Testing

Context: Software Quality Assurance
Design intent: Validation and Verification

Sources of flaws

For software: “flaw” = “defect” = “fault” = “bug”

• Flaws in understanding of intent
  • Validation – Does the system do the right thing?
  • Have we correctly analyzed the problem?

• Flaws in specification – incorrect capture of requirements
  • Intent
    • What the system does—its functionality
    • How the system accomplishes its task—performance
    • How the system responds to unexpected situations—robustness
    • What services are provided by system infrastructure—environment
      • Libraries, frameworks, hardware, etc.

• Flaws in realization of intent
  • Verification – Does the system do the thing right?
  • Have we correctly implemented a solution?

• Flaws in design and architecture
  • Incorrect high-level development decisions
• Flaws in code implementation
• Flaws in infrastructure implementation

Specifications in use
  Formal
  Declarations
  Stories
  Models
  Implicit
  Test cases
  Reference code
  etc

Primary focus of this course

Find that bug!

• A published (Mathematica) version of binary search:

```
BinarySearch[l_List, k_Integer, low_Integer, high_Integer, f_] :=
Module[{mid = Floor[(low + high)/2]},
  If[low > high, Return[low - 1/2]];
  If[f[l[[mid]]] == k, Return[mid]];
  If[f[l[[mid]]] > k,
    BinarySearch[l, k, 1, mid - 1, f],
    BinarySearch[l, k, mid + 1, high, f]]
```

This library code contains a flaw that was not detected for five years.
  • The first recursive call to BinarySearch should read:

```
BinarySearch[l, k, low, mid - 1, f]
```
  • Turns a logarithmic algorithm into a linear one

Moral: not all defects cause incorrect output
  • How would you test for this?
Software quality assurance and testing

- **A critical software engineering challenge**

- **Difficulties**
  - Expense: Testing and evaluation typically consume more time and cost in the software engineering process than design and code development
  - Typically 50% of total cost is attributable to quality assurance.
  - Precision: Almost impossible to completely succeed in testing and QA
    - "Very high quality" is rarely achieved, even for critical systems
  - Major gaps in testing and inspection
  - Consequential: The consequences (downside, upside) are considerable
    - NIST report: $60B lost
    - Developers: Holding back features and new capability

- **Trends**
  - There is rapid evolution in technology and practice (more later)
    - Important new techniques are emerging
      - Technical tools
      - Language
    - Engineering for "assurability" or "testability"
      - Requirements
      - Architecture, design, and other models
      - Implementation practices, languages, tools
    - Process
      - Metrics and measurement

Principal Evaluative Techniques

- **Testing**
  - Direct execution of code on test data in a controlled environment
    - Functional and performance attributes
      - Component-level
      - System-level
      - Identify and locate faults – no assurance of complete coverage

- **Inspection**
  - Human evaluation of code, design documents (specs and models)
    - Structural attributes
      - Design and architecture
      - Coding practices
      - Algorithms and design elements
    - Creation and codification of understanding

- **Static analysis**
  - Tool-supported direct static evaluation of formal software artifacts
    - Non-functional attributes
      - Null references
      - Unexpected exceptions
      - Memory usage
      - Can yield partial positive assurance

Other techniques
- Dynamic analysis
- Model checking
- Verification

17-654 Spring 2007 – Aldrich © 2007
Software quality methods – a survey

**Evaluative techniques**

- **Testing**
  - System and integration
  - Unit
  - Performance
  - Function
  - Usability

- **Test management**
  - Strategy
  - Coverage
  - Fault introduction
  - Resource management

- **Inspection**
  - Requirements
  - Design
  - Code

- **Static analysis**
  - Walk tree of code text
  - Follow control paths
  - Follow data paths

- **Dynamic analysis**
  - Monitoring and runtime checking
  - Instrumentation of code
  - Simulation of code

**Preventive techniques**

- **Requirements**
  - Quality stakeholders
  - Non-functional attributes

- **Process**
  - Measurement and feedback
  - CMM, TSP, etc.
  - Testers and their role
  - E.g., S&Es, agile
  - Risk mgmt

- **Architecture**
  - Robustness and self-healing

- **Design**
  - Robustness patterns
  - Safe APIs
  - Analysis

- **Coding**
  - Safe languages
  - Safe coding practices
  - Encapsulation / sandboxing

- **Specific practices**
  - Use of tools
  - Defect tracking
  - Root cause analysis

**Criteria** for evaluating techniques

- **Cost**
  - Ease of use
  - Resource requirements: people, time, computing

- **Timeliness: when we get answers**
  - E.g., unit test with scaffold
  - E.g., mock-up and prototyping

- **Accuracy**
  - False positives
  - False negatives

- **Development value**
  - E.g., Easier to modify code, add features
  - Risks of adoption

- **Metrics: observability of outcomes**

- **Scope: What kinds of defects it addresses**
  - System scale and complexity
  - Error vs. fault focus
  - Non-functional attributes: performance, usability, security, safety, etc.
  - Functionality

- **Integration and value during development**
  - Defect prevention support
  - Architecture design
  - Code management
  - Modeling and design intent capture
Faults, Errors, Failures, and Hazards

- **Fault**
  - **Type 1** – a flaw in an attached physical component
    - Traditional notion of a fault in hardware reliability theory (physical parts wearing out)
  - **Type 2** – a static flaw in software code
    - Syntactically local in code or structurally pervasive
    - Software faults cause errors only when triggered by use.

- **Error** – incorrect state at execution time caused by a fault
  - E.g., buffer overflow, race condition, deadlock, corrupted data

- **Failure** – effect of an error on system capability
  - E.g., program crashes, attacker gains control, program becomes unresponsive, incorrect output

- **Severity** – cost of failure to stakeholders
  - E.g., Loss of life, privacy compromise

- **Hazard** – product of failure probability and severity
  - Equivalent to risk exposure
Relating faults, errors, failures, hazards

- The unconstrained situation (black box)
  - Any fault can potentially lead to any failure
  - Dependability and security challenges cannot be prioritized

- Commitment to mission profile
  - Mission profile defines hazards
    - It determines relative priorities for action
    - Which faults/errors/failures have greatest hazard/risk?

- Commitments to design and structure
  - Design commitments constrain the mapping
    - An ideal design fully mitigates faults without affecting system functionality
  - Examples
    - Architecture and structure
      - E.g., self-healing, autonomic architectures
      - E.g., domain-specific architectures
    - Coding practice and patterns
      - E.g., state estimators, robustness tests, etc.
      - E.g., tools, patterns, anti-patterns
      - E.g., measurements of potential fault sites and mitigations
    - Tools and measurements

Fault Tolerance

- How does the system behave in the presence of errors in the environment

- Tolerating the faults of sensors, affectors, other physical components in a system
  - “Tolerating” means diminishing the likelihood or severity of failure in response to the fault

- Examples
  - Memory parity errors
  - Network transient faults
  - Sensor failures
  - Actuator anomalies
  - Processor transient faults
Robustness

• What happens when the system receives incorrect inputs?
  • Important for **security**
  • Examples:
    • Buffer under/overflow
    • Protocol violations
    • Null references
    • Precondition errors

• “Fault tolerance” with respect to **design faults** in other software components.

Robustness

• A range of possible responses
  • -1
  • Exception raised
  • Silent failure
  • Recoverable system crash
  • Unrecoverable (hard reboot) system crash
  • Corrupted local data
  • Corrupted database

• Examples of robustness failures
  • Windows device driver errors
  • API misuse by client code
  • Person-in-the-loop

• Testing for robustness
  • “Beyond the edge cases”
Buffer overflow errors / exploits

```c
#include
#define BUF_LEN 40
void main(int argc, char **argv)  {
   char buf[BUF_LEN];
   if (argv > 1)  {
      printf("buffer length: %d \n parameter length: %d", BUF_LEN, strlen(argv[1]));
      strcpy(buf, argv[1]);
   }
}
```

What are the results of an input that is too long?

```
% bad.exe AAAABBBBCCCC
```

```
% bad.exe AAAABBBBCCCCDDDEEEFFFFGGGGHHHHIIIJJJJKKKKLLLLMMMMM
```

Java buffer overflow

```java
public class Bad {
   static final int bufLen = 40;
   public static void main (String argv[]) {
      char[] buf = new char[bufLen];
      if (argv.length >0) {
         int len = argv[0].length();
         char[] tmp = argv[0].toCharArray();
         System.out.println("buffer length: " + bufLen + " parameter length: " + len);
         System.arraycopy(tmp, 0, buf, 0, len);
      }
   }
}
```

```
% java Bad AAAABBBBCCCCDDDEEEFFFFGGGGHHHHIIIJJJJKKKKLLLLMMMMM
```

```
buffer length: 40 parameter length: 52
java.lang.ArrayIndexOutOfBoundsException
at java.lang.System.arraycopy(Native Method)
at Bad.main(Bad.java:16)
Exception in thread "main"
```

```
Severity of failure is decreased relative to C
```
API and Framework Usage Errors

- Related to robustness errors

- **Aspects** of API and framework usage
  - Method call protocol (order of calls and state management)
  - Respect for callbacks (library is only caller)
  - Required set up and tear down
  - Aliasing (inappropriate object references)
  - Effects (inappropriate access or update to state)
  - Locking roles (e.g., caller must acquire a lock)
  - Use of threads (e.g., Java AWT)
  - Exception handling (e.g., correct handling of library exceptions)

- **Key point**
  - Many developers have difficulty in understanding and respecting the “bureaucracy” of an API

  - How to model?
  - How to assure?

```
int WINAPI WinMain(<<CUT>>)
{
    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow)) return FALSE;
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_HELLO);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
            TranslateMessage(&msg); DispatchMessage(&msg);
        }
    }
    <<CUT>>
}
```

Program runs fine
API: Hello World MFC: or 2, 1, 3

```c
int APIENTRY WinMain(<<CUT>>) {
    <<CUT>>
    // Perform application initialization:
    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_HELLO);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    <<CUT>>
}
```

Program runs fine

---

API: Hello World MFC: but not 2, 3, 1

```c
int APIENTRY WinMain(<<CUT>>) {
    <<CUT>>
    // Perform application initialization:
    hAccelTable = LoadAccelerators(hInstance);
    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    <<CUT>>
}
```

Lost process that must be killed from the Task Manager...
MFC Framework gives no compile or runtime help to identify this error
API: Hello World MFC: but not 2, 3, 1

- Could MFC detect the incorrect use of the API?
  - Instead of:
    ```
    hello.exe - 0 error(s), 0 warning(s)
    ```
  - We should get:
    ```
    Error: GetMessage() called before InitInstance() - Illegal API use
    hello.exe - 1 error(s), 0 warning(s)
    ```
  - Or at least:
    ```
    Warning: GetMessage() called before InitInstance() - API warning
    hello.exe - 0 error(s), 1 warning(s)
    ```

- Call order is a critical property for the API client
  - Typically verified by inspection
  - Model checking, typestate are emerging approaches

- A more robust design reduces severity of failure
  - In this case, by making it easier to identify the fix

---

API Bureaucracy and Usability

```java
public void focusLost(FocusEvent fevt) {
  <<CUT>>
  if (fieldName.equals(AGE)) {
    try {
      int i = Integer.parseInt(val);
      status.setText("Age is Valid.");
    }
    catch (NumberFormatException nfe) {
      // FAILED VALIDATION
      status.setText("Age is Invalid.");
      tf.requestFocus();
      status.setText("Failed Field Validation. Re-enter.");
    }
  }
  <<CUT>>
}
```

When a bad age is typed focus is not sent back to the “Age” field as the code requests. It goes to birthday.
API Bureaucracy and Usability

// get the focus back to try again
new FocusRequester(c); // c is the JTextField

class FocusRequester implements Runnable {
    private Component comp;
    public FocusRequester(Component comp) {
        this.comp = comp;
        try {
            SwingUtilities.invokeLater(this);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
    public void run() { comp.requestFocus(); }
}

Swing utility method invokeLater() used to “re-invoke” the age event correctly

Testing – The Big Questions

1. What is testing?
   • And why do we test?

2. What do we test?
   • Levels of structure: unit, integration, system...

3. How do we select a set of good tests?
   • Value-driven testing
   • Functional (black-box) testing
   • Structural (white-box) testing

4. How do we assess our test suites?
   • Coverage, Mutation, Capture/Recapture...

5. Practices for testability
   • What are known best test practices?
   • How does testing integrate into lifecycle and metrics?

6. What are the limits of testing?
   • What are complementary approaches?
     • Inspections
     • Static and dynamic analysis


- **What is testing?**
  - Direct execution of code on test data in a controlled environment

- **Goals of testing**
  - To reveal failures
    - Most important goal of testing
  - To assess quality
    - Difficult to quantify, but still important
  - To clarify the specification
    - Always test with respect to a spec
    - Testing shows inconsistency
      - Either spec or program could be wrong
  - To learn about program
    - How does it behave under various conditions?
    - Feedback to rest of team goes beyond bugs
  - To verify contract
    - Includes customer, legal, standards

---

**Testing is NOT to show correctness**

- **Theory:** "Complete testing" is impossible
  - For realistic programs there is always untested input
  - The program may fail on this input

- **Psychology:** Test to find bugs, not to show correctness
  - Showing correctness: you fail when program does
  - Psychology experiment
    - People look for blips on screen
    - They notice more if rewarded for finding blips than if penalized for giving false alarms
  - Testing for bugs is more successful than testing for correctness
    - [Teasley, Leventhal, Mynatt & Rohlman]
Testing – The Big Questions

1. What is testing?
   - And why do we test?

2. What do we test?
   - Levels of structure: unit, integration, system...

3. How do we select a set of good tests?
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing

4. How do we assess our test suites?
   - Coverage, Mutation, Capture/Recapture...

5. Practices for testability
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?

6. What are the limits of testing?
   - What are complementary approaches?
     - Inspections
     - Static and dynamic analysis

2. What do we test – the Focus of Concern

Examples of systems in context
- Mars rover
- Cell phone
- Clothes washing machine
- Point of sale system
- Telecom switch
- Software development tool
The Focus of Concern

Levels of Testing
- User testing, field testing
- System testing
  - With or without hardware
  - Integration testing
  - Unit testing

Unit Test and Scaffolding
Unit Test and Scaffolding

Examples: Database, Network, Interconnected systems

Back end services

Code to be tested

Example: Network client with GUI

Unit Test and Scaffolding

Code to be tested

Substitute: Test support code
Unit Test and Scaffolding

Code to be tested

Collaborating unit code
Unit Test and Scaffolding

Unit Code to be tested

Cluster

Unit Test and Scaffolding

Unit Code to be tested

Cluster

Stub

Driver
Unit Test for Components

- **Unit testing**
  - Separate testing of individual components of a system

- **Purpose**
  - Catch bugs early
  - Improve coverage
  - Validate internal interface/API designs
  - Catch bugs before code is written
    - Use test suites to guide implementation
    - Test components before client/service code is available

- **Done by development teams**
  - Original component developer
  - Tester / collaborator

- **Integration testing**
  - Test groups of interacting components
    - Exercise interactions among code components
    - Test in context of portions of the real environment
  - Same techniques and tools as for unit testing

Techniques for Unit Testing 1: Scaffolding

- Use "scaffold" to simulate external code

- **External code – scaffold points**
  1. Client code
  2. Underlying service code

1. **Client API**
   - Model the software client for the service being tested
   - Create a test driver
   - Object-oriented approach:
     - Test individual calls and sequences of calls

Testers write driver code
Techniques for Unit Testing 1: Scaffolding

- Use "scaffold" to simulate external code

- External code – scaffold points
  1. Client code
  2. Underlying service code

2. Service code
   - Underlying services
     - Communication services
     - Database queries and transactions
     - Network/web transactions
     - Device interfaces
     - Simulate device behavior and failure modes
   - File system
     - Create file data sets
     - Simulate file system corruption
   - Etc
   - Create a set of **stub** services or **mock** objects
     - *Minimal* representations of APIs for these services

Testers write stub code

Scaffolding

- **Purposes**
  - Catch bugs early
    - Before client code or services are available
  - Limit the scope of debugging
    - Localize errors
  - Improve coverage
    - System-level tests may only cover 70% of code [Massol]
    - Simulate unusual error conditions – test internal robustness
  - Validate internal interface/API designs
    - Simulate clients in advance of their development
    - Simulate services in advance of their development
  - Capture developer intent (in the absence of specification documentation)
    - A test suite formally captures elements of design intent
    - Developer documentation
  - Enable division of effort
    - Separate development / testing of service and client
  - Improve low-level design
    - Early attention to ability to test – "testability"
Testing – The Big Questions

1. What is testing?
   • And why do we test?

2. What do we test?
   • Levels of structure: unit, integration, system...

3. How do we select a set of good tests?
   • Value-driven testing
     • Functional (black-box) testing
     • Structural (white-box) testing

4. How do we assess our test suites?
   • Coverage, Mutation, Capture/Recapture...

5. Practices for testability
   • What are known best test practices?
     • How does testing integrate into lifecycle and metrics?

6. What are the limits of testing?
   • What are complementary approaches?
     • Inspections
     • Static and dynamic analysis

What makes a test case valuable?

• Value-driven testing
  • Focus on tests that have biggest benefit per unit cost

• Value is driven by quality improvement
  • Some value of information as well

• Value Factors
  • Does it find a bug?
  • How severe is the bug?
  • How common is the bug?
  • How easy is it to fix the bug?
  • Is it distinct from other tests?
    • Unique bug? Unique code? Unique domain coverage?
  • How general is it?
  • What did we learn about the program?

• Much of this is hard to predict in advance!
3. How do we select a set of good tests

**Test coverage**

- Why "coverage"?
  - All inputs cannot be tested.

- Consider strategy for testing these systems:
  - Visual Studio, Eclipse, etc.
  - Automotive navigation/communication system – with many configurations
  - An operating system
  - An e-commerce container framework (J2EE, .net) and its components

- Only very rarely can we test exhaustively.
  - Deterministic embedded controllers

**Test coverage – Ideal and Real**

- **An Ideal Test Suite**
  - Uncovers all errors in code
    - That are detectable through testing
  - Uncovers all errors in requirements capture
    - All scenarios covered
    - Non-functional attributes: performance, code safety, security, etc.
  - Minimum size and complexity
  - Uncovers errors early in the process
    - Ideally when code is being written ("test cases first")

- **A Real Test Suite**
  - Uncovers some portion of errors in code
  - Has errors of its own
  - Assists in exploratory testing for validation
  - Does not help very much with respect to non-functional attributes
  - Includes many regression tests
    - Inserted after errors are repaired to ensure they won't reappear
Ways of analyzing coverage

- **Code visibility – glass box or white box**
  - Visibility to internal code elements – better for non-functional attributes
  - Can use design information to guide creation and analysis of test suites
  - Can test internal elements directly
  - **Code coverage analysis**

- **Code visibility – black box**
  - Cannot see internal code elements of the service being tested
  - Test through the public API – better for functional attributes
  - **Domain coverage analysis**

### White Box: Statement Coverage

- **Statement coverage**
  - What portion of program statements (nodes) are touched by test cases

- **Advantages**
  - Test suite size linear in size of code
  - Coverage easily assessed

- **Issues**
  - Dead code is not reached
  - May require some sophistication to select input sets (McCabe basis paths)
  - Fault-tolerant error-handling code may be difficult to "touch"
  - Metric: Could create incentive to remove error handlers!
White Box: Branch Coverage

- **Branch coverage**
  - What portion of condition branches are covered by test cases?
  - *Or:* What portion of relational expressions and values are covered by test cases?
  - Condition testing (Tai)
  - **Multicondition coverage** – all boolean combinations of tests are covered

- **Advantages**
  - Test suite size and content derived from structure of boolean expressions
  - Coverage easily assessed

- **Issues**
  - Dead code is not reached
  - Fault-tolerant error-handling code may be difficult to “touch”

White Box: Path Coverage

- **Path coverage**
  - What portion of all possible paths through the program are covered by tests?
  - Loop testing: Consider representative and edge cases:
    - Zero, one, two iterations
    - If there is a bound $n$: $n-1$, $n$, $n+1$ iterations
    - Nested loops/conditionals from inside out

- **Advantages**
  - Better coverage of logical flows

- **Disadvantages**
  - Not all paths are possible, or necessary
    - What are the *significant* paths?
  - Combinatorial explosion in cases unless careful choices are made
    - E.g., sequence of $n$ if tests can yield up to $2^n$ possible paths
  - Assumption that program structure is basically sound
White Box: Path Coverage

- **Path coverage**
  - What portion of all possible paths through the program are covered by tests?
  - Loop testing: Consider representative and edge cases:
    - Zero, one, two iterations
    - If there is a bound $n$: $n-1$, $n$, $n+1$ iterations
    - Nested loops/conditionals from inside out

- **Advantages**
  - Better coverage of logical flows

- **Disadvantages**
  - Not all paths are possible, or necessary
  - What are the significant paths?
  - Combinatorial explosion in cases unless careful choices are made
    - E.g., sequence of $n$ if tests can yield up to $2^n$ possible paths
  - Assumption that program structure is basically sound
• **Path coverage**
  - What portion of all possible paths through the program are covered by tests?
  - Loop testing: Consider representative and edge cases:
    - Zero, one, two iterations
    - If there is a bound \( n \): \( n-1, n, n+1 \) iterations
    - Nested loops/conditionals from inside out

• **Advantages**
  - Better coverage of logical flows

• **Disadvantages**
  - Not all paths are possible, or necessary
  - What are the **significant** paths?
  - Combinatorial explosion in cases unless careful choices are made
    - E.g., sequence of \( n \) if tests can yield up to \( 2^n \) possible paths
  - Assumption that program structure is basically sound
White Box: Assessing structural coverage

- Coverage assessment tools
  - Track execution of code by test cases
  - Techniques
    - Modified runtime environment (e.g., special JVM)
    - Source code transformation

- Count visits to statements
  - Develop reports with respect to specific coverage criteria

- Example: Clover tool for JUnit tests

<table>
<thead>
<tr>
<th>Coverage</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods: 124 / 274</td>
<td>45.3%</td>
</tr>
<tr>
<td>Statements: 655 / 1,066</td>
<td>33.3%</td>
</tr>
<tr>
<td>Conditionals: 79 / 228</td>
<td>34.2%</td>
</tr>
<tr>
<td>TOTAL:</td>
<td>39.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code: 2,910 Classes: 54</td>
</tr>
<tr>
<td>NC Lines of Code: 1,814 Files: 54</td>
</tr>
<tr>
<td>Methods: 274 Packages: 5</td>
</tr>
</tbody>
</table>

Clover in Eclipse

- Breakdown by package and class
- Coverage
  - Methods
  - Statements
  - Conditionals

- Some metric data
  - LOC
  - Class count
  - Method count
Clover in Eclipse

- Coverage report in editor window
- Warning for statements not covered by test cases

```java
public boolean equals(Object anObject) {
    if (isZero())
        if (anObject instanceof IMoney)
            return ((IMoney)anObject).isZero();
    else if (anObject instanceof Money)
        Money aMoney = (Money)anObject;
    return aMoney.currency().equals(currency())
    && amount() == aMoney.amount();
}
```

Transformed Source Code
White Box Testing: Checkpoints

- Use "checkpoints" in code
  - Access to intermediate values
  - Enable checks during execution

Three approaches

- Logging
  - Create a log record of internal events
  - Tools to support
    - java.util.Logging
    - org.apache.log4j
  - Log records can be analyzed for patterns of events
    - Listener events
    - Protocol events
    - Etc.

- Assertions
  - Logical statements explicitly checked during test runs
  - (No side effects on program variables)
  - Check data integrity
    - Absence of null pointer
    - Array bounds
    - Etc.

- Breakpoints
  - Provide interactive access to intermediate state when a condition is raised

Benefits of White-Box

- Tool support can measure coverage
  - Helps to evaluate test suite (careful!)
  - Can find untested code
- Can test program one part at a time
- Can consider code-related boundary conditions
  - If conditions
    - Boundaries of function input/output ranges
    - e.g. switch between algorithms at data size=100
White Box: Limitations

- Is it possible to achieve 100% coverage?
- Can you think of a program that has a defect, even though it passes a test suite with 100% coverage?
- Exclusive focus on coverage focus misses important bugs
  - Missing code
  - Incorrect boundary values
  - Timing problems
  - Configuration issues
  - Data/memory corruption bugs
  - Usability problems
  - Customer requirements issues
- Coverage is not a good adequacy criterion
  - Instead, use to find places where testing is inadequate

Black-Box (Functional) Testing

- Verify each piece of functionality of the system
  - Black-box: don’t look at the code
  - More common in practice than white-box
- Benefit: finds bugs white-box doesn’t
  - Think like a user, not a programmer
    - The programmer already checked the code!
  - Timing, unanticipated errors, UI, concurrency, configuration issues, performance, hardware failures
- Drawbacks
  - No insight into code structure
  - But good testers will guess anyway!
Black Box: Representative Values

- **Test cases have a statistical distribution similar to expected inputs**
  - Keep generating random inputs until coverage criterion is met
  - Challenge: Do we have a model for the expected input set?

Black Box: Equivalence Class Testing

- **Equivalence classes**
  - A partition of a set
  - Usually the input domain of the program
  - Based on some equivalence relation
    - Intuition: all inputs in an equivalence class will fail or succeed in the same way
Equivalence Class Example

- Program Specification
  - Given 3 numbers, output whether a triangle formed from these number is equilateral, isosceles, or scalene
- Equivalence classes?

Finding Equivalence Classes

- Intuition that test cases are similar
  - This is useful, but can be incomplete
- Use cases in the specification
  - Impractical if you don’t have the spec
  - What if the spec is incomplete?
- One class per code path
  - Impractical if you don’t have code
- Risk-based
  - Consider a possible error as a risk
  - Given that error, what test cases will produce the same result?
Equivalence Class Hueristics

- Invalid inputs
- Ranges of numbers
- Membership in a group
- Equivalent outputs
  - Can you force the program to output an invalid or overflow value?
- Error messages
- Equivalent operating environments

What value to choose from an Equivalence Class?

- **Risk-based**
  - Consider the cost of consequences
    - Vs. frequency of occurrence
    - Focus test data around potential high-impact failures
  
  \[
  \text{Risk} = (\text{cost of consequence}) \times (\text{probability of occurrence})
  \]

  - Challenge: How to model this set of high-consequence failures?

  - Selection heuristic – consider boundary values
    - Extreme or unique cases at or around “boundaries” with respect to preconditions or program decision points
    - Examples: zero-length inputs, very long inputs, null references, etc.
    - Will usually find errors that are present in any other member of the equivalence class, but may find off-by-one errors as well

  - Suited to black box and white box

- **Input**: Information regarding fault/failure relationships

  - **Input**: Information regarding boundary cases
    - Requirements
    - Implementation
Robustness Testing

- Test erroneous inputs and boundary cases
  - Assess consequences of misuse or other failure to achieve preconditions

```java
public static int binsrch (int[] a, int key) {
    int low  = 0;
    int high = a.length - 1;
    while (true) {
        if ( low > high ) return -(low+1);
        int mid = (low+high) / 2;
        if      ( a[mid] < key )  low  = mid + 1;
        else if ( a[mid] > key )  high = mid - 1;
        else    return mid;
    }
}
```

What if the array reference a is null?

Robustness Testing

- Test erroneous inputs and boundary cases
  - Assess consequences of misuse or other failure to achieve preconditions
  - Bad use of API
  - Bad program input data
  - Bad files (e.g., corrupted) and bad communication connections
  - Buffer overflow (security exploit) is a robustness failure
    - Triggered by deliberate misuse of an interface.

- Test apparatus needs to be able to catch and recover from crashes and other hard errors
  - Sometimes multiple inputs need to be at/beyond boundaries

- The question of responsibility
  - Is there external assurance that preconditions will be respected?
  - This is a design commitment that must be considered explicitly

What if the array reference a is null?
Triangle Example

- Program Specification
  - Given 3 numbers, output whether a triangle formed from these number is equilateral, isosceles, or scalene

- Boundary tests?

- Robustness tests?

Combination Testing

- Some errors might be triggered only if two or more variables are at boundary values
- Test combinations of boundary values
  - Combinations of valid input
  - One invalid input at a time
    - In many cases no added value for multiple invalid inputs

- Subtlety required
  - What are the boundary cases for an application that deals with months and days?
Protocol Testing

- **Object protocols**
  - Develop test cases that involve representative sequence of operations on objects
    - Example: Dictionary structure
      - Create, AddEntry*, Lookup, ModifyEntry*, DeleteEntry, Lookup, Destroy
    - Example: IO Stream
      - Open, Read, Read, Close, Read, Open, Write, Read, Close, Close
  - Test concurrent access from multiple threads
    - Example: FIFO queue for events, logging, etc.

- **Approach**
  - Develop representative sequences – based on use cases, scenarios, profiles
  - Randomly generate call sequences
    - Example: Account
      - Open, Deposit, Withdraw, Withdraw, Deposit, Query, Withdraw, Close
    - Coverage: Conceptual states
  - Also useful for protocol interactions within distributed designs

---

Testing example

- **Test preparation**
  - Client scaffold
  - Failure recovery: exceptions

- **Test case selection**
  - Expected cases
    - key found
    - key not found
  - Extreme cases
    - empty array
    - singleton array
    - large array
  - “Sub-unit” testing
    - ordering relation over domain (…)
  - Non-functional testing
    - Performance measurement
    - Expectation: algorithmic analysis
    - Broken code: can yield a linear-time implementation vs. log-time
      - E.g., 1m elements: 20 steps vs. 1,000,000 steps

- **Coverage analysis**
  - Statement, branch, path coverage
  - Data coverage

- **Static analysis and inspection**
  - Initialization; array bounds; arithmetic exceptions; coding style
Testing – The Big Questions

1. What is testing?  
   - And why do we test?

2. What do we test?  
   - Levels of structure: unit, integration, system...

3. How do we select a set of good tests?  
   - Value-driven testing  
   - Functional (black-box) testing  
   - Structural (white-box) testing

4. How do we know when we’re done?  
   - Coverage, Mutation, Capture/Recapture...

5. Practices for testability  
   - What are known best test practices?  
   - How does testing integrate into lifecycle and metrics?

6. What are the limits of testing?  
   - What are complementary approaches?  
     - Inspections  
     - Static and dynamic analysis

When are you done testing?
When are you done testing?

- **Coverage criterion**
  - Must reach X% coverage
  - Legal requirement to have 100% coverage for avionics software
  - Drawback: focus on 100% coverage can distort the software so as to avoid any unreachable code

- **Can look at historical data**
  - How many bugs are remaining, based on matching current project to past experience?
  - Key question: is the historical data applicable to a new project?

- **Can use statistical models**
  - Test on a realistic distribution of inputs, measure % of failed tests
  - Ship product when quality threshold is reached
  - Only as good as your characterization of the input
  - Usually, there’s no good way to characterize this
  - Exception: stable systems for which you have empirical data (telephones)
  - Exception: good mathematical model (avionics)

- **Rule of thumb**: when error detection rate drops
  - Implies diminishing returns for testing investment

---

When are you done testing?

- **Mutation testing**
  - *Perturb code slightly in order to assess sensitivity*
  - Focus on low-level design decisions
  - Examples:
    - Change "<" to ">"
    - Change "0" to "1"
    - Change "≤" to "<"
    - Change "argv" to "argx"
    - Change "a.append(b)" to "b.append(a)"

- **Assess effectiveness of test suite**
  - How many seeded defects are found?
  - coverage metric
  - Principle: % of mutants not found ~ % of errors not found
    - Is this really true?
    - Depends on how well mutants match real errors
      - Some evidence of similarity (e.g. off by one errors) but clearly imperfect
When are you done inspecting?

- **Capture/Recapture assessment**
  - Most applicable for assessing inspections
  - Measure overlap in defects found by different inspectors
  - Use overlap to estimate number of defects not found

- **Example**
  - Inspector A finds \( n_1 = 10 \) defects
  - Inspector B finds \( n_2 = 8 \) defects
  - \( m = 5 \) defects found by both A and B
  - \( N \) is the (unknown) number of defects in the software

- **Lincoln-Petersen analysis** [source: Wikipedia]
  - Consider just the 10 (total) defects found by A
  - Inspector B found 5 of these 10 defects
  - Therefore the probability that inspector B finds a given defect is \( 5/10 \) or 50%
  - So, inspector B should have found 50% of the \( N \) defects in the software, so

\[
N = \frac{n_1 \times n_2}{m} = \frac{10 \times 8}{5} = 20 \text{ defects}
\]

- **Assumptions**
  - All defects are equally easy to find
  - All inspectors are equally effective at finding defects
  - Are these realistic?

---

When are you done testing?

- **Most common**
  - Run out of time or money

- **Ultimately a judgment call**
  - Resources available
  - Schedule pressures
  - Available estimates of quality
Testing – The Big Questions

1. What is testing?
   • And why do we test?

2. What do we test?
   • Levels of structure: unit, integration, system...

3. How do we select a set of good tests?
   • Value-driven testing
   • Functional (black-box) testing
   • Structural (white-box) testing

4. How do we know when we’re done?
   • Coverage, Mutation, Capture/Recapture...

5. Practices for testability
   • What are known best test practices?
   • How does testing integrate into lifecycle and metrics?

6. What are the limits of testing?
   • What are complementary approaches?
   • Inspections
   • Static and dynamic analysis

5a. Practices for testability

1. Document interfaces
   • Write down explicit “rules of the road” at interfaces, APIs, etc

2. Design by contract
   • Specify a contract between service client and its implementation
     • System works if both parties fulfill their contract
     • Use pre- and post-conditions, etc

3. Testing
   • Verify pre- and post-conditions during execution
     • Important Limitation
     • Not all logical formulas can be evaluated directly (forall x in S...)
   • Assign responsibility based on contract expectations
   • Executions become a set of unit tests
5a. Integration/System Testing

2. Do incremental integration testing
   - Test several modules together
   - Still need scaffolding for modules not under test

   • Avoid “big bang” integrations
     - Going directly from unit tests to whole program tests
     - Likely to have many big issues
     - Hard to identify which component causes each

   • Test interactions between modules
     - Ultimately leads to end-to-end system test

   • Used focused tests
     - Set up subsystem for test
     - Test specific subsystem- or system-level features
       - no “random input” sequence
     - Verify expected output

5a. Frequent (Nightly) Builds

3. Build a release of a large project every night
   - Catches integration problems where a change “breaks the build”
     - Breaking the build is a BIG deal—may result in midnight calls to the responsible engineer
   - Use test automation
     - Upfront cost, amortized benefit
     - Not all tests are easily automated – manually code the others

   • Run simplified “smoke test” on build
     - Tests basic functionality and stability
     - Often: run by programmers before check-in
     - Provides rough guidance prior to full integration testing
Practices – Regressions

4. Use regression tests
   - Regression tests: run every time the system changes

   - Goal: catch new bugs introduced by code changes
     - Check to ensure fixed bugs stay fixed
     - New bug fixes often introduce new issues/bugs
     - Incrementally add tests for new functionality

Practices – Acceptance, Release, Integrity Tests

5. Acceptance tests (by customer)
   - Tests used by customer to evaluate quality of a system
   - Typically subject to up-front negotiation

6. Release Test (by provider, vendor)
   - Test release CD
     - Before manufacturing!
     - Includes configuration tests, virus scan, etc
     - Carry out entire install-and-run use case

7. Integrity Test (by vendor or third party)
   - Independent evaluation before release
   - Validate quality-related claims
   - Anticipate product reviews, consumer complaints
   - Not really focused on bug-finding
Practices: Reporting Defects

8. Develop good defect reporting practices

- Reproducible defects
  - Easier to find and fix
  - Easier to validate
    - Built-in regression test
    - Increased confidence
  - Easier to validate
  - More value doing the fix
  - Helps root-cause analysis

- Simple and general
  - State the problem

- Non-antagonistic
  - Don't blame

Practices: Social Issues

9. Respect social issues of testing

- There are differences between developer and tester culture
- Acknowledge that testers often deliver bad news
- Avoid using defects in performance evaluations
  - Is the defect real?
  - Bad will within team
- Work hard to detect defects before integration testing
  - Easier to narrow scope and responsibility
  - Less adversarial
- Issues vs. defects
Practices: Root cause analysis

10. How can defect analysis help prevent later defects?

- Identify the "root causes" of frequent defect types, locations
  - Requirements and specifications?

- Try to find all the paths to a problem
  - If one path is common, defect is higher priority
  - Each path provides more info on likely cause

- Try to find related bugs
  - Helps identify underlying root cause of the defect
  - Can use to get simpler path to problem
    - This can mean easier to fix

- Identify the most serious consequences of a defect

Testing – The Big Questions

1. What is testing?
   - And why do we test?

2. What do we test?
   - Levels of structure: unit, integration, system...

3. How do we select a set of good tests?
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing

4. How do we know when we’re done?
   - Coverage, Mutation, Capture/Recapture...

5. Practices for testability
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?

6. What are the limits of testing?
   - What are complementary approaches?
     - Inspections
     - Static and dynamic analysis
5b. Testing and Lifecycle Issues

1. Testing issues should be addressed at every lifecycle phase
   - Initial negotiation
     - Acceptance evaluation: evidence and evaluation
     - Extent and nature of specifications
   - Requirements
     - Opportunities for early validation
     - Opportunities for specification-level testing and analysis
     - Which requirements are testable: functional and non-functional
   - Design
     - Design inspection and analysis
     - Designing for testability
       - Interface definitions to facilitate unit testing
   - Follow both top-down and bottom-up unit testing approaches
     - Top-down testing
       - Test full system with stubs (for undeveloped code).
       - Tests design (structural architecture), when it exists.
     - Bottom-up testing
       - Units → Integrated modules → system

Lifecycle issues

2. Favor unit testing over integration and system testing
   - Unit tests find defects earlier
     - Earlier means less cost and less risk
     - During design, make API specifications specific
       - Missing or inconsistent interface (API) specifications
       - Missing representation invariants for key data structures
       - What are the unstated assumptions?
         - Null refs ok?
         - Pass out this exception ok?
         - Integrity check responsibility?
         - Thread creation ok?
   - Over-reliance on system testing can be risky
     - Possibility for finger pointing within the team
     - Difficulty of mapping issues back to responsible developers
     - Root cause analysis becomes blame analysis
Test Plan

3. Create a QA plan document

- Which quality techniques are used and for what purposes

Overall system strategy
- Goals of testing
  - Quality targets
  - Measurements and measurement goals
- What will be tested/what will not
  - Don’t forget quality attributes!
- Schedule and priorities for testing
  - Based on hazards, costs, risks, etc.
- Organization and roles: division of labor and expertise
- Criteria for completeness and deliverables

Make decisions regarding when to unit test
- There are differing views
  - CleanRoom: Defer testing. Use separate test team
  - Agile: As early as possible, even before code, integrate into team

Test Strategy Statement

Examples:
- We will release the product to friendly users after a brief internal review to find any truly glaring problems. The friendly users will put the product into service and tell us about any changes they’d like us to make.

- We will define use cases in the form of sequences of user interactions with the product that represent ... the ways we expect normal people to use the product. We will augment that with stress testing and abnormal use testing (invalid data and error conditions). Our top priority is finding fundamental deviations from specified behavior, but we will also use exploratory testing to identify ways in which this program might violate user expectations.

- We will perform parallel exploratory testing and automated regression test development and execution. The exploratory testing will focus on validating basic functions (capability testing) to provide an early warning system for major functional failures. We will also pursue high-volume random testing where possible in the code.

[adapted from Kaner, Bach, Pettichord, Lessons Learned in Software Testing ]
Why Produce a Test Plan?

4. Ensure the test plan addresses the needs of stakeholders
   - Customer: may be a required product
     - Customer requirements for operations and support
     - Examples
       - Government systems integration
       - Safety-critical certification: avionics, health devices, etc.
   - A separate test organization may implement part of the plan
     - “IV&V” – Independent verification and validation
   - May benefit development team
     - Set priorities
     - Use planning process to identify areas of hazard, risk, cost
   - Additional benefits – the plan is a team product
     - Test quality
       - Improve coverage via list of features and quality attributes
       - Analysis of program (e.g. boundary values)
       - Avoid repetition and check completeness
     - Communication
       - Get feedback on strategy
       - Agree on cost, quality with management
     - Organization
       - Division of labor
       - Measurement of progress

Defect Tracking

5. Track defects and issues
   - Issue: Bug, feature request, or query
     - May not know which of these until analysis is done, so track in the same database (Issuezilla)
   - Provides a basis for measurement
     - Defects reported: which lifecycle phase
     - Defects repaired: time lag, difficulty
     - Defect categorization
     - Root cause analysis (more difficult!)
   - Provides a basis for division of effort
     - Track diagnosis and repair
     - Assign roles, track team involvement
   - Facilitates communication
     - Organized record for each issue
     - Ensures problems are not forgotten
   - Provides some accountability
     - Can identify and fix problems in process
     - Not enough detail in test reports
     - Should not be used for HR evaluation
## Testing – The Big Questions

1. **What is testing?**  
   - And why do we test?

2. **What do we test?**  
   - Levels of structure: unit, integration, system...

3. **How do we select a set of good tests?**  
   - Value-driven testing  
   - Functional (black-box) testing  
   - Structural (white-box) testing

4. **How do we know when we’re done?**  
   - Coverage, Mutation, Capture/Recapture...

5. **Practices for testability**  
   - What are known best test practices?  
   - How does testing integrate into lifecycle and metrics?

6. **What are the limits of testing?**  
   - What are complementary approaches?  
     - Inspections  
     - Static and dynamic analysis

---

### 6. What are the limits of testing?

**What we can test**  
- Attributes that can be directly evaluated externally  
  - Examples  
    - **Functional** properties: result values, GUI manifestations, etc.
  - Attributes relating to resource use  
    - Many well-distributed **performance** properties  
    - Storage use

**What is difficult to test?**  
- Attributes that **cannot easily be measured externally**  
  - Is a design evolvable?  
  - Is a design secure?  
  - Is a design technically sound?  
  - Does the code conform to a design?  
  - Where are the performance bottlenecks?  
  - Does the design meet the user’s needs?

- Attributes for which **tests are nondeterministic**  
  - Real time constraints  
  - Race conditions  
  - Rate monotonic scheduling  
  - Analysis of locking

- Attributes relating to the **absence of a property**  
  - Absence of security exploits  
  - Absence of memory leaks  
  - Absence of functional errors  
  - Absence of non-termination  
  - Microsoft’s Standard Annotation Language  
  - Cyclone, Purify  
  - Hoare Logic  
  - Termination analysis
Assurance beyond Testing and Inspection

- **Design analysis: check correctness early**
  - Design Structure Matrices – evolvability analysis
  - Security Development Lifecycle – architectural analysis for security
  - Alloy – systematically exploring a model of a design

- **Static analysis: provable correctness**
  - Reflexion models, ArchJava – conformance to design
  - Fluid – concurrency analysis for race conditions
  - Metal, Fugue – API usage analysis
  - Type systems – eliminate mechanical errors
  - Standard Annotation Language – eliminate buffer overflows
  - Cyclone – memory usage

- **Dynamic analysis: run time properties**
  - Performance analysis
  - Purify – memory usage
  - Eraser – concurrency analysis for race conditions
  - Test generation and selection – lower cost, extend range of testing

- **Manual analysis: human verification**
  - Hoare Logic – verification of functional correctness
  - Real-time scheduling