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)

Cost of program: over €1 billion

- 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

Ariane 5 - Wikipedia

Mars Polar Lander (1999)

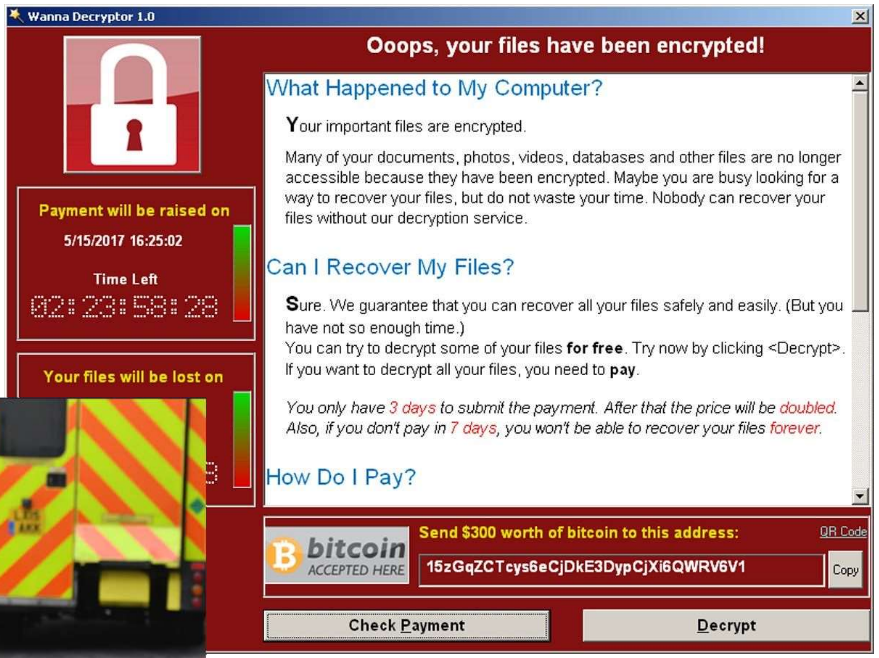
Cost of program:
$165 million

- 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

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



Ooops, your files have been encrypted!

Payment will be raised on

5/15/2017 16:25:02

Time Left

02:23:58:28

Your files will be lost on

What Happened to My Computer?

Your important files are encrypted.

Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?

Sure. We guarantee that you can recover all your files safely and easily. (But you have not so enough time.)
You can try to decrypt some of your files **for free**. Try now by clicking <Decrypt>.
If you want to decrypt all your files, you need to **pay**.

You only have 3 days to submit the payment. After that the price will be doubled.
Also, if you don't pay in 7 days, you won't be able to recover your files forever.

How Do I Pay?

bitcoin ACCEPTED HERE

Send $300 worth of bitcoin to this address:

QR Code

15zGqZCTcys6eCjDkE3DypCjXi6QWRV6V1

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

NHS - 70,000 hospital devices were impacted

WannaCry ransomware attack - Wikipedia

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

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?

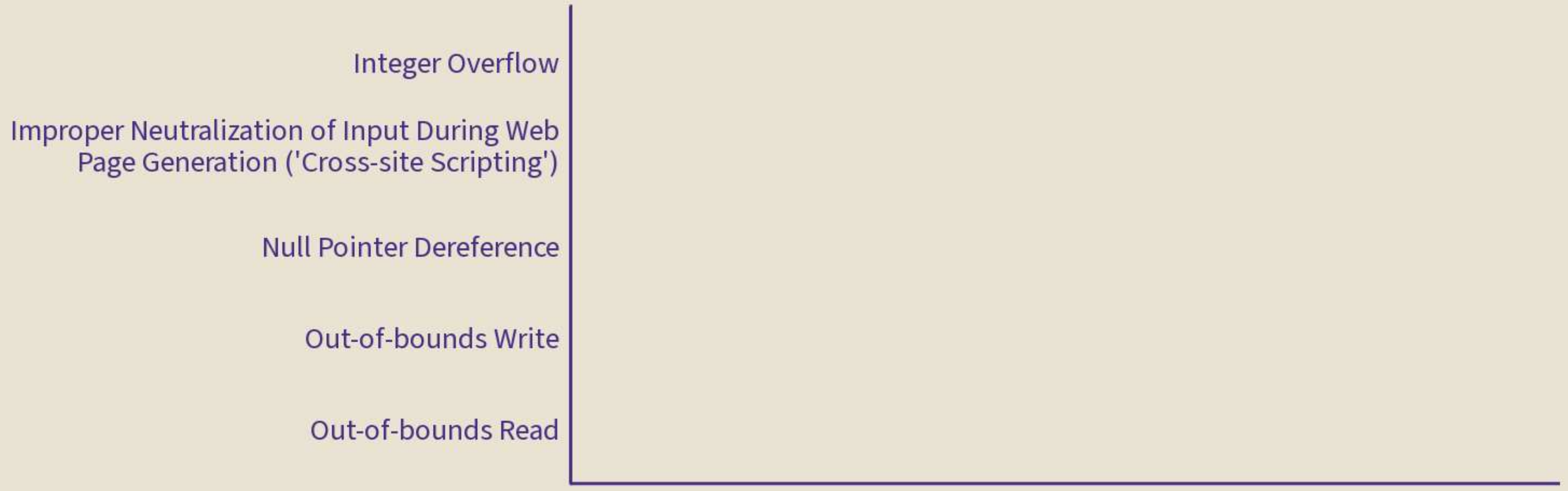
- Integer Overflow
- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Null Pointer Dereference
- Out-of-bounds Write
- Out-of-bounds Read

Total Results: 0

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?

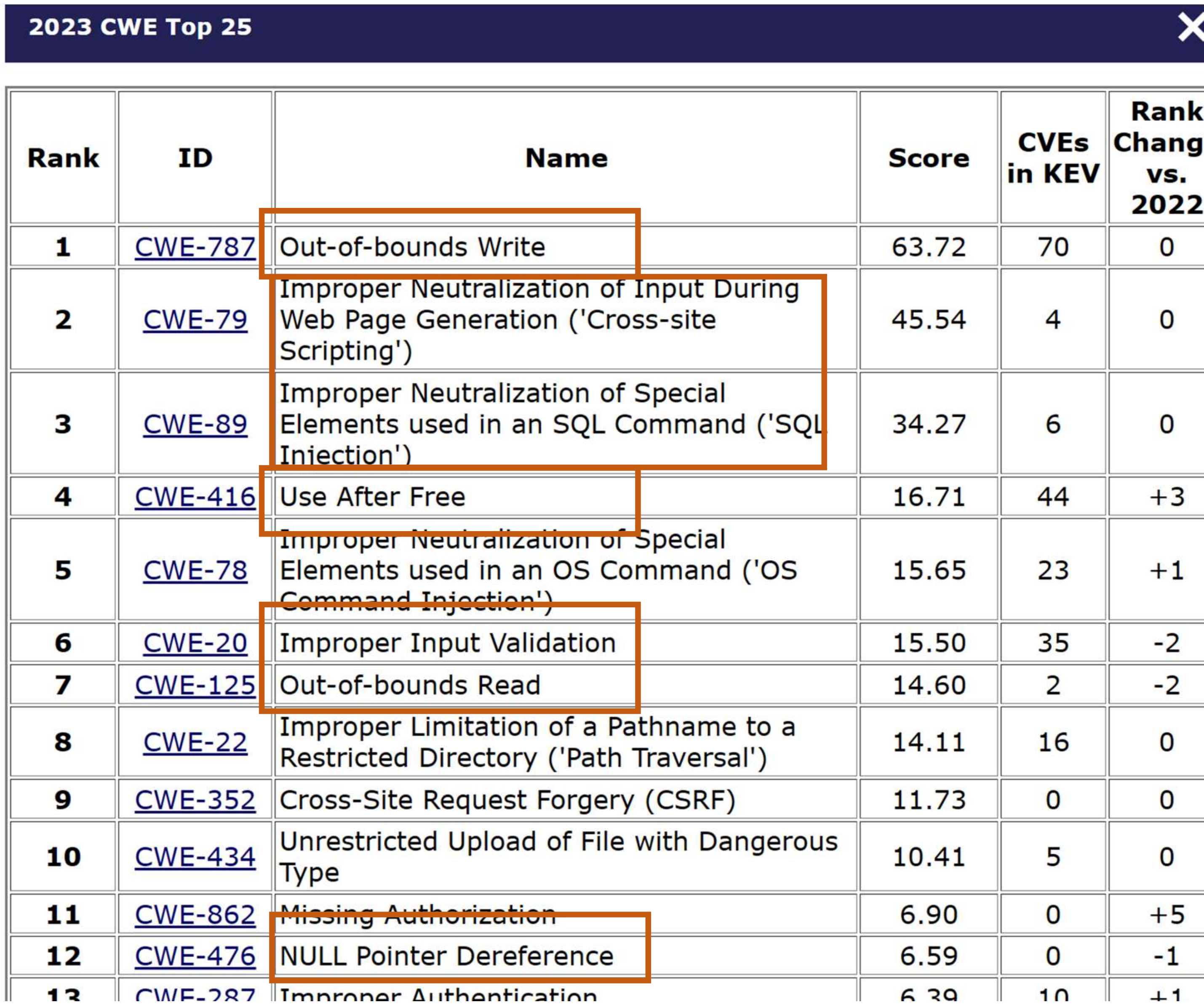


Total Results: 0

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)](#)



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 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

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

Proverb

- 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

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
- …

Today’s outline

Software testing

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

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

Starting at the top

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. <=, 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

```
<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

```
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`

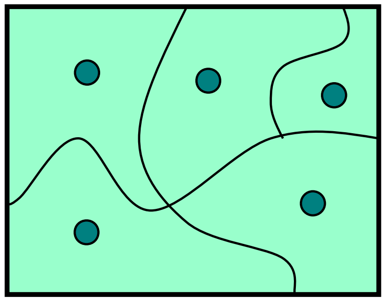
- How about…

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

Javadoc 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 <u>same</u> 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**

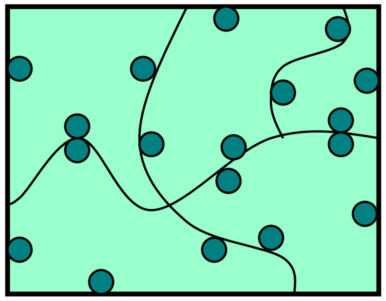
    Subdomain is revealing for an error, E, if
    1) Each element has same behavior
    2) If program has error E, it is revealed by test

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

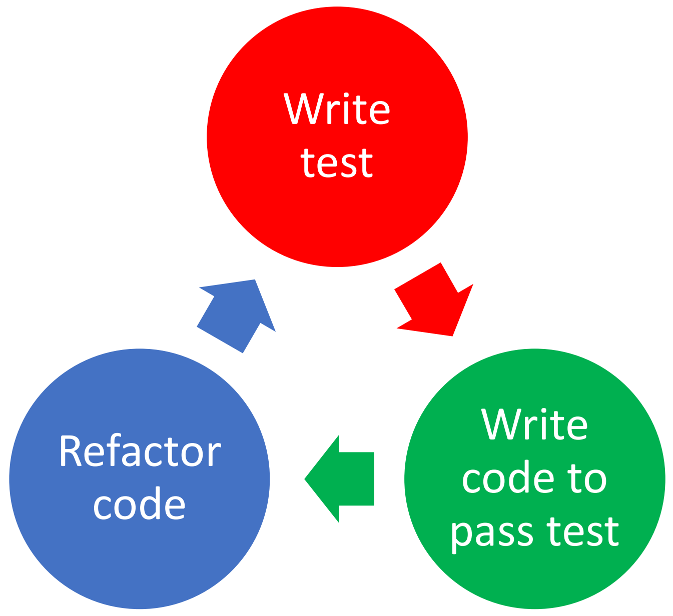
- Catches common boundary case bugs:
  - **Arithmetic**
    - Smallest/largest values
    - Zero
  - **Objects**
    - Null
    - Circular
    - Same object passed to multiple arguments (aliasing)



Black box: test driven development

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

```
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 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

```
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 = 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?

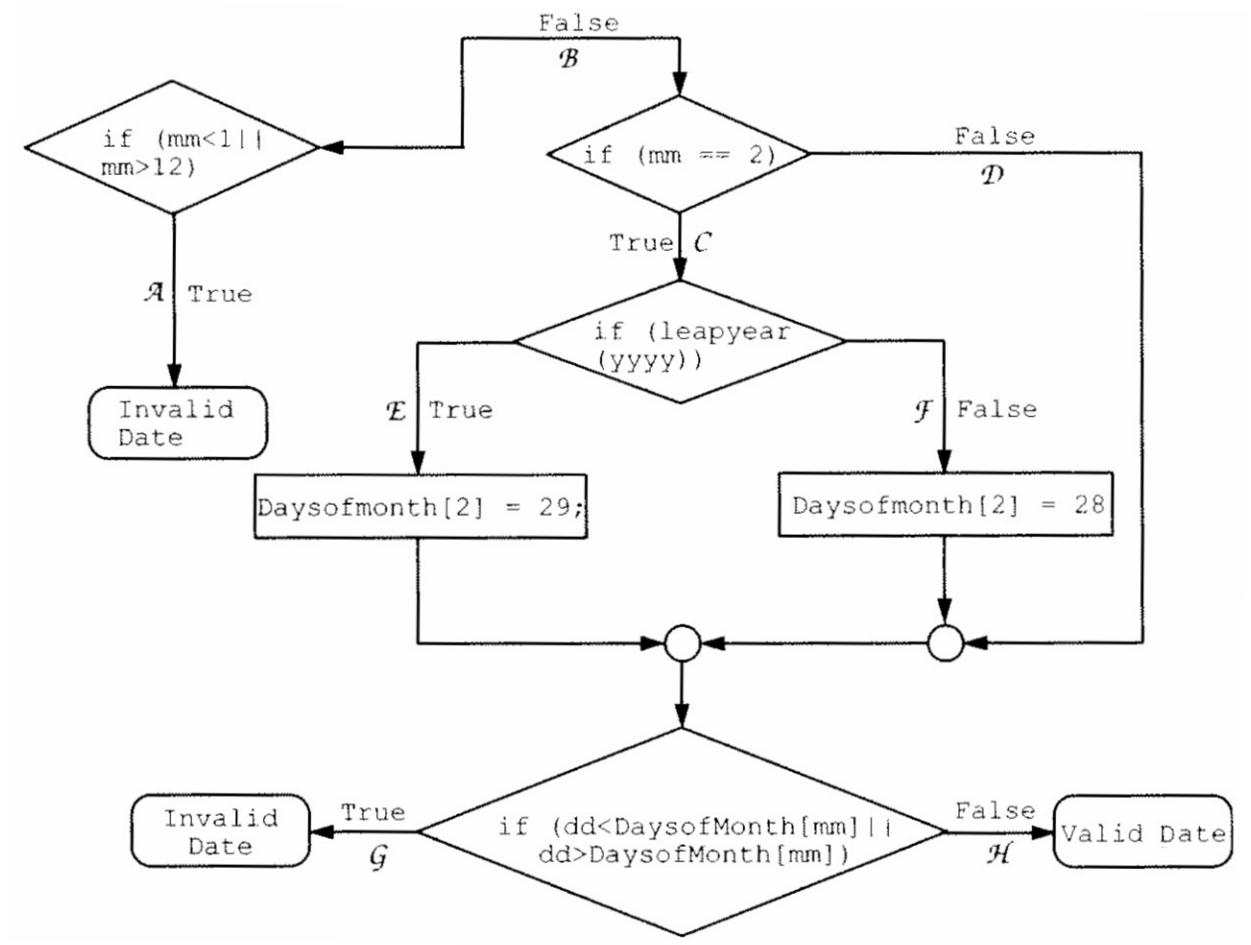
Double click on code coverage testing

code coverage testing: Examines what fraction of the code under test is reached by existing unit tests

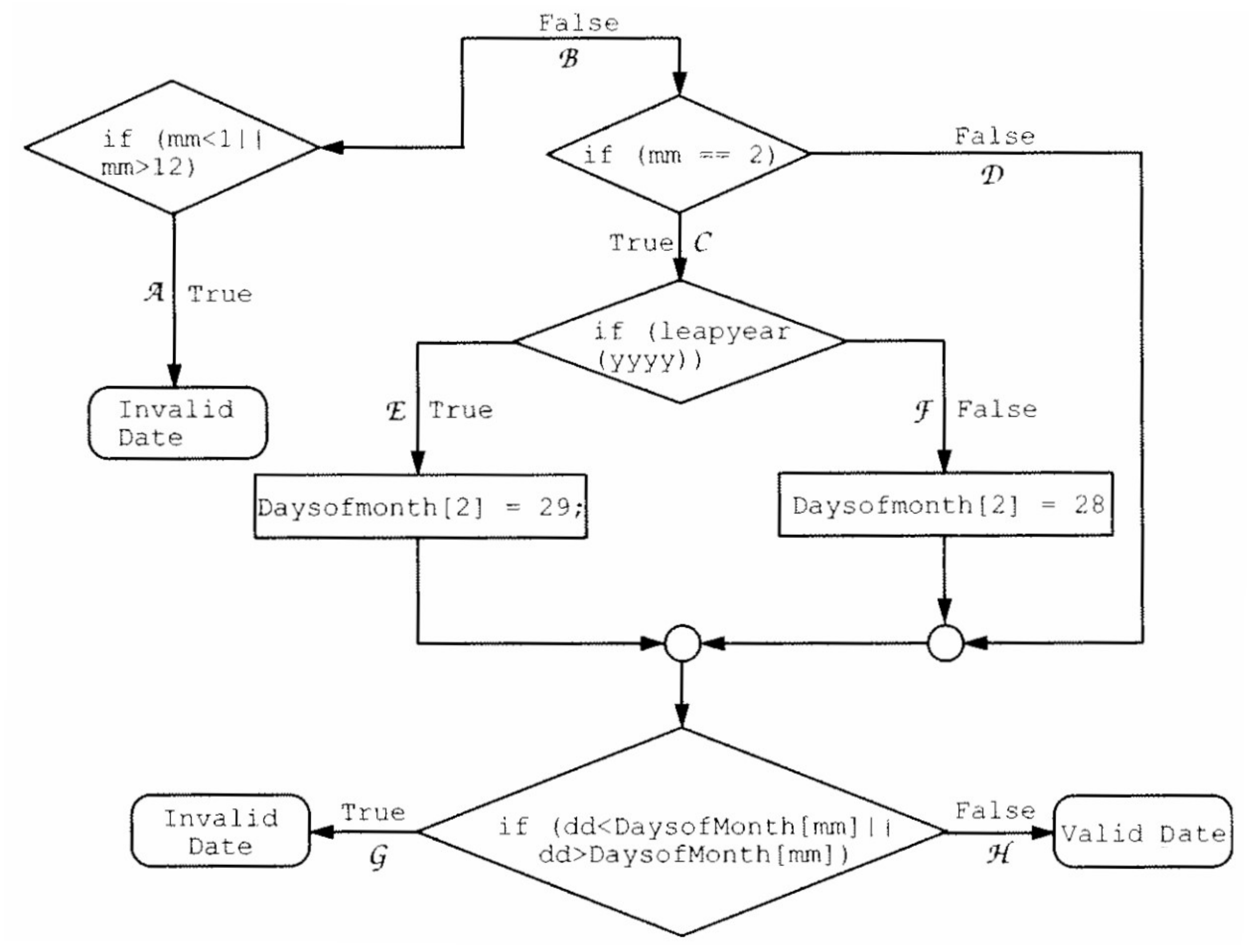
• statement coverage	- tries to reach every line (practical?)
• path coverage	- follow every distinct branch through code
• condition coverage	- every condition that leads to a branch
• function coverage	- treat every behavior / end goal separately

Dead code? A distraction? Or important?

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



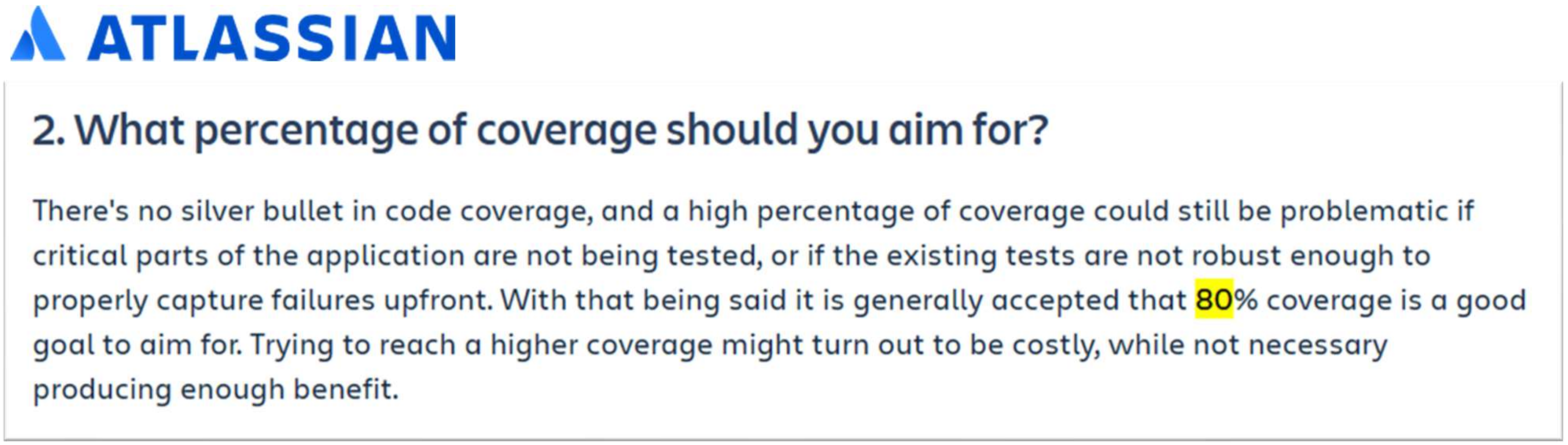
Consider tests to cover all paths for the **`Date`** class



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%



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 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