Software Quality, part 2

Testing

Travis Breaux (with slides by Bill Scherlis)
1. What is testing?
   • And why do we test?

2. What do we test?
   • How much information do we have about the system?
     • White box . . . Gray box . . . Black box
   • What levels of structure do we test?
     • Unit test . . . Cluster/Integration test . . . System test

3. How do we select a set of good tests?
   • Do we have all cases “covered”?
     • Coverage criteria
   • What are examples of properties that are difficult to test?

4. How do we assess our test sets?
   • Theories of testing

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 and reviews
     • Analysis prospects and demonstration

- **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
    - Psychology experiment: Look for blips on a screen
      - More blips found if your are rewarded for finding them than if you are penalized for false alarms
      - (Teasley, Leventhal, Mynatt & Rohlman)
  - 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
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
Techniques for 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
Unit Test and Scaffolding

Examples: Database, Network, Interconnected systems

Example: Network client with GUI

Code to be tested
Unit Test and Scaffolding

Code to be tested

Substitute: Test support code
Unit Test and Scaffolding

Code to be tested

Cluster
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”
JUnit on one slide

• **To create test cases**
  - Create a new class for your tests (usually one per each class under test)
  - Write a set of **public** methods, annotated with @Test
  - In each method, invoke the assert family of methods to perform checks
    - In class `org.junit.Assert`
    - Examples: `assertTrue`, `assertEquals`, `assertSame`, `assertNull`
  - Tests are not guaranteed to run in any particular order

• **Optionally**
  - To run initialization code before each test, annotate the initialization method with `@Before`
  - To finalize after each test, annotate a method with `@After`
  - To initialize/finalize before all tests, annotate a method with `@BeforeClass/@AfterClass`

• **Note:** JUnit 3 uses naming conventions, not annotations
Categories of unit tests

- **Code visibility – black box**
  - Cannot see internal code elements of the service being tested
  - Test through the public API – better for functional attributes
  
  - *We can create a version of MoneyTest with reference only to the API*

  - **Domain coverage analysis**

- **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

  - *How would we enhance our version of MoneyTest with this information?*

  - **Code coverage analysis**
Testing – The Big Questions

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

2. What do we test?
   • How much information do we have about the system?
     • White box . . . Gray box . . . Black box
   • What levels of structure do we test?
     • Unit test . . . Cluster/Integration test . . . System test

3. How do we select a set of good tests?
   • Do we have all cases “covered”?
     • Coverage criteria
   • What are examples of properties that are difficult to test?

4. How do we assess our test sets?
   • Theories of testing

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 and reviews
     • Analysis prospects and demonstration
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
Test coverage – Criteria

• **Test coverage criteria**

  ▪ What criteria do we apply to select inputs?

    • Select *representative input element* domains in order to assure coverage criteria are met

    • Direct instrumentation (modifying code to include counters on all edges of program graph) may be used to assess coverage empirically

  ▪ Diverse strategies

    • Testers usually employ some combination of these
Test coverage 1

• Test coverage 1 – 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?

  ▪ A **black box** approach – knowledge of code not essential
    • Finds bugs white-box testing may miss
      • Think like a user, not programmer
      • Good testers will guess internal structure
      • White-box can *bias* testers
        – Focus on code structure rather than requirements structure

  ▪ Scenario testing – build test cases around requirements scenarios and use cases
    • Validation

  ▪ **Input**: Information regarding the distribution of inputs

  ▪ **Input**: Information regarding requirements
Test coverage 1

- Test coverage 1 – 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?

- Approaches
  - Develop a model of the multiple dimensions of the input domain
    - Identify equivalence classes within the input domains for each dimension
      - Any element of the class is representative of any of the others, for the purpose of test outcome
    - Develop classes for good and bad input sets in each dimension
    - Create test cases according to the cross product of these classes
  - Develop a model of the “boundary cases” for each input dimension
    - Examples
      - Empty, singleton, full, overfull input arrays
      - Minimum, zero, and maximum numeric values
      - Extreme floating point values
    - Apply this approach to multiple dimensions simultaneously
      - “Double mode faults”
Test coverage 2 – Risk-based

Consider the cost of consequences
- Vs. frequency of occurrence (see Approach 1)
- Focus test data around potential high-impact failures

Risk = (cost of consequence) * (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.

- Suited to **black box** and **white box**

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

**Input**: Information regarding boundary cases
- Requirements
- Implementation
Test coverage 3

- **Test coverage 3 – Structural analysis**
  - Structural analysis
    - Organized according to program decision structure
    - Touching: statement, branch

```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;
    }
}
```

- Will this statement get executed in a test?
- Does it return the correct result?
- Could this array index be out of bounds?
- Does this return statement ever get reached?
Test coverage 3

• Test coverage 3 – **Structural analysis**
  - Structural analysis
    - *Organized according to program decision structure*
    - Touching: statement, branch

```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 loop executes only once?
• What about the combination of setting low and then setting high?

• Could this division raise an exception? (leading control flow to an abrupt exit)
Test coverage 3: Structural Analysis

```java
public boolean equals(Object anObject) {
    if (isZero())
        return ||IMoney)anObject).isZero();
    if (anObject instanceof IMoney) {
        Money aMoney = (Money)anObject;
        return aMoney.currency().equals(currency())
                && amount() == aMoney.amount();
    }
    return false;
}
```
Test coverage 3a: 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!
Test coverage 3b: 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”
Test coverage 3c: 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
Test coverage 3d: Data flow

- **Data flow coverage**
  - Identify all possible assignment statements (or binding points) at which a given variable use could have been given a value

```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;
    }
}
```

- These two sites
- Where could low have received its value?
Test coverage 3d: Data flow

- **Data flow coverage**
  - Identify all possible assignment statements (or binding points) at which a given variable use could have been given a value

- **Advantages**
  - Enables identification and tracking of bad data values
  - Enables identification of uninitialized variables
  - Enables identification of variables set but not used

- **Disadvantages**
  - Complex and often inexact analysis is needed to support
Test coverage 3: 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**

```
Coverage

<table>
<thead>
<tr>
<th>Methods</th>
<th>Statements</th>
<th>Conditionals</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>124 / 274</td>
<td>655 / 1,666</td>
<td>78 / 228</td>
<td>39.5%</td>
</tr>
</tbody>
</table>

Metrics

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

- Counters added
  - To all statements, including boolean structure in conditionals.

```java
public boolean equals(Object anObject) throws Exception {
    __CLOVER_22_0.M[73]++;
    __CLOVER_22_0.S[273]++;
    if (((isZero()) && (++__CLOVER_22_0.CT[34] != 0)))
        (++__CLOVER_22_0.CF[34] == 0));
    __CLOVER_22_0.S[274]++;
    if (((anObject instanceof IMoney) &&
         (++__CLOVER_22_0.CT[35] != 0)));
    __CLOVER_22_0.S[275]++; return ((IMoney)anObject).isZero();
    __CLOVER_22_0.S[276]++; if (((anObject instanceof Money) &&
        (++__CLOVER_22_0.CT[36] != 0))
        (++__CLOVER_22_0.CF[36] == 0));
    __CLOVER_22_0.S[277]++;
    Money aMoney = (Money)anObject;
    __CLOVER_22_0.S[278]++; return aMoney.currency().equals(currency());
    }
}
```
Test coverage 4 – Mutations

- Test coverage 4 – **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)”

  - Coverage criterion – are all mutants caught?
    - **Relational coverage**: tests can distinguish a>b from a≥b

- **Mutation testing processes**
  - Process 1: Seed defects and identify test cases to find them.
    - This provides a coverage metric.
  - Process 2: One group seeds defects. Another group searches for them.
    - Evaluates effectiveness of testing regime.

- **Advantages**
  - Can assist in coverage analysis
  - Can assist in assuring absence of “off by one” errors

- **Disadvantages**
  - Combinatorial explosion unless clever strategy is used
Test coverage 5 – Robustness

- Test coverage 5 – Robustness
  - 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?
Test coverage 5 – Robustness

- **Test coverage 5 – Robustness**
  - *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 (e.g., *Ballista* tool)
  - 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?
Test coverage 6 – Object Protocols

- **Test coverage 6 – 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.

<table>
<thead>
<tr>
<th>Create</th>
<th>Put</th>
<th>Put</th>
<th>Get</th>
<th>Get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Get</td>
<td>Get</td>
<td>Put</td>
<td>Put</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Get</td>
</tr>
</tbody>
</table>

- **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**
“Coverage” is useful but also dangerous

• Examples of what coverage analysis could miss
  ▪ 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
Regression Testing – No Specifications!

• What is “covered” by a set of regression tests?
• Why do we do regression testing?

• Regression
  ▪ Usual model:
    • Introduce regression tests for bug fixes, etc.
  ▪ Compare results as code evolves
    \[ \text{Code1} + \text{TestSet} \rightarrow \text{TestResults1} \]
    \[ \text{Code2} + \text{TestSet} \rightarrow \text{TestResults2} \]
    • As code evolves, compare \text{TestResults1} with \text{TestResults2}, etc.

• Benefits
  ▪ Ensure bug fixes remain in place and bugs do not reappear
  ▪ Avoids need for specifications
    • The pair \(<\text{TestSet},\text{TestResults1}>\) acts as a specification
1. What is testing?
   • And why do we test?

2. What do we test?
   • How much information do we have about the system?
     • White box . . . Gray box . . . Black box
   • What levels of structure do we test?
     • Unit test . . . Cluster/Integration test . . . System test

3. How do we select a set of good tests?
   • Do we have all cases “covered”?
     • Coverage criteria
   • What are examples of properties that are difficult to test?

4. How do we assess our test sets?
   • Theories of testing

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 and reviews
     • Analysis prospects and demonstration
Theories of Testing

Sources: Frankl, Weyeuker, Hamlet, Taylor
4. How do we assess our test sets?

Definitions

- **Test data** – test inputs
- **Test case** – pairs (input, output) consistent with a specification
- **Test set** – a set of test data or cases (a subset of the input domain)
- **Code + Test set** → **test results**
- **Test selection criterion** – conditions for selection of a TestSet
  - E.g., different kinds of **coverage**

Choosing good test selection criteria

- **Valid** test-set selection criterion
  - If the program is incorrect, then there is a failing test set that meets the criterion
- **Reliable** test-set selection criterion
  - If one test set fails, then all do
- **Example**:
  - **Specification**: compute x+x
  - **Program**: actually computes x*x
  - Criterion #1: test set is {0,2}: reliable, but not valid
  - Criterion #2: test set is {0,1,2,3,4}: valid, but not reliable
  - What if program actually computes d+2? (#1 is valid, not reliable)
  - What if program computes d+5? (#2 is both valid and reliable)
- **Ideal** criterion: both valid and reliable
  - Rarely achieved
Subdomains and homogeneity

- **Motivation**
  - Cannot generally achieve validity and reliability

- **Subdomain**
  - $S$ is a subset of the input domain $D$
  - A criterion $C$ is **revealing** (a.k.a. **homogeneous**) for $S$ if
    - A successful test in $S$ implies program $P$ is correct for $S$
    - An incorrect test in $S$ implies all tests in $S$ will fail
    - *Intuition*: test cases are “similar” – all will give similar behavior

- **Problem**
  - Cannot generally achieve homogeneity
  - Difficult to select subdomains

- **Better approach**
  - Select subdomains revealing for an error

  - **Subdomain** $S$ is **revealing for an error** $E$ if
    - If there is an error $E$ in $P$ that causes a wrong result for any case in $S$, then all cases in $S$ fail.
      - Equivalently: If *any* test in $S$ is ok, then the error $E$ is not in $P$
    - Why: Programmers know where errors are likely to occur.
Partitioning (or “equivalence class analysis”)

• Organize the input domain D into subdomains
  ▪ Goal: Many of the subdomains are revealing
  ▪ Example: intersect path domains (derived from the program) with problem domains (derived from the specification)

• Partitioning the domain
  ▪ Path, branch, flow

• Overlapping subdomains
  ▪ Mutation, suspicion, specification
In practice

- “Random is as good as subdomain” for confidence building
  - [Hamlet&Taylor]
    - We don’t know the right subdomains
    - They aren’t homogeneous

- For critical failure cases
  - Use fault-trees and partitions
1. **What is testing?**
   - And why do we test?

2. **What do we test?**
   - How much information do we have about the system?
     - *White box* . . . *Gray box* . . . *Black box*
   - What levels of structure do we test?
     - *Unit test* . . . *Cluster/Integration test* . . . *System test*

3. **How do we select a set of good tests?**
   - Do we have all cases “covered”?
     - *Coverage criteria*
   - What are examples of properties that are difficult to test?

4. **How do we assess our test sets?**
   - *Theories of testing*

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 and reviews*
     - *Analysis prospects and demonstration*
Test Driven Development Techniques

Source for testing experience
Kaner, Falk, Nguyen.
5a. Practices for testability

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

• 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

• 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

4.4.2 delete_binding

The delete_binding API call causes one or more instances of bindingTemplate to be removed from the registry.

4.4.2.1 Syntax:

```xml
<delete_binding generic="2.0" xmlns="urn:xudi:org:api_v2"> 
    ???????? <authInfo/>
    ???????? <bindingKey/> [<?<bindingKey/> ?>]
</delete_binding>
```

4.4.2.2 Arguments:

? **authInfo**: this required argument is an element that contains an authenticating token obtained using the get_authToken API call.

? **bindingKey**: one or more uuid_key values that represent specific instances of bindings.

4.4.2.3 Returns:

Upon successful completion, a dispositionReport is returned with a single success bindingTemplates that are deleted as a result of this call, such as those referenced in Redirector elements) are not affected.

4.4.2.4 Caveats:

If any error occurs in processing this API call, a dispositionReport structure will be Fault. If the following error number information will be relevant:

? **E_invalidKeyPassed**: signifies that one of the uuid_key values passed bindingKey values. No partial results will be returned if any bindingKey in the message contained multiple instances of a uuid_key value, this error caused the problem will be clearly indicated in the error text.

? **E_authTokenExpired**: signifies that the authentication token value passed
Practices – Integration 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
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

```java
public void testRegression() {
    try {
        new Path("C:|\eclipse");
    } catch(Exception e) {
        fail("1.0", e);
    }
    try {
        if (WINDOWS) {
            IPath path = new Path("d:|\ive");
            assertTrue("2.0", \path.isUNC());
            assertEquals("2.1", 1, path.segmentCount());
            assertEquals("2.2", "ive", path.segment(0));
        }
    } catch(Exception e) {
        fail("2.99", e);
    }
```

* This test is for bizarre cases that previously caused errors. */

Scherlis, Jan 2009 ©
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

- Simple and general
  - More value doing the fix
  - Helps root-cause analysis

- Non-antagonistic
  - State the problem
  - 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

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

2. What do we test?  
   • How much information do we have about the system?  
     • White box . . . Gray box . . . Black box  
   • What levels of structure do we test?  
     • Unit test . . . Cluster/Integration test . . . System test

3. How do we select a set of good tests?  
   • Do we have all cases “covered”?  
     • Coverage criteria  
   • What are examples of properties that are difficult to test?

4. How do we assess our test sets?  
   • Theories of testing

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 and reviews  
     • Analysis prospects and demonstration
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
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
    - Not rapid enough response to bug reports
  - Should not be used for HR evaluation
6. Completeness?

- **Statistical thresholds**
  - Defects reported/repaired
  - Relative proportion of defect kinds
  - Predictors on “going gold”

- **Coverage criterion**
  - E.g., 100% coverage required for avionics software
  - Distorts the software
  - Matrix: Map test cases to requirements use cases

- **Can look at historical data**
  - Within an organization, can compare across projects
    - Develop expectations and predictors
    - (More difficult across organizations, due to difficulty of commensurability.)
      - E.g., telecon switches vs. consumer software

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

- **Most common**
  - Run out of time or money
Testing – The Big Questions

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

2. **What do we test?**
   - How much information do we have about the system?
     - *White box* . . . *Gray box* . . . *Black box*
   - What levels of structure do we test?
     - *Unit test* . . . *Cluster/Integration test* . . . *System test*

3. **How do we select a set of good tests?**
   - Do we have all cases “covered”?
     - *Coverage criteria*
   - What are examples of properties that are difficult to test?

4. **How do we assess our test sets?**
   - *Theories of testing*

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 and reviews*
     - *Analysis prospects and demonstration*
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**
    - “Universal properties”
    - *Examples*
      - Security attributes: Confidentiality (covert channels), Integrity
      - Safe use of storage and encapsulation
  - Attributes for which **tests are nondeterministic**
    - Results may vary with each execution
    - *Examples*
      - Race conditions and interference
  - Attributes that depend on **“deep” properties**
    - Are there any references to private objects?
    - Are there any other threads executing?
    - The program *always* terminates, regardless of input
  - Attributes relating to an **absence of a property**
    - Absence of malicious code
    - Absence of trap doors (Excel flight simulator)

---

**Kinds of software defects**
- Functional errors
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Design defects
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
  - Method call protocol
  - Respect for callbacks
  - Set up / tear down
  - Aliasing
  - Effects
  - Locking roles
  - Use of threads
  - Exception handling
Tough bugs [Kaner]

- When entering data, a user fidgets, pressing alternatively number and backspace. When the number is finally entered, all the numbers and backspace keys are flushed from a buffer to the server, overflowing the input buffer and crashing the system.

- A database management program breaks with files a multiple of 16,384 bytes long

- A word processor deleting paragraphs from large, fragmented disk files during editing

- A telephone system has 6 states, one of which involves placing a caller on hold. The system keeps a stack of calls placed on hold. If the caller hangs up while on hold, the information is left on the stack until the phone is idle. But if 30 callers hang up before the phone is next idle, the stack overflows and the phone crashes.
A Sample of testing difficulties

- **Reactive systems** – modeling and testing programs that react
  - Conventional model of programs as mathematical functions
  - Includes embedded and real-time software
  - Includes software with GUIs

- **Human-in-the-loop systems** – people are part of the system
  - People must generally be part of testing the system
  - Critical systems, transactions, etc.

- **Non-determinism** – there is asynchrony / concurrency in execution.
  - That is, there are multiple ways for the program to execute on a given input
  - In general, testing cannot detect race conditions and potential for deadlocks

- **Software embedded in systems** – the models of the environment (physical and software) in which the system executes are incorrect
  - Have potential failures and problems in the environment been anticipated?
  - What architectural models yield testable dependable systems?

- **Requirements engineering** – have we correctly modeled the requirements
  - Validation – Do the test cases reflect the intended correct results
  - User interface – simulate all possible actions through a UI

- **Performance and stress** – how does the system handle large loads, and, generally, how does it scale up?
  - What infrastructure/resources are needed to support scale up?
Testing – The Big Questions

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

2. What do we test?
   - How much information do we have about the system?
     - White box . . . Gray box . . . Black box
   - What levels of structure do we test?
     - Unit test . . . Cluster/Integration test . . . System test

3. How do we select a set of good tests?
   - Do we have all cases "covered"?
     - Coverage criteria
   - What are examples of properties that are difficult to test?

4. How do we assess our test sets?
   - Theories of testing

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 and reviews
     - Analysis prospects and demonstration
Assurance 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
End