Testing and Testability

- Good Test Cases
- Black-Box Testing
- White-Box Testing
  - rationale and approach
  - white box testing exercise
  - code coverage
  - complexity
- Testability
  - Observability
  - Controlability

Good Test Cases

- Fundamental Characteristics
  - decisive: it provides a yes/no answer
  - valid: that determination is correct
  - repeatable: yields consistent results
- Usability Characteristics
  - isolated: it does not affect other test cases
  - complete: it is self-contained
  - automated: it runs w/o human assistance

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
  - known failures of underlying services
- common for acceptance criteria

improving Black-Box tests

- equivalence partitioning tries to choose
  - parameters from all specified classes
- 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
  - code poorly exercised by primary interfaces
  - state results from combinations of operations
  - interactions involving asynchronous events
  - operations affected by states of other components
  - state ill-captured by return values and output
  - component maintains large and complex internal state
  - errors in this state could easily go unnoticed
  - functionality not described by requirements
  - mechanisms defined by implementation strategy

Test Cases

- name: unique identifier for this test case
- purpose: brief prose description of what it tests
- set-up: brief description of pre-conditions for test
- operation: brief description of operation(s) to be performed
- results: brief description of how we will determine whether or not the component passed the test.
EXERCISE

```c
void insertValue(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 prev 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;
}
```

```c
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 );
}
```

Code Coverage

- the (too) simple goal
  - to ensure 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

what makes code “testable”

- incremental construction
  - able to build and test from the earliest stages
- controllability
  - all interesting program behavior is triggerable
- observability
  - all interesting program behavior is visible
- logical isolate-ability of functionality
  - so we can exercise one function at a time
- these result from architecture and design
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 data bases
• Some inputs can be simulated
  – messages from other components
  – errors that are not easily caused
  – these simulations must be realistic?

Observability

• 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

Internal State & Methods

• internal state (in-memory data-bases)
  – 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

For Next Lecture

• McConnell, chapter 29
  – overview of basic integration strategies
• Kampe, Integration Strategy
  – integration, architecture, testing, and schedule
• Kampe, Solaris Trainer Model
  – incremental integration for existing products
• Fowler, Continuous Integration
  – good advice for a more rational process
• Wikipedia, Software Testing
  – overview of types and approaches to testing
• Wikipedia, Test Driven Development
  – introduction to a useful agile development practice
• Kampe, Test Harnesses
  – introduction to a general class of testing tools

Supplementary Slides

自动化是非常重要的

• 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
White/Black - how to choose?

- 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”
- black/white aren’t competing philosophies
  - they are approaches to test case definition
  - each with its own strengths and weaknesses

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