnesses and in-vitro testing
 - exercise components outside of the system