Why Does Software Contain Bugs?

- Our limited ability to predict the behavior of our code
  - Software is extremely complex
  - No developer can understand the whole system

- We make mistakes
  - Unclear requirements, miscommunication
  - Wrong assumptions (e.g., behavior of operating system)
  - Design errors (e.g., capacity of data structure too small)
  - Coding errors (e.g., wrong loop condition)
“First actual case of bug being found.”
Increasing Software Reliability

Fault Avoidance
- Detect faults statically without executing the program
- Includes development methodologies, reviews, and program verification

Fault Detection
- Detect faults by executing the program
- Includes testing

Fault Tolerance
- Recover from faults at runtime (e.g., transactions)
- Includes adding redundancy (e.g., n-version programming)
Goal of Testing

- An error is a deviation of the observed behavior from the required (desired) behavior
  - Functional requirements (e.g., user-acceptance testing)
  - Nonfunctional requirements (e.g., performance testing)

- Testing is the process of executing a program with the intent of finding errors

- A successful test is one that finds errors
Limitations of Testing

*Testing can only show the presence of bugs, not their absence.*  

---

- It is impossible to completely test any nontrivial module or system
  - Theoretical limitations: termination
  - Practical limitations: prohibitive in time and cost
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
Test Stages

- Requirements Elicitation
- Detailed Design
- System Design
- Implementation
- Unit Test
- Integration Test
- System Test
Creation of Test Harness

- **Test driver**
  - Applies test cases to UUT including setup and clean-up

- **Test stub**
  - Partial, temporary impl. of a component used by UUT
  - Simulates the activity of a missing component by answering to the calling sequence of the UUT and returning back fake data
Unit Testing

- Testing individual subsystems (collection of classes)

- Goal: Confirm that subsystem is correctly coded and carries out the intended functionalities
### Unit Test Example (JUnit)

```java
class SavingsAccount {
    ...
    public void deposit( int amount ) { ... }
    public void withdraw( int amount ) { ... }
    public int getBalance( ) { ... }
}

@Test
public void withdrawTest( ) {
    SavingsAccount target = new SavingsAccount();
    target.deposit( 300 );
    int amount = 100;
    target.withdraw( amount );
    Assert.assertTrue( target.getBalance( ) == 200 );
}
```

**Test Driver**
Implement `SavingsAccount` and its methods.

**Create Test Data**
Set initial deposit to 300 and withdraw amount to 100.

**Create Test Oracle**
Verify `getBalance` returns 200 after withdrawal.
Unit Testing: Discussion

- To achieve a reasonable test coverage, one has to test each method with several inputs
  - To cover valid and invalid inputs
  - To cover different paths through the method

```java
@Test
public void withdrawTest() {
    SavingsAccount target = new SavingsAccount();
    target.deposit(500);
    int amount = 0;
    target.withdraw(amount);
    Assert.assertTrue(target.getBalance() == 500);
}
```

Boiler-plate code for creating test data and writing test oracles
Parameterized Unit Tests (NUnit)

- Parameterized tests that take arguments for test data
  - Decouple test driver (logic) from test data

```csharp
[Test]
public void withdrawTest(int balance, int amount) {
    SavingsAccount target = new SavingsAccount();
    target.deposit(balance);
    target.withdraw(amount);
    Assert.IsTrue(target.getBalance() == balance - amount);
}
```

- Test data can be specified as values, ranges, or random values
- Requires generic test oracles
Generic Test Oracles: Example

```csharp
public static void bubbleSort( int[] a ) {
    for( int i = 0; i < a.Length - 1; i++ ) {
        for( int j = i + 1; j < a.Length; j++ ) {
            if( a[ i ] > a[ j ] )
            {
                int tmp = a[ i ]; a[ i ] = a[ j ]; a[ j ] = tmp;
            }
        }
    }
}
```

```csharp
[Test]
public void bubbleSortTest( ) {
    int[] a = { 7, 2, 5, 2 };
    bubbleSort( a );
    int[] expected = { 2, 2, 5, 7 };
    Assert.AreEqual( expected, a );
}
```

Create test data
Create test oracle
Generic Test Oracles: Example

```csharp
[Test]
public void bubbleSortTest(int[] a) {
    int[] original = (int[])a.Clone();
    bubbleSort(a);
    for (int i = 0; i < a.Length - 1; i++)
        Assert.IsTrue(a[i] <= a[i + 1]);

    bool[] visited = new bool[a.Length];
    for (int i = 0; i < a.Length; i++) {
        int j;
        for (j = 0; j < a.Length; j++) {
            if (!visited[j] && a[i] == original[j])
                { visited[j] = true; break; }
        }
        Assert.IsFalse(j == a.Length);
    }
}
```

- Save test data for later comparison
- Check that array is sorted
- Check that array is a permutation of original array
- Value a[i] is not in the original array
Parameterized Unit Tests: Discussion

- Parameterized unit tests avoid boiler-plate code

- Writing generic test oracles is sometimes difficult
  - Analogous to writing strong postconditions

- Still several test methods are needed, for instance, for valid and invalid input

- Parameterized unit tests are especially useful when test data is generated automatically (see later)
Test Execution

- Execute the test cases

- Re-execute test cases after every change
  - Automate as much as possible
  - For instance, before every commit to the repository

- Regression testing
  - Testing that everything that used to work still works after changes are made to the system
  - Also important for system testing
Eight Rules of Testing

1. Make sure all tests are fully automatic and check their own results
2. A test suite is a powerful bug detector that reduces the time it takes to find bugs
3. Run your tests frequently—every test at least once a day
4. When you get a bug report, start by writing a unit test that exposes the bug
5. Better to write and run incomplete tests than not run complete tests
6. Concentrate your tests on boundary conditions
7. Do not forget to test exceptions raised when things are expected to go wrong
8. Do not let the fear that testing can’t catch all bugs stop you from writing tests that will catch most bugs [M. Fowler]
Integration Testing

- Testing groups of subsystems and eventually the entire system

- Goal: Test interfaces between subsystems
Integration Testing Strategy

- The order in which the subsystems are selected for testing and integration

- Typical strategies
  - Big-bang integration (non-incremental)
  - Bottom-up integration
  - Top-down integration

- Selection criteria
  - Amount of test harness (stubs and drivers)
  - Scheduling concerns
System Testing

- Testing the entire system

Goal: Determine if the system meets the requirements (functional and non-functional)
System Testing Stages

1. Entire System
2. Functional Test
   - Functional requirements
3. Performance Test
   - Non-functional requirements
4. Acceptance Test
   - Client’s understanding of requirements
5. Installation Test
   - User Environment
Functional Testing

- **Goal:** Test functionality of system
  - System is treated as black box

- **Test cases are designed from requirements**
  - Based on use cases
  - Alternative source: user manual

- **Test cases describe**
  - Input data
  - Flow of events
  - Results to check
Acceptance Testing

- **Goal:** Demonstrate that the system meets customer requirements and is ready to use
- **Performed by the client,** not by the developer

- **Alpha test**
  - Client uses the software at the developer’s site
  - Software used in a controlled setting, with the developer ready to fix bugs

- **Beta test**
  - Conducted at client’s site (developer is not present)
  - Software gets a realistic workout in target environments
Independent Testing

- Programmers have a hard time believing/accepting that they have made a mistake
  - Plus a vested interest in not finding mistakes
  - Often stick to the data that makes the program work

- Designing and programming are constructive tasks
  - Testers must seek to break the software

- Testing is done best by independent testers
Independent Testing: Responsibilities

- **System Test**
  - Performed by independent test team
    - One exception
      - Clients perform acceptance testing

- **Integration Test**
  - Performed by independent test team

- **Unit Test**
  - Performed by programmer
    - Requires detailed knowledge of code
    - Immediate bug fixing
Independent Testing: Wrong Conclusions

- The developer should not be testing at all
- Testers get only involved once software is done
- Toss the software over the wall for testing
  - Testers & developers collaborate in developing test suite
- Testing team is responsible for assuring quality
  - Quality is assured by a good software process
Summary

- **Main objective**
  - Design tests that systematically uncover different classes of errors with a minimum amount of time and effort
  - A good test has a high probability of finding an error
  - A successful test uncovers an error

- **Secondary benefits**
  - Demonstrate that software appears to be working according to specification (functional and non-functional)
  - Data collected during testing provides indication of software reliability and software quality
  - Good testers clarify the specification (creative work)
5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.4 Structural Testing
Testing Steps

Select what will be tested
- What parts of the system?
- What aspects of the system?

Select test strategy
- What integration strategy?
- How is the test data determined?

Define test cases
- What are the test data?
- How is the test carried out?

Create test oracle
- What are the expected results?
- Defined before executing tests
Example: Solve Quadratic Equation

```java
void roots(double a, double b, double c) {
    double q = b*b - 4*a*c;
    if( q > 0 && a != 0 ) {
        numRoots = 2;
        double r = Math.sqrt( q );
        x1 = (-b + r) / (2 * a);
        x2 = (-b - r) / (2 * a);
    } else if( q == 0 ) {
        numRoots = 1;
        x1 = -b / (2 * a);
    } else {
        numRoots = 0;
    }
}
```

Fails if \( a=0 \) and \( b^2 - 4ac = 0 \)

Wrong result if \( a=0 \) and \( b^2 - 4ac > 0 \)

\[ x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \]
\[ x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \]
Strategy 1: Exhaustive Testing

- Check UUT for all possible inputs
  - Not feasible, even for trivial programs

```c
void roots( double a, double b, double c ) {
    ...
}
```

- Assuming that `double` represents 64-bit values, we get $(2^{64})^3 \approx 10^{58}$ possible values for a, b, c

- Programs with heap data structures have a much larger state space!
Strategy 2: Random Testing

- Select test data uniformly

```c
void roots( double a, double b, double c ) {
    double q = b*b - 4*a*c;
    if( q > 0 && a != 0 ) {
        ...
    } else if( q == 0 ) {
        numRoots = 1;
        x1 = -b / (2 * a);
    } else {
        ...
    }
}
```

Fails if $a==0$ and $b*b-4*a*c == 0$

The likelihood of selecting $a==0$ and $b==0$ randomly is $1/10^{38}$
Random Testing: Observations

- Random testing focuses on generating test data fully automatically

- Advantages
  - Avoids designer/tester bias
  - Tests robustness, especially handling of invalid input and unusual actions

- Disadvantages
  - Treats all inputs as equally valuable
Strategy 3: Functional Testing

- Use **requirements knowledge** to determine test cases

Given three values, $a$, $b$, $c$, compute all solutions of the equation $ax^2 + bx + c = 0$

<table>
<thead>
<tr>
<th>Two solutions</th>
<th>One solution</th>
<th>No solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \neq 0$ and $b^2-4ac &gt; 0$</td>
<td>$a = 0$ and $b \neq 0$ or $a \neq 0$ and $b^2-4ac = 0$</td>
<td>$a = 0$, $b = 0$, and $c \neq 0$ or $a \neq 0$ and $b^2-4ac &lt; 0$</td>
</tr>
</tbody>
</table>

Test each case of the specification
Functional Testing: Observations

- Functional testing focuses on input/output behavior
  - Goal: Cover all the requirements

- Attempts to find
  - Incorrect or missing functions
  - Interface errors
  - Performance errors

- Limitations
  - Does not effectively detect design and coding errors (e.g., buffer overflow, memory management)
  - Does not reveal missing cases in the specification
Strategy 4: Structural Testing

- Use design knowledge about system structure, algorithms, data structures to determine test cases that exercise a large portion of the code

```c
void roots( double a, double b, double c ) {
    double q = b*b - 4*a*c;
    if( q > 0 && a != 0 ) {
        ...
    } else if( q == 0 ) {
        ...
    } else {
        ...
    }
}
```

Test this case
and this case
and this case

Error might still be missed, for instance, when case is tested with a==1, b==2, c==1
Structural Testing: Observations

- Structural testing focuses on thoroughness
  - Goal: Cover all the code

- Not well suited for system test
  - Focuses on code rather than on requirements, for instance, does not detect missing logic
  - Requires design knowledge, which testers and clients do not have
  - Thoroughness would lead to highly-redundant tests
Testing Strategies: Summary

Functional testing
- Goal: Cover all the requirements
- Black-box test
- Suitable for all test stages

Structural testing
- Goal: Cover all the code
- White-box test
- Suitable for unit testing

Random testing
- Goal: Cover corner cases
- Black-box test
- Suitable for all test stages
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
Applications of Functional Testing

- Black-box test a unit against its requirements

Diagram:

- Unit Test
- Integration Test
- System Test
- Functional test
- Acceptance test
- Test interfaces between subsystems
- During test-driven development, when code is not yet written
5. Testing

5.1 Test Stages
5.2 Test Strategies

5.3 Functional Testing
   5.3.1 Partition Testing
   5.3.2 Selecting Representative Values
   5.3.3 Combinatorial Testing

5.4 Structural Testing
Finding Representative Inputs

- Divide inputs into equivalence classes
  - Each possible input belongs to one of the equivalence classes
  - Goal: some classes have higher density of failures

- Choose test cases for each equivalence class

- Requirement not implemented
- Requirement implemented incorrectly
- Requirement implemented correctly
- Failure
- No failure
### Equivalence Classes: Example

Given a month (an integer in \([1;12]\)) and a year (an integer), compute the number of days of the given month in the given year (an integer in \([28;31]\))

<table>
<thead>
<tr>
<th>month</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month with 28 or 29 days</td>
<td>Leap years (year (\text{mod} \ 4 = 0) and year (\text{mod} \ 100 \neq 0) or year (\text{mod} \ 400 = 0))</td>
</tr>
<tr>
<td>Months with 30 days</td>
<td>Non-leap years year (\text{mod} \ 4 \neq 0) or (year (\text{mod} \ 100 = 0) and year (\text{mod} \ 400 \neq 0))</td>
</tr>
<tr>
<td>Months with 31 days</td>
<td></td>
</tr>
</tbody>
</table>

month = 2
month \(\in\) \{4, 6, 9, 11\}
month \(\in\) \{1, 3, 5, 7, 8, 10, 12\}

Invalid inputs missing
Equivalence Classes: Example (cont’d)

Given a month (an integer in [1;12]) and a year (an integer), compute the number of days of the given month in the given year (an integer in [28;31])

<table>
<thead>
<tr>
<th>month</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month with 28 or 29 days</td>
<td>Leap years (year mod 4 = 0 and year mod 100 \neq 0) or year mod 400 = 0</td>
</tr>
<tr>
<td>Months with 30 days</td>
<td>Non-leap years (year mod 4 \neq 0 or (year mod 100 = 0 and year mod 400 \neq 0))</td>
</tr>
<tr>
<td>Months with 31 days</td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>month = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month with 28</td>
<td>month \in {4, 6, 9, 11}</td>
</tr>
<tr>
<td>or 29 days</td>
<td></td>
</tr>
<tr>
<td></td>
<td>month \in {1, 3, 5, 7, 8, 10, 12}</td>
</tr>
<tr>
<td></td>
<td>month &lt; 1 or month &gt; 12</td>
</tr>
</tbody>
</table>

Partitioning seems too coarse
Equivalence Classes: Example (cont’d)

Given a month (an integer in \([1;12]\)) and a year (an integer), compute the number of days of the given month in the given year (an integer in \([28;31]\))

<table>
<thead>
<tr>
<th>month</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month with 28 or 29 days</td>
<td>Standard leap years</td>
</tr>
<tr>
<td>Month (\in{4, 6, 9, 11})</td>
<td>year (\text{mod} 4 = 0) and (\text{year \text{mod} 100 \neq 0})</td>
</tr>
<tr>
<td>Months with 30 days</td>
<td>Standard non-leap years</td>
</tr>
<tr>
<td>Month (\in{4, 6, 9, 11})</td>
<td>year (\text{mod} 4 \neq 0)</td>
</tr>
<tr>
<td>Months with 31 days</td>
<td>Special leap years</td>
</tr>
<tr>
<td>Month (\in{1, 3, 5, 7, 8, 10, 12})</td>
<td>year (\text{mod} 400 = 0)</td>
</tr>
<tr>
<td>Invalid</td>
<td>Special non-leap years</td>
</tr>
<tr>
<td>Month (&lt; 1) or (\text{month} &gt; 12)</td>
<td>year (\text{mod} 100 = 0) and (\text{year \text{mod} 400 \neq 0})</td>
</tr>
</tbody>
</table>
5. Testing

5.1 Test Stages
5.2 Test Strategies

5.3 Functional Testing
   5.3.1 Partition Testing
   5.3.2 Selecting Representative Values
   5.3.3 Combinatorial Testing

5.4 Structural Testing
Selecting Representative Values

- Once we have partitioned the input values, we need to select \textit{concrete values} for the test cases for each equivalence class

- Input from a range of valid values
  - Below, within, and above the range
  - Also applies to multiplicities on aggregations

- Input from a discrete set of values
  - Valid and invalid discrete values
  - Instances of each subclass
Boundary Testing

Given an integer \( x \), determine the absolute value of \( x \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>Valid</th>
<th>all values</th>
</tr>
</thead>
</table>

- A large number of errors tend to occur at boundaries of the input domain
  - Overflows
  - Comparisons (‘<‘ instead of ‘\(\leq\)’, etc.)
  - Missing emptiness checks (e.g., collections)
  - Wrong number of iterations

```java
int abs(int x) {
    if (0 <= x) return x;
    return -x;
}
```

Negative result for \( x == \text{Integer.MIN}\_\text{VALUE} \)
**Boundary Testing: Example**

- Select elements at the “edge” of each equivalence class (in addition to values in the middle)
  - Ranges: lower and upper limit
  - Empty sets and collections

<table>
<thead>
<tr>
<th>month</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Month with 28 or 29 days</td>
<td>month = 2</td>
</tr>
<tr>
<td>Months with 30 days</td>
<td>month ∈ {4, 6, 9, 11}</td>
</tr>
<tr>
<td>Months with 31 days</td>
<td>month ∈ {1, 3, 5, 7, 8, 10, 12}</td>
</tr>
<tr>
<td>Invalid</td>
<td>month &lt; 1 or month &gt; 12</td>
</tr>
</tbody>
</table>

There is only one value
Choose all values
Choose 1 and 12 plus one more
Choose MIN_VALUE, 0, 13, MAX_VALUE
Boundary Testing: Example (cont’d)

<table>
<thead>
<tr>
<th></th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard leap years</td>
<td>year mod 4 = 0 and year mod 100 ≠ 0</td>
</tr>
<tr>
<td>Standard non-leap years</td>
<td>year mod 4 ≠ 0</td>
</tr>
<tr>
<td>Special leap years</td>
<td>year mod 400 = 0</td>
</tr>
<tr>
<td>Special non-leap years</td>
<td>year mod 100 = 0 and year mod 400 ≠ 0</td>
</tr>
</tbody>
</table>

Choose for instance:
-200.004, -4, 4, 2012, 400.008

Choose for instance:
-200.003, -1, 1, 2011, 400.009

Choose for instance:
-200.000, 0, 2000, 400.000

Choose for instance:
-200.100, 1900, 400.100
Parameterized Unit Test for Leap Years

```csharp
[ Test ]
public void TestDemo29(
    [ Values( -200004, -200000, -4, 0, 4, 2000, 2012, 400000, 400008 ) ]
    int year )
{
    int d = Days( 2, year );
    Assert.IsTrue( d == 29 );
}
```

- Analogous test cases for February in non-leap year, months with 30 days, and months with 31 days
### Parameterized Unit Test for Invalid Inputs

```csharp
[Test]
[ExpectedException(typeof(ArgumentException))]

public void TestDemoInvalid(
    [Values(int.MinValue, 0, 13, int.MaxValue)] int month,
    [Values(-200100, -200004, -200003, -200000, -4, -1, 0, 1, 4, 1900, 2000, 2011, 2012, 400000, 400008, 400009, 400100)] int year)
{
    int d = Days(month, year);
}
```

- **Expected result:** an exception
- **All selected invalid values for month**
- **All selected values for year**
5. Testing

5.1 Test Stages

5.2 Test Strategies

5.3 Functional Testing

5.3.1 Partition Testing

5.3.2 Selecting Representative Values

5.3.3 Combinatorial Testing

5.4 Structural Testing
Combinatorial Testing

- Combining equivalence classes and boundary testing leads to many values for each input
  - Twelve values for month and 17 values for year in the Leap Year example

- Testing all possible combinations leads to a combinatorial explosion \((12 \times 17 = 204\) tests)

- Reduce test cases to make effort feasible
  - Random selection
  - Semantic constraints
  - Combinatorial selection
 Eliminating Combinations

- Inspect test cases for unnecessary combinations
  - Especially for invalid values
  - Use problem domain knowledge

<table>
<thead>
<tr>
<th>month</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Month with 28 or 29 days</td>
<td>month = 2</td>
</tr>
<tr>
<td>Months with 30 days</td>
<td>month ∈ {4, 6, 9, 11}</td>
</tr>
<tr>
<td>Months with 31 days</td>
<td>month ∈ {1, 3, 5, 7, 8, 10, 12}</td>
</tr>
<tr>
<td>Invalid</td>
<td>month &lt; 1 or month &gt; 12</td>
</tr>
</tbody>
</table>

- Reduces test cases from 204 to 17 + 4 + 3 + 4 = 28
Eliminating Combinations: NUnit Example

```csharp
[Test, Sequential]
[ExpectedException( typeof(ArgumentException) )]
public void TestDemoInvalid(
    [Values( int.MinValue, 0, 13, int.MaxValue )] int month,
    [Values( -200100, -200004, -200003, -200000 )] int year ) {
    int d = Days( month, year );
}
```

All selected invalid values for month

One value for year for each value for month
Selecting Object References

- Objects are different from values because they have identity

\[
a1 = \text{new Account( 1000 )};
\]
\[
a2 = \text{new Account( 1000 )};
\]
\[
a1.\text{transfer( a2, 500 );}
\]

- When selecting test data for objects, one has to consider object identities and aliasing
- Referenced objects lead to combination problem

\[
a1 = \text{new Account( 1000 )};
\]
\[
a1.\text{transfer( a1, 500 );}
\]

Might behave differently (e.g., deadlock)
Roots Example

Given three values, a, b, c, compute all solutions of the equation \( ax^2 + bx + c = 0 \)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>any value</td>
<td>any value</td>
<td>any value</td>
</tr>
<tr>
<td>Invalid</td>
<td>infinity, NaN</td>
<td>infinity, NaN</td>
<td>infinity, NaN</td>
</tr>
</tbody>
</table>

Boundary testing: 
\[
a, b, c \in \{ \text{Double.MIN\_VALUE}, -5, 0, 5, \text{Double.MAX\_VALUE} \}
\]

- \(5^3 = 125\) test cases for valid inputs
Roots Example (cont’d)

Given three values, \(a\), \(b\), \(c\), compute all solutions of the equation \(ax^2 + bx + c = 0\)

<table>
<thead>
<tr>
<th>Two solutions</th>
<th>One solution</th>
<th>No solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \neq 0) and (b^2 - 4ac &gt; 0)</td>
<td>(a = 0) and (b \neq 0) or (a \neq 0) and (b^2 - 4ac = 0)</td>
<td>(a = 0), (b = 0), and (c \neq 0) or (a \neq 0) and (b^2 - 4ac &lt; 0)</td>
</tr>
</tbody>
</table>

- Semantic constraints on combinations
- Partitioning seems too coarse
- Look at dependencies between inputs

Given three values, \(a\), \(b\), \(c\), compute all solutions of the equation \(ax^2 + bx + c = 0\)
Roots Example (cont’d)

Given three values, a, b, c, compute all solutions of the equation \( ax^2 + bx + c = 0 \)

<table>
<thead>
<tr>
<th></th>
<th>Two solutions</th>
<th>One solution</th>
<th>No solution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linear equation</strong></td>
<td></td>
<td>a = 0 and b ≠ 0</td>
<td>a = 0, b = 0, and c ≠ 0</td>
</tr>
<tr>
<td><strong>(Truly) quadratic equation</strong></td>
<td>a ≠ 0 and ( b^2-4ac &gt; 0 )</td>
<td>a ≠ 0 and ( b^2-4ac = 0 )</td>
<td>a ≠ 0 and ( b^2-4ac &lt; 0 )</td>
</tr>
</tbody>
</table>

Not all inputs are covered: a=b=c=0
Roots Example (cont’d)

Given three values, $a$, $b$, $c$, compute all solutions of the equation $ax^2 + bx + c = 0$; report an error if all three values are zero.

<table>
<thead>
<tr>
<th></th>
<th>Two solutions</th>
<th>One solution</th>
<th>No solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear equation</td>
<td></td>
<td>$a = 0$ and $b \neq 0$</td>
<td>$a = 0$, $b = 0$, and $c \neq 0$</td>
</tr>
<tr>
<td>(Truly) quadratic equation</td>
<td>$a \neq 0$ and $b^2-4ac &gt; 0$</td>
<td>$a \neq 0$ and $b^2-4ac = 0$</td>
<td>$a \neq 0$ and $b^2-4ac &lt; 0$</td>
</tr>
<tr>
<td>Invalid input</td>
<td></td>
<td>$a = 0$, $b = 0$, $c = 0$</td>
<td></td>
</tr>
</tbody>
</table>
Roots Example: Summary

- Classifying the combinations according to semantic constraints did not reveal any irrelevant test cases

- But we did identify an omission in the specification
  - It is common that testers clarify the specification

- One option is to manually choose a manageable number of test cases such that there is at least one test case for each semantic constraint
  - Note that omitting test cases might leave errors such as arithmetic overflow undetected
Semantic Constraints: Discussion

- Semantic constraints potentially reduce the number of test cases
  - They also help increasing the coverage

- But too many combinations remain
  - Especially when there are many input values, for instance, for the fields of objects
Influence of Variable Interactions

- Empirical evidence suggests that most errors do not depend on the interaction of many variables

<table>
<thead>
<tr>
<th>Vars</th>
<th>Medical Devices</th>
<th>Browser</th>
<th>Server</th>
<th>NASA GSFC</th>
<th>Network Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>66%</td>
<td>29%</td>
<td>42%</td>
<td>68%</td>
<td>20%</td>
</tr>
<tr>
<td>2</td>
<td>97%</td>
<td>76%</td>
<td>70%</td>
<td>93%</td>
<td>65%</td>
</tr>
<tr>
<td>3</td>
<td>99%</td>
<td>95%</td>
<td>89%</td>
<td>98%</td>
<td>90%</td>
</tr>
<tr>
<td>4</td>
<td>100%</td>
<td>97%</td>
<td>96%</td>
<td>100%</td>
<td>98%</td>
</tr>
<tr>
<td>5</td>
<td>99%</td>
<td>96%</td>
<td></td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>100%</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Interactions of two or three variables trigger most errors
Pairwise-Combinations Testing

- Instead of testing all possible combinations of all inputs, focus on all possible combinations of each pair of inputs
  - Pairwise-combinations testing is identical to combinatorial testing for two or less inputs

- Example: Consider a method with four boolean parameters
  - Combinatorial testing requires \(2^4 = 16\) test cases
  - Pairwise-combinations testing requires 5 test cases: \(TTTT, TFFF, FTFF, FFTF, FFFT\)

- Can be generalized to k-tuples (k-way testing)
Pairwise-Combinations Testing: Complexity

- For n parameters with d values per parameter, the number of test cases grows logarithmically in n and quadratic in d
  - Handles larger number of parameters, for instance, fields of objects
  - The number d can be influenced by the tester

- Result holds for large n and d, and for all k in k-way testing
Pairwise-Combinations Testing: Example

<table>
<thead>
<tr>
<th>Two solutions</th>
<th>One solution</th>
<th>No solution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a = 0 and b ≠ 0</td>
<td>a = 0, b = 0, and c ≠ 0</td>
</tr>
<tr>
<td>a ≠ 0 and b^2-4ac &gt; 0</td>
<td>a ≠ 0 and b^2-4ac = 0</td>
<td>a ≠ 0 and b^2-4ac &lt; 0</td>
</tr>
<tr>
<td></td>
<td>a = 0, b = 0, c = 0</td>
<td></td>
</tr>
</tbody>
</table>

- Three parameters, five values each
  - Double.MIN_VALUE, -5, 0, 5, Double.MAX_VALUE
  - 5^3 = 125 test cases for combinatorial testing
  - 25 test cases for pairwise-combinations testing

- Bug is still detected (depends only on a and b)
- Some cases depend on three parameters, e.g., invalid input
Pairwise-Combinations Testing: Discussion

- Pairwise-combinations testing (or k-way testing) reduces the number of test cases significantly while detecting most errors.
- Pairwise-combinations testing is especially important when many system configurations need to be tested.
  - Hardware, operating system, database, application server, etc.
- Should be combined with other approaches to detect errors that are triggered by more complex interactions among parameters.
Functional Testing: Summary

1. Functional Requirements
2. Independently Testable Feature
3. Representative Values
4. Test Case Specification
5. Test Cases

- Equivalence classes, boundary testing
- Exhaustive enumeration, semantic constraints, pairwise combinations
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
Motivating Example

Given a non-null array of integers, sort the array in-place in ascending order

```java
public void sort( int[ ] a ) {
    if( a == null || a.length < 2 ) // array is trivially sorted
        return;
    // check if array is already sorted
    int i;
    for( i = 0; i < a.length - 1; i++ )
        if( a[ i ] < a[ i + 1 ] )
            break;
    if( i >= a.length - 1 ) // array is already sorted
        return;
    // use quicksort to sort the array in ascending order
}
```

Error: check for sortedness should use ‘>’
Motivating Example: Functional Testing

Given a non-null array of integers, sort the array in-place in ascending order

<table>
<thead>
<tr>
<th></th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>any non-null array</td>
</tr>
<tr>
<td>Invalid</td>
<td>null</td>
</tr>
</tbody>
</table>

Choose for instance {}, { 1 }, { 1, 2, 3 }

- The requirements give no clue that one should test with an array that is sorted in descending order
Motivating Example: Discussion

- Detailed design and coding introduce many behaviors that are not present in the requirements
  - Choice of data structures
  - Choice of algorithms
  - Optimizations such as caches

- Functional testing generally does not thoroughly exercise these behaviors
  - No data structure specific test cases, e.g., rotation of AVL-tree
  - No test cases for optimizations, e.g., cache misses
Applications of Structural Testing

- White-box test a unit to cover a large portion of its code

Diagram:

- Unit Test
  - Integration Test
    - System Test

Use design knowledge to cover most of the code
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
   5.4.1 Control Flow Testing
   5.4.2 Advanced Topics of Control Flow Testing
   5.4.3 Data Flow Testing
   5.4.4 Interpreting Coverage
Basic Blocks

- A **basic block** is a sequence of statements such that the code in a basic block:
  - has **one entry point**: no code within it is the destination of a jump instruction anywhere in the program
  - has **one exit point**: only the last instruction causes the program to execute code in a different basic block

- Whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order
Basic Blocks: Example

```java
public void sort( int[] a ) {
    if (a == null || a.length < 2)
        return;
    int i;
    for (i = 0; i < a.length - 1; i++) {
        if (a[i] < a[i + 1])
            break;
    }
    if (i >= a.length - 1)
        return;
    qsort(a, 0, a.length);
}
```
Intraprocedural Control Flow Graphs

- An intraprocedural control flow graph (CFG) of a procedure $p$ is a graph $(N,E)$ where:
  - $N$ is the set of basic blocks in $p$ plus designated entry and exit blocks
  - $E$ contains:
    - an edge from $a$ to $b$ with condition $c$ iff the execution of basic block $a$ is succeeded by the execution of basic block $b$ if condition $c$ holds
    - an edge $(\text{entry}, a, \text{true})$ if $a$ is the first basic block of $p$
    - edges $(b, \text{exit}, \text{true})$ for each basic block $b$ that ends with a (possibly implicit) return statement
Control Flow Graphs: Example

\[\begin{align*}
    b_1 &= (a == \text{null} \lor a.\text{length} < 2); \\
    b_2 &= (i < a.\text{length} - 1); \\
    b_3 &= (a[i] < a[i + 1]); \\
    b_4 &= (i >= a.\text{length} - 1); \\
    i &= 0; \\
    \text{break}; \\
    \text{qsort}(a, 0, a.\text{length}); \\
    \text{return}; \\
\end{align*}\]
Test Coverage

- The CFG can serve as an **adequacy criterion** for test cases
- The more parts are executed, the higher the chance to uncover a bug
- “parts” can be nodes, edges, paths, etc.
Test Coverage: Example

- Consider the input
  \( \text{a} = \{3, 7, 5\} \)
Statement Coverage

- Assess the quality of a test suite by measuring how much of the CFG it executes

- Idea: one can detect a bug in a statement only by executing the statement

Statement Coverage = \[
\frac{\text{Number of executed statements}}{\text{Total number of statements}}
\]

- Can also be defined on basic blocks
Statement Coverage: Example

- Consider the input: \( a = \{ 3, 7, 5 \} \)
- This single test case executes 7 out of 10 basic blocks
- Statement coverage: 70%
5. Testing – Structural Testing

Statement Coverage: Example (cont’d)

- We can achieve 100% statement coverage with three test cases
  - a = { 1 }
  - a = { 5, 7 }
  - a = { 7, 5 }

- The last test case detects the bug

Zhendong Su – Rigorous Software Engineering
Statement Coverage: Discussion

```java
boolean contains( int[ ] a, int x ) {
    if( a == null ) return false;
    boolean found = false;
    for( int i = 0; i <= a.length; i++ ) {
        if( a[ i ] == x ) {
            found = true;
            break;
        }
    }
    return found;
}
```

5. Testing – Structural Testing
Statement Coverage: Discussion (cont’d)

- We can achieve 100% statement coverage with two test cases
  - `a = null`
  - `a = { 1, 2 }, x = 2`
- The test cases do not detect the bug!
- More thorough testing is necessary
Branch Coverage

- Idea: test all possible branches in the control flow

- An edge \((m, n, c)\) in a CFG is a branch iff there is another edge \((m, n', c')\) in the CFG with \(n \neq n'\)

\[
\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}
\]

- Conveniently define branch coverage to be 100% if the code contains no branches
Branch Coverage: Example 1

- Consider the input \( a = \{3, 7, 5\} \)
- This single test case executes 4 out of 8 branches
- Branch coverage: 50%
- Three test cases needed for 100% branch coverage
Branch Coverage: Example 2

- The two test cases
  - \( a = \text{null} \)
  - \( a = \{1, 2\}, x = 2 \)
execute 5 out of 6 branches

- Branch coverage: 83%
Achieving 100% branch coverage would require a test case that runs the loop to the end
- a = \texttt{null}
- a = \{ 1 \}, x = 1
- a = \{ 1 \}, x = 3

The last test case detects the bug
Branch Coverage: Discussion

- Branch coverage leads to more thorough testing than statement coverage
  - Complete branch coverage implies complete statement coverage
  - But “at least n% branch coverage” does not generally imply “at least n% statement coverage”

- Most widely-used adequacy criterion in industry
Branch Coverage: Discussion (cont’d)

```java
int[ ] reverse( int[ ] a ) {
    int j = a.length – 1;
    int[ ] res = new int[ a.length ];
    for( int i = 0; i < a.length; i++) {
        res[ j ] = a[ i ];
    }
    return res;
}
```

---

5. Testing – Structural Testing

---
Branch Coverage: Discussion (cont’d)

- We can achieve 100% branch coverage with one test case
  - a = \{ 1 \}
- The test case does not detect the bug!
- More thorough testing is necessary

```
j = a.length – 1;
res = new int[ a.length ];
i = 0;

res[ j ] = a[ i ];
i++;
```

```
return res;

entry

j = a.length – 1;
res = new int[ a.length ];
i = 0;

b1 = ( i < a.length );

b1

res[ j ] = a[ i ];
i++;

return res;

exit

-b1

```
int foo( boolean a, boolean b ) {
    int x = 1;
    int y = 1;
    if( a )
        x = 0;
    else
        y = 0;
    if( b )
        return 5 / x;
    else
        return 5 / y;
}
Branch Coverage: Discussion (cont’d)

- We can achieve 100% branch coverage with two test cases
  - $a = true$, $b = false$
  - $a = false$, $b = true$
- The test cases do not detect the bug!
- More thorough testing is necessary
Path Coverage

- Idea: test all possible paths through the CFG

- A path is a sequence of nodes $n_1, \ldots, n_k$ such that
  - $n_1 = \text{entry}$
  - $n_k = \text{exit}$
  - There is an edge $(n_i, n_{i+1}, c)$ in the CFG

\[
\text{Path Coverage} = \frac{\text{Number of executed paths}}{\text{Total number of paths}}
\]
Path Coverage: Example 1

- The two test cases
  - \(a = \text{true}, \ b = \text{false}\)
  - \(a = \text{false}, \ b = \text{true}\)

execute two out of four paths

- Path coverage: 50%
Path Coverage: Example 1 (cont’d)

- We can achieve 100% path coverage with four test cases
  - \( a = \text{true}, b = \text{false} \)
  - \( a = \text{false}, b = \text{true} \)
  - \( a = \text{true}, b = \text{true} \)
  - \( a = \text{false}, b = \text{false} \)

- The two additional test cases detect the bugs

\[
\begin{align*}
\text{entry} & \quad x = 1; \\
& \quad y = 1; \\
& \quad b1 = a; \\
& \quad b2 = b; \\
& \quad \text{b1} \quad \neg b1 \\
& \quad \text{b2} \quad \neg b2 \\
& \quad \text{return } 5 / x; \\
& \quad \text{return } 5 / y; \\
\text{exit} &
\end{align*}
\]
Path Coverage: Example 2

```java
boolean contains(int[] a, int x) {
    if (a == null) return false;
    boolean found = false;
    for (int i = 0; i <= a.length; i++) {
        if (a[i] == x) {
            found = true;
            break;
        }
    }
    return found;
}
```
Path Coverage: Example 2 (cont’d)

- Number of loop iterations is not known statically (depends on input)

- An arbitrarily large number of test cases is needed for complete path coverage

```
b1 = ( a == null );
b2 = ( i <= a.length );
b3 = ( a[i] == x );

found = false;
i = 0;
i++;

found = true;
break;
return found;

return false;

entry
```

5. Testing – Structural Testing
Path Coverage: Discussion

- Path coverage leads to more thorough testing than both statement and branch coverage
  - Complete path coverage implies complete statement coverage and complete branch coverage
  - But “at least n% path coverage” does not generally imply “at least n% statement coverage” or “at least n% branch coverage”

- Complete path coverage is not feasible for input-dependent loops
  - Unbounded number of paths
Branch Coverage: Discussion (cont’d)

```java
int[] reverse(int[] a) {
    int j = a.length - 1;
    int[] res = new int[a.length];
    for (int i = 0; i < a.length; i++) {
        res[j] = a[i];
    }
    return res;
}
```

```
entry

j = a.length - 1;
res = new int[a.length];
i = 0;
```

```
b1 = (i < a.length);
```

```
res[j] = a[i];
i++;
```

```
return res;
```

```
exit
```
Loop Coverage

- Idea: for each loop, test zero, one, and more than one (consecutive) iterations

\[
\text{Loop Coverage} = \frac{\text{Number of executed loops with 0, 1, and more than 1 iterations}}{\text{Total number of loops} \times 3}
\]

- Loop coverage is typically combined with other adequacy criteria such as statement or branch coverage
Loop Coverage: Example

- The test case
  - \( a = \{ 1 \} \)
  executes one out of three possible cases for the loop

- Loop coverage: 33%
Loop Coverage: Example

- We can achieve 100% loop coverage with three test cases
  - \( a = \{ \} \)
  - \( a = \{ 1 \} \)
  - \( a = \{ 1, 2 \} \)

- The last test case detects the bug

```java
j = a.length – 1;
res = new int[ a.length ];
i = 0;

for ( ; i < a.length ; i++) {
    res[ j ] = a[ i ];
    i++;
}

return res;
```
Measuring Coverage

- Coverage information is collected while the test cases execute.

- Use code instrumentation or debug interface to count executed basic blocks, branches, etc.

```java
int foo( boolean a, boolean b ) {
    int x = 1;  int y = 1;
    if( a ) {
        branchCovered[0] = true;  x = 0;
    } else {
        branchCovered[1] = true;  y = 0;
    }
    if( b ) {
        branchCovered[2] = true;
        return 5 / x;
    } else {
        branchCovered[3] = true;
        return 5 / y;
    }
}
```
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
  5.4.1 Control Flow Testing
  5.4.2 Advanced Topics of Control Flow Testing
  5.4.3 Data Flow Testing
  5.4.4 Interpreting Coverage
static <E> void filter(
  Collection<E> from,
  Filter<E> f,
  Collection<E> to ) {
if( from == null ) return;
Iterator<E> i = from.iterator();
while( i.hasNext() ) {
  E e = i.next();
  if( f.apply( e ) )
    to.add( e );
}
}
Dynamically-Bound Method Calls

- Intraprocedural CFGs treat method calls as simple statements
- Yet, calls invoke different code depending on the dynamic type of the receiver
- Testing should cover the possible behaviors

```java
static <E> void filter(
    Collection<E> from,
    Filter<E> f,
    Collection<E> to ) {
    if( from == null ) return;
    Iterator<E> i = from.iterator( );
    while( i.hasNext( ) ) {
        E e = i.next( );
        if( f.apply( e ) )
            to.add( e );
    }
}
```
Testing Dynamically-Bound Method Calls

- A dynamically-bound method call can be regarded as a case distinction on the type of the receiver.

```cpp
f.apply(e)
if (type(f) == Filter) 
f.Filter::apply(e);
else if (type(f) == NullFilter) 
f.NullFilter::apply(e);
else // type(f) == Duplicates 
f.Duplicates::apply(e);
```

- Now we can apply branch testing.
Testing Dynamically-Bound Calls (cont’d)

- Treating dynamically-bound method calls as branches leads to a combinatorial explosion.

- Use semantic constraints and pairwise-combinations testing.

```java
static <E> void filter(
    Collection<E> from,
    Filter<E> f,
    Collection<E> to ) {
    if( from == null ) return;
    Iterator<E> i = from.iterator();
    while( i.hasNext() ) {
        E e = i.next();
        if( f.apply( e ) )
            to.add( e );
}
```
Exceptions

```
static <E> void filter(
    Collection<E> from,
    Filter<E> f,
    Collection<E> to ) {
    if( from == null ) return;
    if( f == null || to == null )
        throw new
            IllegalArgumentException( );
    Iterator<E> i = from.iterator( );
    while( i.hasNext( ) ) {
        E e = i.next( );
        if( f.apply( e ) )
            to.add( e );
    }
}
```
CFG: Exceptions

- Exceptions add a control flow edge from the basic block where the exception is thrown to the exit block or the block where the exception is caught.

- Idea: Cover exceptional control flow like normal control flow during testing.
  - Test oracle is checked when method terminates normally.

```csharp
[Test]
[ExpectedException(typeof(ArgumentException))]
public void TestDemoInvalid(…) {
    int d = Days(month, year);
}
```
Example: Documented Exceptions

```java
static <E> void filter(
    Collection<E> from,
    Filter<E> f,
    Collection<E> to ) {
    if( from == null ) return;
    if( f == null || to == null )
        throw new IllegalArgumentException( );
    Iterator<E> i = from.iterator( );
    while( i.hasNext( ) ) {
        E e = i.next( );
        if( f.apply( e ) )
            to.add( e );
    }
}
```

Might throw:
- UnsupportedOperationException
- ClassCastException
- NullPointerException
- IllegalArgumentException
- IllegalStateException

Might throw:
- NoSuchElementException
- UnsupportedOperation
- ClassCastException
- NullPointerException
- IllegalArgumentException
- IllegalStateException
Example: Documented Exceptions (cont’d)

- `b1 = (from == null);`
- `b2 = (f == null || to == null);`
- `throw new IllegalArgumentException();`
Example: Undocumented Exceptions

```java
static <E> void filter(
    Collection<E> from,
    Filter<E> f,
    Collection<E> to ) {
  if( from == null ) return;
  if( f == null || to == null )
    throw new
        IllegalArgumentException( );
  Iterator<E> i = from.iterator( );
  while( i.hasNext( ) ) {
    E e = i.next( );
    if( f.apply( e ) )
      to.add( e );
  }
}
```

The example might also throw:

- ConcurrentModificationException
- NoClassDefFoundError
- NoSuchMethodError
- OutOfMemoryError
- StackOverflowError
- ThreadDeath
- VirtualMachineError
- etc.
Example: Undocumented Exceptions (cont’d)

```
iterator<e> i = from.iterator();
b1 = ( from == null );
b2 = ( f == null || to == null );

if (!b1) {
    throw new IllegalArgumentException();
}

if (!b2) {
    Iterator<E> i = from.iterator();
b3 = i.hasNext();
e = i.next();
b4 = f.apply(e);
to.add(e);
}

b3 = i.hasNext();
e = i.next();
b4 = f.apply(e);
to.add(e);
exit
```

It is impractical to represent and test all exceptional control flow in the CFG.
Checked vs. Unchecked Exceptions

- Some programming languages distinguish between checked and unchecked exceptions

- **Checked exceptions** represent invalid conditions outside the immediate control of the program
  - Invalid user input, database problems, network outages, absent files

- **Unchecked exceptions** represent defects in the program or the execution environment
  - Illegal arguments, null-pointer dereferencing, division by zero, assertion violation, etc.
  - In Java: all subclasses of RuntimeException and Error
Testing Unchecked Exceptions

- Unchecked exceptions are not supposed to occur
- When computing the CFG, ignore unchecked exceptions thrown by other methods and virtual machine
  - But consider throw statements
Unchecked Exceptions: Bad Example

```java
static boolean contains( String[] a, String s ) {
    if ( a == null || s == null )
        throw new IllegalArgumentException;
    for ( int i = 0; i < a.length; i++ ) {
        try {
            if ( a[ i ].equals(s) )
                return true;
        } catch ( NullPointerException e ) {
            i++;
        }
    }
    return false;
}
```

- Never use unchecked exceptions to encode control flow!
Bad Example Fixed

```java
static boolean contains( String[] a, String s ) {
    if( a == null || s == null )
        throw new IllegalArgumentException( );
    for( int i = 0; i < a.length; i++ ) {
        if( a[ i ] != null ) {
            if( a[ i ].equals(s) )
                return true;
        } else {
            i++;
        }
    }
    return false;
}
```

Normal control flow will be covered

Bug will be detected
Testing Checked Exceptions

- Checked exceptions represent regular control flow that needs to be tested
  - Include control flow in CFG, testing, and coverage

- In Java, checked exceptions are declared in method signatures

```java
interface RemoteBuffer extends Remote {
    void put( String s ) throws RemoteException;
}
```

- For each call, add appropriate control flow edges
Checked Exceptions: Example

class Producer {
    RemoteBuffer b;

    void produce() throws RemoteException {
        boolean retried = false;
        boolean success = false;
        while (!success) {
            try {
                b.put( "Product" );
                success = true;
            } catch (RemoteException e) {
                if (retried) throw e;
            }
        }
    }
}
Testing Exceptions: Summary

- Checked exceptions encode the program’s reaction to invalid conditions in the environment
  - Test like normal control flow
- Unchecked exceptions represent defects
  - Test unchecked exceptions explicitly thrown by method under test (argument validation, precondition check)
  - Unchecked exceptions thrown by methods being called indicate defect in method under test (precondition violation) or in the called method
  - Unchecked exceptions thrown by virtual machine indicate defect in method under test (e.g., infinite recursion) or deployment error (e.g., class not found)
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
  5.4.1 Control Flow Testing
  5.4.2 Advanced Topics of Control Flow Testing
  5.4.3 Data Flow Testing
  5.4.4 Interpreting Coverage
Example Revisited

```java
int foo( boolean a, boolean b ) {
    int x = 1;
    int y = 1;
    if( a )
        x = 0;
    else
        y = 0;
    if( b )
        return 5 / x;
    else
        return 5 / y;
}
```
Data Flow Testing

- Testing all paths is not feasible
  - Number grows exponentially in the number of branches
  - Loops

- Idea: Test those paths where a computation in one part of the path affects the computation of another
Variable Definition and Use

- A **variable definition** for a variable v is a basic block that assigns to v
  - v can be a local variable, formal parameter, field, or array element

- A **variable use** for a variable v is a basic block that reads the value from v
  - In conditions, computations, output, etc.
Definition-Clear Paths

- A **definition-clear path** for a variable $v$ is a path $n_1, \ldots, n_k$ in the CFG such that:
  - $n_1$ is a variable definition for $v$
  - $n_k$ is a variable use for $v$
  - No $n_i$ ($1 < i \leq k$) is a variable definition for $v$ (unless each assignment to $v$ occurs after a use)

- Note: definition-clear paths do not go from entry to exit (in contrast to our earlier definition of path)
Definition-Use Pairs

- A definition-use pair for a variable $v$ is a pair of nodes $(d,u)$ such that there is a definition-clear path $d, \ldots, u$ in the CFG.

- We say **DU-pair** for definition-use pair.
Definition-Use Pairs: Examples

```
entry
x = 1;
y = 1;
b1 = a;
```

```
entry
x = 1;
y = 1;
b1 = a;
```

```
x = 0;
return 5 / x;
exit
```

```
x = 0;
return 5 / x;
exit
```

```
y = 0;
```

```
y = 0;
```

```
b2 = b;
```

```
b2 = b;
```

```
¬b1
```

```
¬b1
```

```
¬b2
```

```
¬b2
```

```
x = 1;
y = 1;
return 5 / y;
```

```
x = 1;
y = 1;
return 5 / y;
```

```
x = 0;
return 5 / y;
exit
```

```
x = 0;
return 5 / y;
exit
```
DU-Pairs Coverage

- Idea: test all paths that provide a value for a variable use

\[
\text{DU-Pairs Coverage} = \frac{\text{Number of executed DU-Pairs}}{\text{Total number of DU-Pairs}}
\]
DU-Pairs Coverage: Example

- The two test cases
  - $a = \text{true}, b = \text{false}$
  - $a = \text{false}, b = \text{true}$

achieve 100% branch coverage, but only 50% DU-pairs coverage

- In this example, DU-pairs coverage is equivalent to path coverage
Determining all DU-Pairs

- DU-Pairs are computed using a static *reaching-definitions* analysis
- For each node n and for each variable v, compute all variable definitions for v that possibly reach n via a definition-clear path

- The reaching definitions at a node n are:
  - The reaching definitions of n’s predecessors in the CFG
  - minus the definitions killed by one of n’d predecessors
  - plus the definitions made by one of n’d predecessors
Reaching Definitions: Algorithm

- **Input**
  - \( \text{pred}( n ) = \{ m \mid (m,n,c) \text{ is an edge in the CFG} \} \)
  - \( \text{succ}( m ) = \{ n \mid (m,n,c) \text{ is an edge in the CFG} \} \)
  - \( \text{gen}( n ) = \{ v_n \mid n \text{ is a variable definition for } v \} \)
  - \( \text{kill}( n ) = \{ v_m \mid n \text{ is a variable definition for } v \text{ and } m \neq n \} \)

- **We compute via fixpoint iteration**
  - \( \text{Reach}( n ) \): The reaching definitions at the beginning of \( n \)
  - \( \text{ReachOut}( n ) \): The reaching definitions at the end of \( n \)
Reaching Definitions: Algorithm (con’t)

```plaintext
foreach node n do
    ReachOut( n ) := ∅
end
worklist := nodes
while worklist ≠ ∅ do
    n := any( worklist )
    remove n from worklist
    Reach( n ) := \( \bigcup_{m \in \text{pred}(n)} \text{ReachOut}( m ) \)
    ReachOut( n ) := Reach( n ) \ kill( n ) \cup \text{gen}( n )
    if ReachOut( n ) has changed then
        worklist := worklist \cup \text{succ}( n )
    end
end
```
Reaching Definitions: Example

```plaintext
entry

1: x = 1;
y = 1;

2: b1 = a;

3: x = 0;

4: y = 0;

5: b2 = b;

6: return 5 / x;

7: return 5 / y;

exit

<table>
<thead>
<tr>
<th>n</th>
<th>Reach(n)</th>
<th>ReachOut(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\emptyset)</td>
<td>(x_1, y_1)</td>
</tr>
<tr>
<td>2</td>
<td>(x_1, y_1)</td>
<td>(x_1, y_1)</td>
</tr>
<tr>
<td>3</td>
<td>(x_1, y_1)</td>
<td>(x_3, y_1)</td>
</tr>
<tr>
<td>4</td>
<td>(x_1, y_1)</td>
<td>(x_1, y_4)</td>
</tr>
<tr>
<td>5</td>
<td>(x_1, x_3, y_1, y_4)</td>
<td>(x_1, x_3, y_1, y_4)</td>
</tr>
<tr>
<td>6</td>
<td>(x_1, x_3, y_1, y_4)</td>
<td>(x_1, x_3, y_1, y_4)</td>
</tr>
<tr>
<td>7</td>
<td>(x_1, x_3, y_1, y_4)</td>
<td>(x_1, x_3, y_1, y_4)</td>
</tr>
</tbody>
</table>
```
From Reaching Definitions to DU-Pairs

- The set of DU-pairs is easily determined as 
  \[ \{ (d,u) \mid u \text{ is a variable use for } v \text{ and } v_d \in \text{Reach}(u) \} \]

- DU-pairs for x: 
  (1,6), (3,6)

- DU-pairs for y: 
  (1,7), (4,7)
Data Flow Testing Example

- Convert character sequence to integer
  - Input format: \( d_{\text{dec}}^* | 'x'(d_{\text{hex}}^*) \), where \( d \) is a (decimal or hexadecimal) digit

```java
static int convert( char[] a ) {
    int base;  int i = 0;  int val = 0;
    if ( a.length == 0 ) return 0;
    if ( a[ i ] == 'x' ) {
        base = 12;  i = i + 1;
    } else {
        base = 10;
    }
    while ( i < a.length ) {
        val = val * base + Character.digit( a[ i ], base );
        i = i + 1;
    }
    return val;
}
```

We assume here that all inputs are of the required format.
Data Flow Testing Example: CFG

```
entry

1: i = 0;
val = 0;
b1 = ( a.length == 0 );
¬b1

2: b2 = ( a[i] == 'x' );
b2 ¬b2

4: base = 12;
i = i + 1;

5: base = 10;

6: b3 = ( i < a.length );
¬b3

7: val = val * base + Character.digit( a[i], base );
i = i + 1;

8: return val;

3: return 0;

exit
```
Data Flow Testing Example: DU-Pairs

- We get 14 DU-pairs

- DU-pairs for i:
  (1,2), (1,4), (1,6), (4,6), (7,6), (1,7), (4,7), (7,7)

- DU-pairs for val:
  (1,7), (7,7), (1,8), (7,8)

- DU-pairs for base:
  (4,7), (5,7)
Data Flow Testing Example: Bug

- Consider the test cases
  - \( a = \{ \} \)
  - \( a = \{ 'x' \} \)
  - \( a = \{ '1' \} \)
  - \( a = \{ '1', '2' \} \)

- The bug is not detected!

- Branch and loop coverage: 100%
- DU-pairs missed: (4,7) for \( i \), base (coverage 86%)
Data Flow Testing Example: Observation

- DU-pairs for i and val include (7,7)
- Complete DU-pairs coverage requires more than one loop iteration

```java
static int convert( char[] a ) {
    int base; int i = 0; int val = 0;
    if ( a.length == 0 ) return 0;
    if( a[ i ] == 'x' ) { base = 16; i = i + 1; }
    else { base = 10; }
    while( i < a.length ) {
        val = val * base + Character.digit( a[ i ], base );
        i = i + 1;
    }
    return val;
}
```
Determining all DU-Pairs: Heap Structures

- Determining whether a definition and a usage refer to the same heap location, a static analysis would need arithmetic and aliasing information.

- Static analysis has to over-approximate

```java
static void repeat( int[] from, int[] to ) {
    int i = 0;
    if ( from.length == 0 ) return;
    while ( i < to.length ) {
        to[ i ] = to[ i ] + from[ i % from.length ];
        i = i + 1;
    }
}
```
Measuring DU-Pairs Coverage

- Keep track of currently active definitions
  - defCover: Variable $\rightarrow$ Block

- Keep track of executed DU-pairs
  - useCover: Variable $\times$ Block$^{\text{def}}$ $\times$ Block$^{\text{use}}$ $\rightarrow$ $\mathbb{N}$

- Maps can be encoded as arrays, indexed by identifiers for variables and basic blocks
Measuring DU-Pairs Coverage: Example

```
int foo( boolean a, boolean b ) {
    int x = 1;  defCover[ “x” ] = 0;
    int y = 1;  defCover[ “y” ] = 0;
    if( a ) {
        x = 0;  defCover[ “x” ] = 1;
    } else {
        y = 0;  defCover[ “y” ] = 2;
    }
    if( b ) {
        useCover[ “x”, defCover[ “x” ], 3 ]++;  
        return 5 / x;
    } else {
        useCover[ “y”, defCover[ “y” ], 4 ]++;  
        return 5 / y;
    }
}
```

Current variable definition for x is basic block 0

Current variable definition for x is basic block 1

DU-pair for variable x with current definition and use-block 3 has been executed
Data Flow Testing: Discussion

- Data flow testing complements control flow testing
  - Choose test cases that maximize branch and DU-pairs coverage

- Like with path coverage, not all DU-pairs are feasible
  - Static analysis over-approximates data flow
  - Complete DU-pairs coverage might not be possible
Data Flow Testing: Discussion (cont’d)

- DU-pairs coverage is not the only adequacy criterion for data flow testing
  - All definitions, all predicate-usages, all simple-DU-paths, etc.

- DU-pair “anomalies” may point to errors
  - Use before definition (not possible for locals in Java)
  - Double definition without use
  - Termination after definition without use
5. Testing

5.1 Test Stages
5.2 Test Strategies
5.3 Functional Testing
5.4 Structural Testing
  5.4.1 Control Flow Testing
  5.4.2 Advanced Topics of Control Flow Testing
  5.4.3 Data Flow Testing
  5.4.4 Interpreting Coverage
Interpreting Coverage

- High coverage does not mean that code is well tested
  - But: low coverage means that code is not well tested
  - Make sure you do not blindly optimize coverage but develop test suites that test the code well

- Coverage tools do not only measure coverage metrics, they also identify which parts of the code have not been tested
Experimental Evaluation: Approach

- Several studies investigate the benefit of coverage metrics

- Approach
  - Seed defects in the code
  - Develop test suites that satisfy various coverage criteria
  - Measure how many of the seeded defects are found by the test suits
  - Extrapolate to “real” defects in the code
Experimental Evaluation: Some Findings

- The test suite size grows exponentially in the coverage.
- More demanding coverage criteria lead to larger test suites, but do not detect more bugs.
  - Block, decision, data flow coverage.
- There is no significant difference in the cost-efficiency of the various coverage metrics.
- All adequacy criteria lead to test suites that detect more bugs than random testing, especially for large test suites.