CSE 403 Software Engineering

Testing
Autumn 2023

Project document tips

Celebrate your brand – include a title in your doc
Make your content count – remove the assignment instructions
Today’s outline

Software testing

• Motivating examples
• Categories of tests
• Double click on 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.
  - And more ...

Cost of bugs: (at least) death in 6 patients

Therac-25 - Wikipedia
Ariane 5 rocket (1996)

• European heavy-lift space launch vehicle - self-destructed 37 seconds after launch
• What happened?
  • A control software bug 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

Cost of program: over €1 billion
Mars Polar Lander (1999)

• 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

Cost of program: $165 million

Mars Polar Lander - Wikipedia
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 bug in the Server Message Block (SMB) protocol
  - MSFT provided a security-patch earlier but many customers hadn’t installed it yet

Cost of exploit: 100s of millions to billions of $
It’s important – at times, critically important – to release quality software.

Examples showed particularly costly errors but every error adds up.

Many of the most common and impactful bugs can be caught with testing.
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
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
And the data says ...

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

Bugs identified as root cause of reported vulnerabilities
NVD - Home (nist.gov)

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

• Whenever you find a bug
  • Store the input that triggered that bug, plus the correct output
  • Add these to the test suite
  • Verify that the test suite fails
  • Fix the bug
  • Verify the fix

• Ensures that your fix solves the problem
• Helps to populate test suite with good tests
• Protects against reversions that reintroduce bug
  • It happened at least once, and it might will happen again

Fool me once, shame on you
Fool me twice, shame on me

Proverb
Time out: 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
• Code reviews/walkthroughs
• Atomic commits
• Pair programming
• ...
Software testing
• Motivating examples
• Categories of tests
  • Double click on unit testing
    • Black box testing
      • Boundary case testing
      • Test driven development
    • White box testing
      • Static code analysis
      • Code coverage testing

Unit Testing
Test that a method/class/module behaves as specified

We are here
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
- Trace data flow directly
- Bug report contains more detail on source of defect
Black-box testing

• Black-box is based on requirements and functionality, not code

• Tester may have actually seen the code before ("gray box")
  • But doesn't look at it while constructing the tests

• Often done from the end user or client's perspective

• Emphasis on parameters, inputs/outputs (and their validity)
Black box: boundary case testing

**Boundary case testing:**

- What: test edge conditions
- Why?
  - #1 and #7 Most Dangerous Software Weakness!
  - Likely source of programmer errors (\(<\ vs. \leq\), etc.)
  - Requirement specs may be fuzzy about behavior on boundaries
  - Often uncovers internal hidden limits in code
    - Example: array list must resize its internal array when it fills capacity
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 paths?
  
  ( [4, 5, 5], 5 ) => 1
Boundary case #2

```java
<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)

\[
\text{void appendList(List\text{\textless}E\text{\textgreater} src, List\text{\textless}E\text{\textgreater} dest) \{} \\
\text{// modifies: src, dest} \\
\text{// effects: removes all elements of src and appends them} \\
\text{// \hspace{1cm} in reverse order to the end of dest} \\
\]

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

```java
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)

```java
class Example {
    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 on `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
There are a lot of possible inputs!

- Consider input **subdomains**
  - Identify input sets with **same** behavior
  - Try one input from each set

- “Same” behavior depends on specification
  - Say that program has “same behavior” on two inputs if
    1) gives correct result on both, or
    2) gives incorrect result on both

- **A subdomain is a subset of possible inputs**
  Subdomain is **revealing** for an error, E, if
  1) Each element has **same behavior**
  2) If program has error E, it is revealed by test

Goal is to divide possible inputs into sets of **revealing subdomains** for various errors
Boundary case testing heuristic

- Create tests at the boundaries of subdomains
- Catches common boundary case bugs:
  - **Arithmetic**
    - Smallest/largest values
    - Zero
  - **Objects**
    - Null
    - Circular
    - Same object passed to multiple arguments (aliasing)
Test driven development (TDD):

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

• Why?
  • (Reported) significantly less defect rate
  • Improved understanding of requirements and ability to influence design
Let’s try it out with this `avgAbs` spec

```java
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**

```java
class Date
    • Date(int yyy, int mm, int dd)
        // Creates date dd/mm/yyyy
    • 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
Trace data flow directly
Bug report contains more detail on source of defect
White (clear, glass) box testing

• Ultimate goal:
  For test suite to cover (execute) all code of the program

• Assumption:
  High code coverage correlates with improved quality

• Focus:
  Features not described by specification, e.g.,
  • Control-flow details
  • Performance optimizations
  • Alternate algorithms for different cases

Static code analysis is one type of white-box testing
(Monday’s “build” class)

Test suite code coverage is another (today!)
Here’s a motivating example for WB testing

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

Consider an important transition around \( x = \text{CACHE\_SIZE} \)
White box testing has advantages

- Greater confidence in code quality
  - Correlating to greater amount of code covered by tests
  - If tests cover all of the code in the program, are you confident it’s error free?

- Insight into test cases
  - Which tests 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?
code coverage testing: Examines what fraction of the code under test is reached by existing unit tests

- statement coverage - tries to reach every line (practical?)
- path coverage - follow every distinct branch through code
- condition coverage - every condition that leads to a branch
- function coverage - treat every behavior / end goal separately

Dead code? A distraction? Or important?
Consider tests to cover all paths for the Date class.
Consider tests to cover all paths for the Date class.

Monday we’ll try using a code coverage tool together.
How much coverage is enough? 100%?

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

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 today with some Rules of Testing

• First rule of testing: *Do it early and do it often*
  Best to catch bugs soon, before they have a chance to hide
  Automate, automate, automate the process

• Second rule of testing: *Be systematic*
  If you randomly thrash, bugs 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 buggy too
    If you find incorrect, incomplete, ambiguous, and missing corner cases, fix it!
  When you find a bug, fix it + write a regression test for it