CSE 403 Software Engineering

Testing

Today's outline

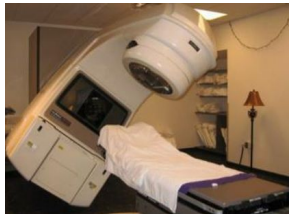
Software testing

- Motivating examples
- Categories of tests
- Unit testing

Could better testing have helped …

Therac-25 radiation therapy machine (1985-87)

- Device to create high energy beams to destroy tumors with minimal impact on surrounding healthy tissue
- Caused excessive radiation in some situations
- What happened?
  - An update removed hardware interlocks that prevented the electron-beam from operating in its high-energy mode.
    So all the safety checks were done in the software.
  - The software set a flag variable by incrementing it. Occasionally an arithmetic overflow occurred, causing the software to bypass safety checks.
  - The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred if the operator changed the setup too quickly.
- More deaths from a radiation machine in Panama (2006)



Cost: (at least) death in 6 patients

Therac-25 - Wikipedia

Ariane 5 rocket (1996)





Cost of program: over €1 billion

- European heavy-lift space launch vehicle - self-destructed 37 seconds after launch
- What happened?
  - A control software defect went undetected –
    - Conversion from 64-bit floating point to 16-bit signed integer caused an exception
    - Floating-point number was larger than 32767 (max 16-bit signed integer), overflow
  - Efficiency considerations had led to the disabling of the exception handler
  - Program crashed □ rocket crashed

Ariane 5 - Wikipedia

Mars Polar Lander (1999)



Cost of program: $165 million

- NASA robotic spacecraft, created to study the soil and climate of a region on Mars
- After descent phase, lander failed to reestablish communication with Earth
- What (most likely) happened?
  - Sensor signal falsely indicated that the craft had touched down when it was 130-feet above the surface and the descent engines to shut down prematurely
  - The error was traced to a single bad line of software code

Mars Polar Lander - Wikipedia

Northeast blackout (2003)

- Defect in alarm system prevented notification about overloaded power lines.
- Cascading failures.
- 55 million people affected
- ~100 deaths



Knight Capital Group (2012)

- Market maker for stock trading
- Changed software, one host didn't get part of the update
- Lost $440m in 45 minutes

WannaCry Ransomware Attack (2017)

- Cryptoworm infecting computers, encrypting their data, and demanding ransom payments
- Estimated to have affected more than 200,000 computers across 150 countries
- What happened?
  - WannaCry exploited a defect in the Server Message Block (SMB) protocol
  - MSFT provided a patch earlier, but many customers hadn't installed it yet



Cost of exploit: 100s of millions to billions of $

NHS - 70,000 hospital devices were impacted

WannaCry ransomware attack - Wikipedia

Quality assurance

Reasons to avoid defects:

- engineering craftsmanship
- cost of failures (sometimes large, but every failure matters)

Testing is an essential, effective way to discover failures

- combine with other techniques: linting, code reviews, proofs, …

Respond at **pollev.com/cse403au**

Text **CSE403AU** to **22333** once to join, then **A, B, C, D, or E**

What is the top most dangerous type of bug reported in 2023?

- Integer Overflow
- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Null Pointer Dereference
- Out-of-bounds Write
- Out-of-bounds Read

Total Results: 0

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Which is the most dangerous error?

- Integer overflow
- SQL injection
- Cross-site scripting
- Null pointer dereference
- Out-of-bounds read
- Out-of-bounds write

SQL injection

Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```


SQL injection

Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```

If the user enters "mernst", the value of `query` is:

SQL injection

Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```

If the user enters "mernst", the value of `query` is:

```
"select from table where user='mernst'"
```


SQL injection

Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```

If the user enters "mernst", the value of `query` is:

```
"select from table where user='mernst'"
```

What if a user enters, as their username: ' or ''='

The value of `query` is:

SQL injection

Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```

If the user enters "mernst", the value of **query** is:

```
  "select from table where user='mernst'"
```

What if a user enters, as their username: ' or ''='

The value of **query** is:

```
  "select from table where user='' or ''=''"
```


SQL injection

Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```

If the user enters "mernst", the value of **query** is:

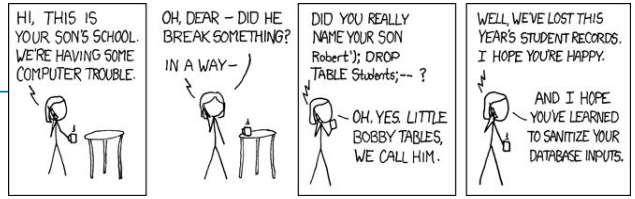
```
  "select from table where user='mernst'"
```

What if a user enters, as their username: ' or ''='

The value of **query** is:

```
  "select from table where user='' or ''=''"
```


SQL injection



Assume this code:

```
String query =
  "select from table where user='" + username + "'";
```

If the user enters "mernst", the value of **query** is:

```
  "select from table where user='mernst'"
```

What if a user enters, as their username: ' or ''='

The value of **query** is:

```
  "select from table where user='' or ''=''"
```


And the data says …

[2023 CWE Top 25 Most Dangerous Software Weaknesses](mitre.org)

Errors identified as root cause of reported vulnerabilities
NVD – Home (nist.gov)

2023 CWE Top 25

Rank	ID	Name	Score	CVEs in KEV	Rank Chang vs. 2022
1	CWE-787	Out-of-bounds Write	63.72	70	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54	4	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27	6	0
4	CWE-416	Use After Free	16.71	44	+3
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	15.65	23	+1
6	CWE-20	Improper Input Validation	15.50	35	-2
7	CWE-125	Out-of-bounds Read	14.60	2	-2
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.11	16	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.73	0	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.41	5	0
11	CWE-862	Missing Authorization	6.90	0	+5
12	CWE-476	NULL Pointer Dereference	6.59	0	-1
13	CWE-287	Improper Authentication	6.39	10	+1

So let's test! Four categories of testing

1. Unit Testing
   - Does each module do what it is supposed to do in isolation?
2. Integration Testing
   - Do you get the expected results when the parts are put together?
3. Validation Testing
   - Does the program satisfy the requirements?
4. System Testing
   - Does the program work as a whole and within the overall environment? (includes full integration, performance, scale, etc.)

What are other testing-related terms?

Let's see if we can name at least 10:

- Regression testing
- Black box, white box testing
- Code coverage testing
- Boundary case testing
- Test-driven development
- Mutation testing
- Fuzz testing
- Performance testing
- Usability testing
- Acceptance testing



Testing vs. debugging

Testing: is there a defect?

Debugging: where is the defect? how to fix the defect?

Regression testing

> Fool me once, shame on you.
> Fool me twice, shame on me.
>
> Proverb

- Whenever you find a defect
  - Create a new test
    - Store the input that triggered that defect, plus the correct output
    - Add these to the test suite
  - Verify that the test fails
  - Fix the defect
  - Verify the fix
- Ensures that your fix solves the problem
- Helps to populate test suite with good tests
- Protects against reversions that reintroduce the defect
  - It happened at least once, and it ~~might~~ will happen again

Also look for similar defects elsewhere!

How else can we build in quality?

What can we do beyond testing?
Hint: build in quality from the start ☺

- Good architecture, design, and planning
- Coding style guides, linting
- Code reviews
- Atomic commits
- Pair programming
- …

Today's outline

Software testing
- Motivating examples
- Categories of tests
- **Unit testing** ← We are here
  - Black box testing
    - Boundary case testing
    - Test driven development
  - White-box testing
    - Static code analysis
    - Code coverage testing

Unit testing

- A **unit** is an **independently testable part** of the software system (e.g. a procedure, class, or module).
- **Goal**: Verify that each software unit performs as specified.
- **Focus**:
  - Individual units (not the interactions between units).
  - Usually input/output relationships.

Testing best practices: motivating example 1

Average of the absolute values of an array of doubles

```
public double avgAbs(double ... numbers) {

  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException("Array numbers must not be null or empty!");
  }

  double sum = 0;
  for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
      sum -= d;
    } else {
      sum += d;
    }
  }

  return sum/numbers.length;
}
```

What tests should we write for this method?

Two possible specifications

Average of the absolute values of an array of doubles

```
/** Returns the average of the absolute values of the given numbers.
    @param numbers a non-null, non-empty array of doubles */
public double avgAbs(double... numbers) {
  ...


/** Returns the average of the absolute values of the given numbers.
    @param numbers an array of doubles
    @throws IllegalArgumentException if numbers is null or empty */
public double avgAbs(double ... numbers) {
  ...
```

Which specification is better?

Testing best practices: motivating example 2

Compare two values

```
public class Comp implements Comparable<Comp> {
  private int number;
  public Comp(int number) {this.number = number;}

  @Override
  public int compareTo(Comp other) {
    if (other.number == this.number) return 0;
    return this.number < other.number ? -1 : 1;
  }
}

public class CompTest {
  @Test
  public void testSmaller() {
    Comp c1 = new Comp(10);
    Comp c2 = new Comp(20);
    assertEquals(c1.compareTo(c2), -1);
  }
}
```

What's wrong with this test? (Note: the compareTo implementation is correct.)

Two types of unit testing

Black box testing
Written without knowledge of the code
Treats the module/system as atomic
Best simulates the customer experience

White box testing
Written with knowledge of the code
Examines the module/system internals
Aims to maximize a measure of test suite quality

Black-box testing

- Black-box is based on specifications (requirements and functionality), not code
- Tester does not look at the code while constructing the tests
- Work from the user or client's perspective

How do you know when you are done?

You have tested all the behaviors, according to the specification?

How do you know when you have tested all the behaviors?

What if the behavior differs from the specification?

Approach:

- build tests according to the text of the specification
  - "cover" the specification
- guess about what errors the programmer might have made
  - add more tests based on these guesses/heuristics

Black box: boundary case testing

Boundary case testing:

- What: test edge conditions
- Why?
  - #1 and #7 Most Dangerous Software Weakness!
  - Off-by-one defects are common (< vs. <=, etc.)
  - Requirement specs may be fuzzy about behavior on boundaries
  - Often uncovers internal hidden limits in code

Black box: boundary case example #1

- Write test cases based on paths through the **specification**

```
int find(int[] a, int value) throws Missing
// returns: the smallest i such that a[i] == value
// throws:  Missing if value not in a[]
```

Black box: boundary case example #1

- Write test cases based on paths through the **specification**

```
int find(int[] a, int value) throws Missing
// returns: the smallest i such that a[i] == value
// throws:  Missing if value not in a[]
```

- Two obvious tests:
  ([4, 5, 6], 5) => 1
  ([4, 5, 6], 7) => throw Missing
- Have we captured all the behaviors?

Black box: boundary case example #1

- Write test cases based on paths through the **specification**

```
int find(int[] a, int value) throws Missing
// returns: the smallest i such that a[i] == value
// throws:  Missing if value not in a[]
```

- Two obvious tests:
  ([4, 5, 6], 5) => 1
  ([4, 5, 6], 7) => throw Missing
- Have we captured all the behaviors?
  ([4, 5, 5], 5) => 1

Boundary case #2

```
<E> void appendList(List<E> src, List<E> dest) {
// modifies: src, dest
// effects:  removes all elements of src and appends them
//           in reverse order to the end of dest
```

What would be a good test in this case?

Boundary case #2 (aliases)

```
<E> void appendList(List<E> src, List<E> dest) {
// modifies: src, dest
// effects:  removes all elements of src and appends them
//           in reverse order to the end of dest
```

What would be a good test in this case?

Consider if src and dest are the same object

Testing aliasing is a good test!

Boundary case #3 (arithmetic overflow)

```
public int abs(int x)
// returns: |x|
```

- What are some values or ranges of x that might be worth probing?

Boundary case #3

```
public int abs(int x)
// returns: |x|
```

- What are some values or ranges of x that might be worth probing?
  - `x < 0, x ≥ 0`
  - `x = 0 (boundary condition)`
  - `Specific tests: say x = -1, 0, 1`

Boundary case #3 (arithmetic overflow)

```
public int abs(int x)
// returns: |x|
```

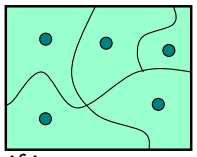
- What are some values or ranges of x that might be worth probing?
  - `x < 0, x ≥ 0`
  - `x = 0 (boundary condition)`
  - `Specific tests: say x = -1, 0, 1`
- How about…

```
int x = -2147483648;            // this is Integer.MIN_VALUE
System.out.println(x<0);        // true
System.out.println(Math.abs(x)<0); // also true!
```

Javadoc of **abs** says … if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative

Theory explains why boundary testing works

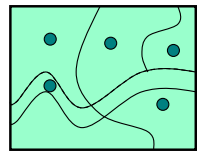
- Divide the input into **subdomains**
  - **A subdomain is a subset of possible inputs**
  - Identify input sets with same behavior
  - Try one input from each set



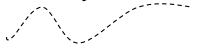
- A program has the "same behavior" on two inputs if it:
  1) gives correct result on both, or
  2) gives incorrect result on both (even if different incorrect result)
  - "Same behavior" is unknowable
  - A subdomain is revealing for an defect, E, if each test input fails (misbehaves)

If the program has a defect, it is revealed by a test in its revealing subdomain

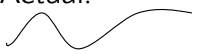
What if you mis-drew the boundaries?



What you drew:

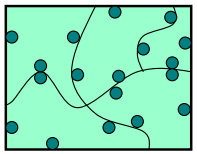


Actual:



Boundary case testing heuristic

- Create tests at the **boundaries** of subdomains

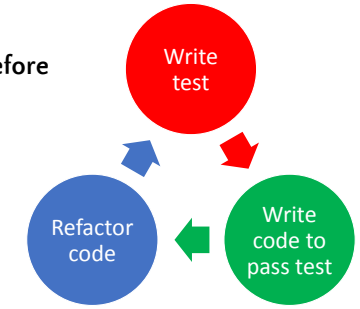


- Catches common boundary case defects:
  - **Arithmetic**
    - Smallest/largest values
    - Zero
  - **Objects**
    - Null
    - Circular
    - Same object passed to multiple arguments (aliasing)
  - **You drew the boundaries wrong**

Black box: test driven development

Test driven development (TDD):

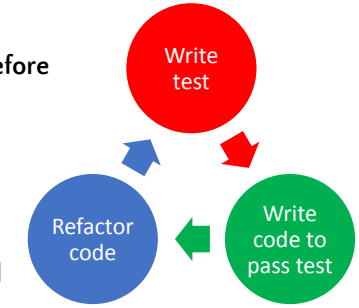
- What:
  - Test is based on the spec and is developed **before** the code is written
  - Will fail initially
  - Write just enough code to make it pass!
- Why?



Black box: test driven development

Test driven development (TDD):

- What:
  - Test is based on the spec and is developed **before** the code is written
  - Will fail initially
  - Write just enough code to make it pass!
- Why?
  - In practice, significantly less defect rate
  - Improved understanding of requirements and ability to influence design
  - Not influenced by implementation choices



Let's try it out with this avgAbs spec

```
double avgAbs(double ... numbers)
// Average of the absolute values of an array of doubles
```

TDD – what tests need to pass in order for us to sign off on the coding?

- `assertEquals(2.0, avgAbs({1.0, 2.0, 3.0}));`
- `assertEquals(2.0, avgAbs({1.0, -2.0, 3.0}));`
- `assertEquals(2.0, avgAbs({2.0}));`
- …

Let's try it out with this date spec

```
class Date
   • Date(int yyyy, int mm, int dd)
     // Creates date yyyy-mm-dd
   • boolean after(Date date1, Date date2)
     // Tests if date1 is after date2
   • Date subtractWeeks(Date date1, int numWks)
     // Subtracts numWks from date1
```

TDD – what tests need to pass in order for us to sign off on the coding?

TDD can result in a lot of tests!

• Develop tests now (TDD) or later – need to be judicious in which to write

Moving on to white box testing

Black box testing
Written without knowledge of the code
Treats the module/system as atomic
Best simulates the customer experience

White box testing
Written with knowledge of the code
Examines the module/system internals
Aims to maximize a measure of test suite quality

White (clear, glass) box testing

• Ultimate goal:
  Test suite covers (executes) all of the program
  Question: what does "all of the program" mean?
    We will address this question soon.

• Assumption:
  Test more behaviors => better test suite quality

• Benefit: tests features not described by specification
  • Control-flow details
  • Performance optimizations
  • Alternate algorithms for different cases

Motivation for white-box testing

```
boolean[] primeTable = new boolean[CACHE_SIZE];
boolean isPrime(int x) {
    if (x <= CACHE_SIZE) {
        return primeTable[x];
    } else {
        for (int i = 2; i < x/2; i++) {
            if (x % i==0) return false;
        }
        return true;
    }
}
```

Consider an important transition around *x* = CACHE_SIZE

White-box testing has advantages

- Greater coverage
  - Increases confidence in code quality
  - If tests cover all of the code in the program, are you confident it's error free?
- Insight into test cases
  - Which test inputs are likely to yield new information (and should be written)
- Can surface an important class of boundaries
  - Consider `CACHE_SIZE`
  - Need to check numbers on each side of `CACHE_SIZE`
    - `CACHE_SIZE-1`, `CACHE_SIZE`, `CACHE_SIZE+1`
  - If `CACHE_SIZE` is mutable, we may need to test with different `CACHE_SIZE`s

What about challenges?

White-box testing has disadvantages

- Focus on the code: miss incompatibilities with spec
- Focus on the algorithm: miss alternate implementations
- Groupthink: think like the coder

White (clear, glass) box testing

- Ultimate goal:
  Test suite covers (executes) all of the program
  Question: what does "all of the program" mean?
- Assumption:
  Test more behaviors => better test suite quality
- Benefit: tests features not described by specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms for different cases

White (clear, glass) box testing

- Ultimate goal:
  Test suite covers (executes) all of the program
  Question: what does "all of the program" mean?
- Assumption:
  Test more behaviors => better test suite quality
- Benefit: tests features not described by specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms for different cases

- every line of code
- every then and else clause
- every CMP instruction in the binary
- every input

White (clear, glass) box testing

- **Ultimate goal:**
  Test suite covers (executes) all of the program
  Question: what does "all of the program" mean?

- Assumption:
  Test more behaviors => better test suite quality

- Benefit: tests features not described by specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms for different cases

- every line of code
- every then and else clause
- every CMP instruction in the binary
- every input
- *every subdomain*

(Artificial) goal:
Maximize a measurement of the test suite

Digression: what matters about a test suite?

More on code coverage

code coverage: What fraction of the code is executed by the tests

- This is a metric of test suite quality

  - statement coverage – tries to execute every line (practical?)
  - path coverage – follow every distinct branch through code
  - condition coverage – every condition that leads to a branch
  - function coverage – invoke every function at least once

More on code coverage

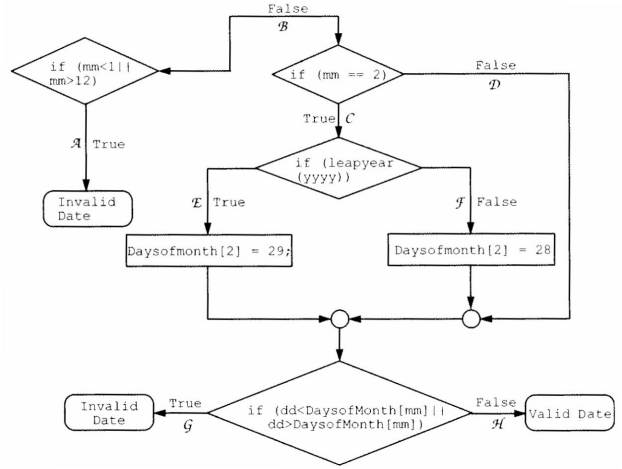
code coverage: What fraction of the code is executed by the tests

- This is a metric of test suite quality

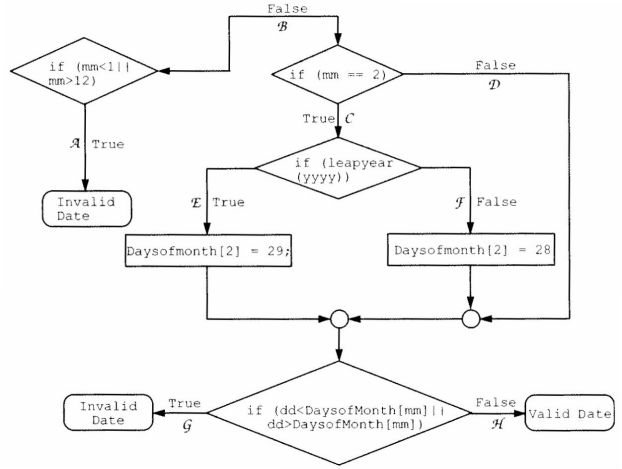
  - statement coverage – tries to execute every line (practical?)
  - path coverage – follow every distinct branch through code
  - condition coverage – every condition that leads to a branch
  - function coverage – invoke every function at least once

What about dead code?

Consider tests to cover all paths for the `Date` class



Consider tests to cover all paths for the `Date` class



You must use a code coverage tool for your project

How much coverage is enough? 100%?

May be subject to the law of diminishing returns.
Common advice: shoot for 80%.

ATLASSIAN

2. What percentage of coverage should you aim for?

There's no silver bullet in code coverage, and a high percentage of coverage could still be problematic if critical parts of the application are not being tested, or if the existing tests are not robust enough to properly capture failures upfront. With that being said it is generally accepted that 80% coverage is a good goal to aim for. Trying to reach a higher coverage might turn out to be costly, while not necessary producing enough benefit.

Good resource on code coverage and code coverage tools:
https://www.atlassian.com/continuous-delivery/software-testing/code-coverage
And a good list of coverage tools:
https://www.browserstack.com/guide/code-coverage-tools

Ending with some Rules of Testing

- First rule of testing: ***Do it early and do it often***
  - Best to catch defects soon, before they have a chance to hide
  - Automate, automate, automate the process
- Second rule of testing: ***Be systematic***
  - If you randomly thrash, defects will hide until you're gone
  - Writing tests is a good way to understand the spec
    - Think about revealing domains and boundary cases
    - If the spec is confusing, write more tests
  - Spec can be defective too
    - If you find incorrect, incomplete, ambiguous, and missing corner cases, fix it!
  - When you find a defect: write a regression test, fix it, look for similar defects