Software testing

CSE 403 Software Engineering
Winter 2025

Today’s outline

Software testing

- Motivating examples
- Categories of tests
- Unit testing

Teammate survey – see Ed Chat for your link - **due by Monday 11:59pm**

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

Boeing 787 Dreamliner (2015)

- The bug occurs when the software 32-bit counter overflows
- This happens if the generator control units are on for 248 days continuously
- Impact – the plane would lose all electrical power, even if in flight
- Bug was found (fortunately!) before it was triggered in service

US aviation authority: Boeing 787 bug could cause 'loss of control'

More trouble for Dreamliner as Federal Aviation Administration warns glitch in control unit causes generators to shut down if left powered on for 248 days

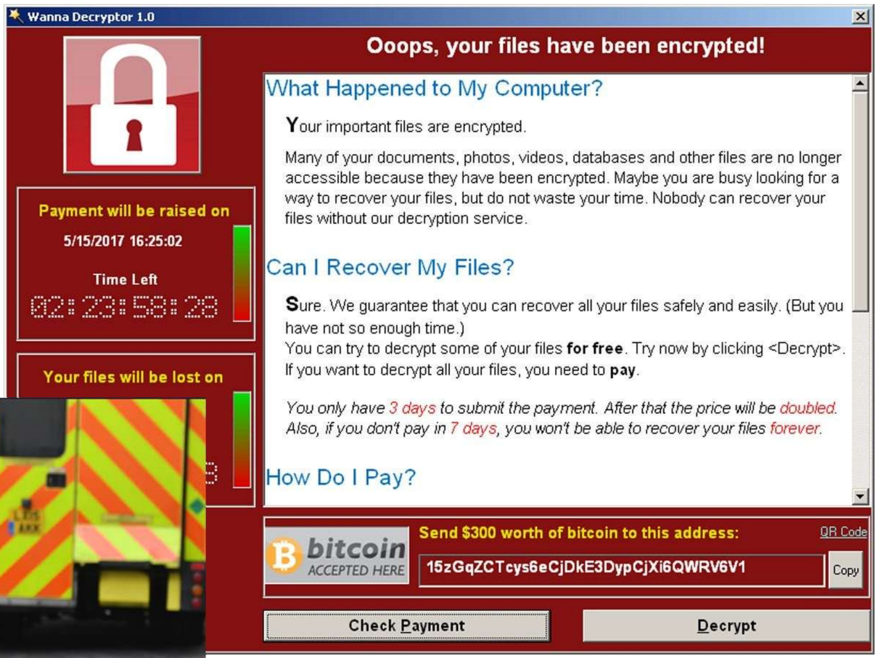
The Boeing 787 has four generator-control units that, if powered on at the same, could fail simultaneously and cause a complete electrical shutdown. Photograph: Elaine Thompson/AP

https://www.theguardian.com/

WannaCry Ransomware Attack (2017)

- Cryptoworm infected 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 $



Ooops, your files have been encrypted!

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?

Send $300 worth of bitcoin to this address:

15zGqZCTcys6eCjDkE3DypCjXi6QWRV6V1

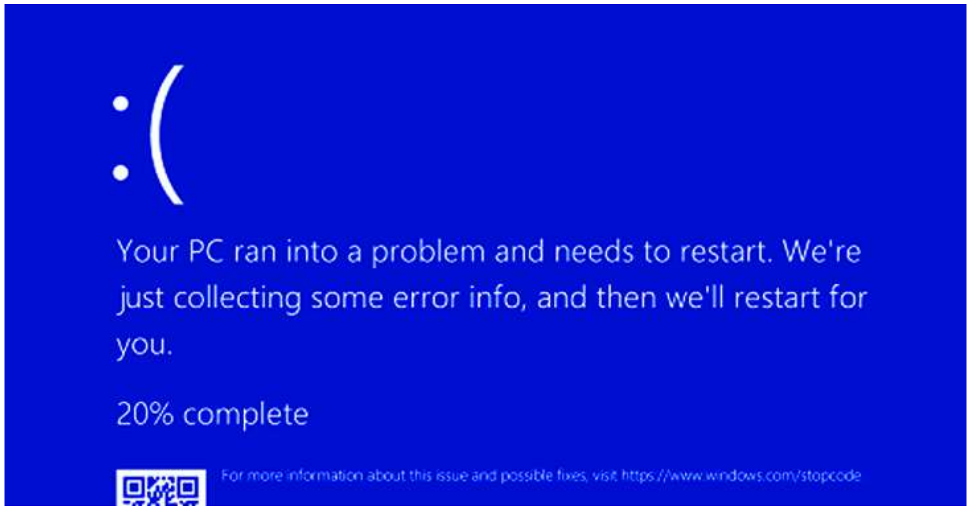
NHS - 70,000 hospital devices were impacted

WannaCry ransomware attack - Wikipedia

CrowdStrike (2024)

Cost of exploit: over $5 billion

- "Routine" update by CrowdStrike crashed Windows devices
- CrowdStrike is security endpoint protection software
- Update expected 20 input fields but provided 21, resulting in an out-of-bounds read
- Lack of testing across diverse environments before deployment
- Affected 8.5 million globally, across banking, healthcare, airlines and more



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

https://www.techtarget.com/

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

Which was the top most dangerous (severe and prevalent) software weakness in 2024?

- 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 + "'";
```

If the user enters “alverson”, the value of query is:

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

SQL injection

Assume this code:

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

If the user enters “alverson”, the value of `query` is:

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

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 “alverson”, the value of query is:

```
  "select from table where user=‘alverson'"
```

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

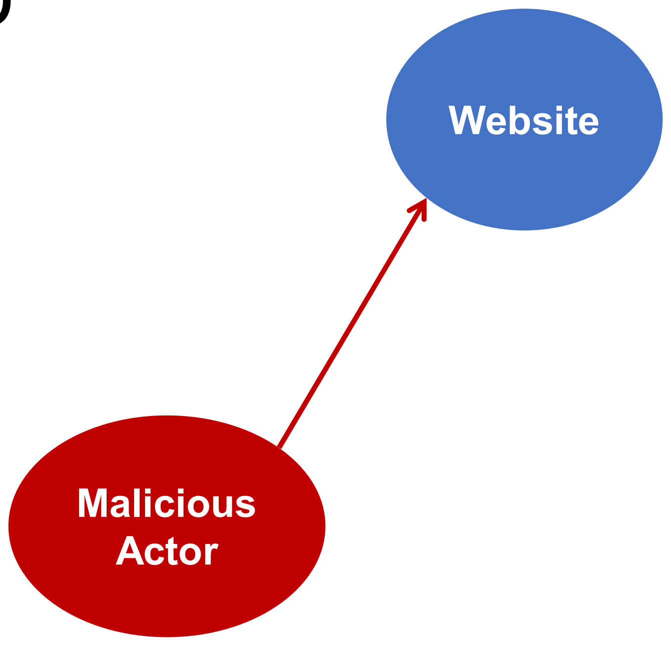
The value of query is:

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

Cross-site scripting

1. Malicious actor discovers a website with a vulnerability that enables a script to be injected

2. Malicious actor injects script that steals website user's info (like session cookies)



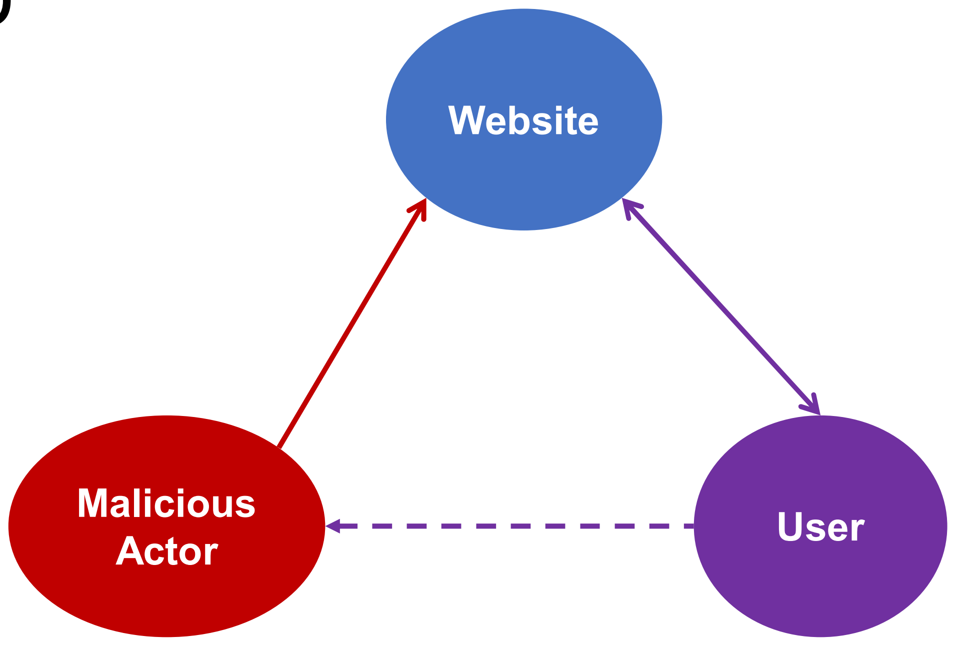
Cross-site scripting

1. Malicious actor discovers a website with a vulnerability that enables a script to be injected

2. Malicious actor injects script that steals website user's info (like session cookies)

3. Each time a user visits the website, the script is activated

4. User's session cookies are sent to malicious actor ☹, who can now access any user account info (like credit card info)



Title logo

What was the top (severe and prevalent) most dangerous software weakness reported in 2024?

0

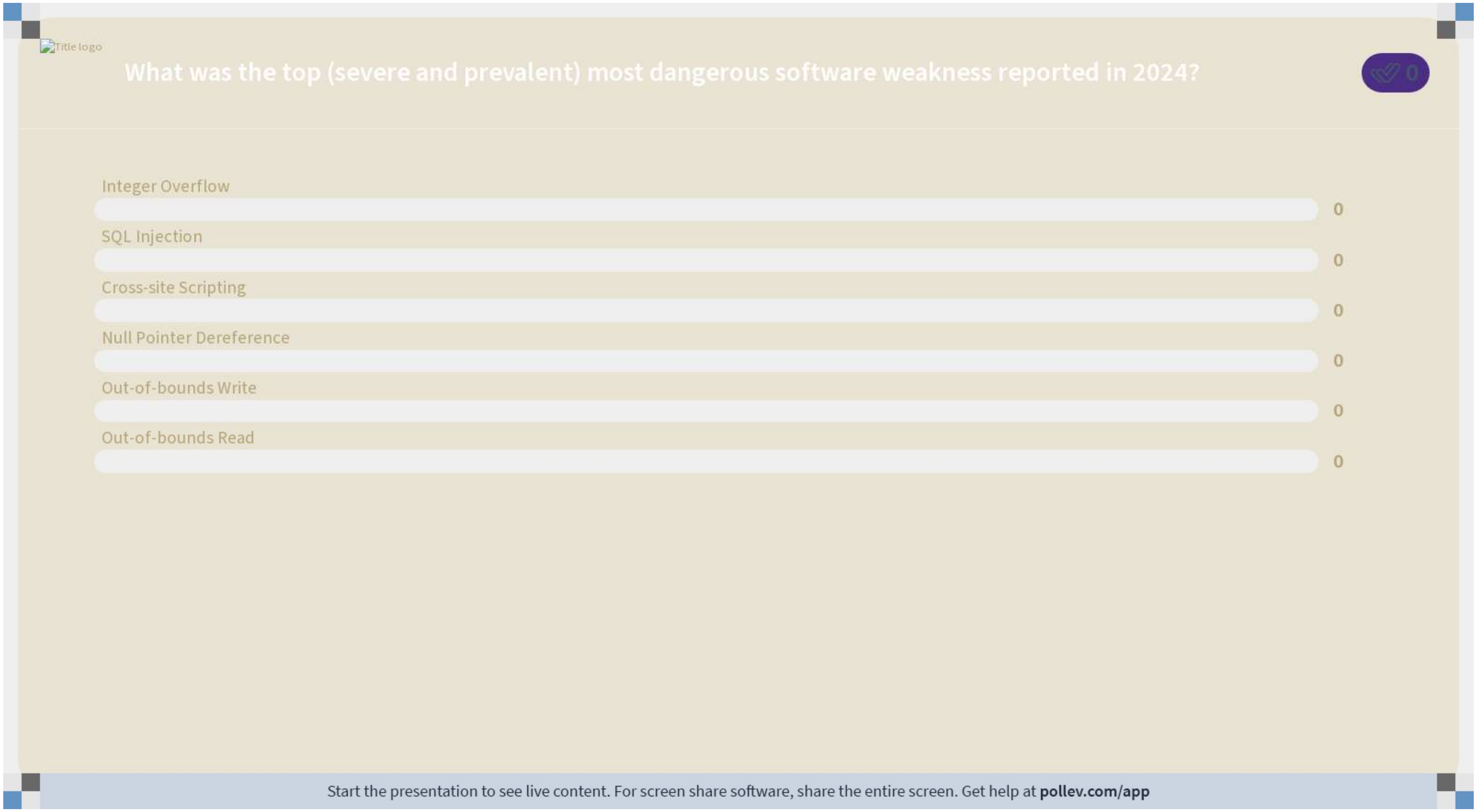
- Integer Overflow
- SQL Injection
- Cross-site Scripting
- Null Pointer Dereference
- Out-of-bounds Write
- Out-of-bounds Read

Title logo

What was the top (severe and prevalent) most dangerous software weakness reported in 2024?

0

Integer Overflow	0
SQL Injection	0
Cross-site Scripting	0
Null Pointer Dereference	0
Out-of-bounds Write	0
Out-of-bounds Read	0



Title logo

What was the top (severe and prevalent) most dangerous software weakness reported in 2024?

0

Integer Overflow	0
SQL Injection	0
Cross-site Scripting	0
Null Pointer Dereference	0
Out-of-bounds Write	0
Out-of-bounds Read	0

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

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.)

Testing vs. debugging

Testing: is there a bug?

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

Regression testing

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

Proverb stemming back to 1600's

- 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
- Protects against updates that reintroduce bug
  - It happened at least once, and it ~~might~~ will happen again

Today’s outline

Software testing

- Motivating examples
- Categories of tests
- **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

Unit testing

- A **unit** is the **smallest testable part** of the software system (e.g., a method in a Java class)
- **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

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?

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)

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
- Educated 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?
  - #2 and #6 2024 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

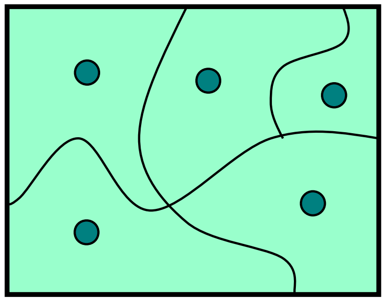
Boundary case #2

```
<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?

Theory explains why boundary testing works

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



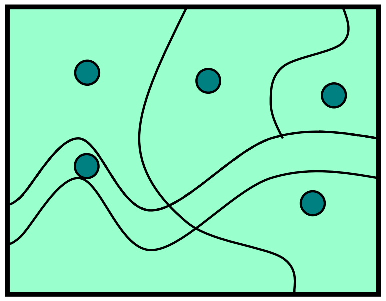
- "Same" behavior depends on specification
  - A program has the "same behavior" on two inputs if it:

    1) gives correct result on both, or

    2) gives incorrect result on both
  - "Same behavior" is unknowable
  - A subdomain is revealing for an error, E, if each test input fails (misbehaves)

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

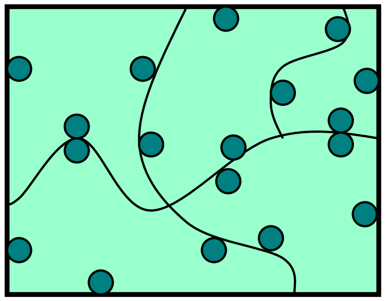
What if you mis-drew the boundaries?



Boundary case testing heuristic

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

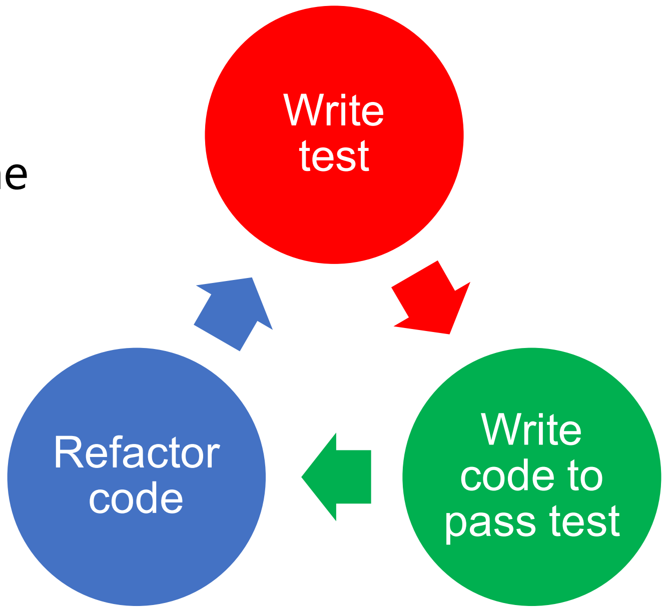
- Catches common boundary case bugs:
  - **Arithmetic**
    - Smallest/largest values
    - Zero
  - **Objects**
    - Null
    - Circular
    - Same object passed to multiple arguments (aliasing)



Black box: test driven development

Test driven development (TDD):

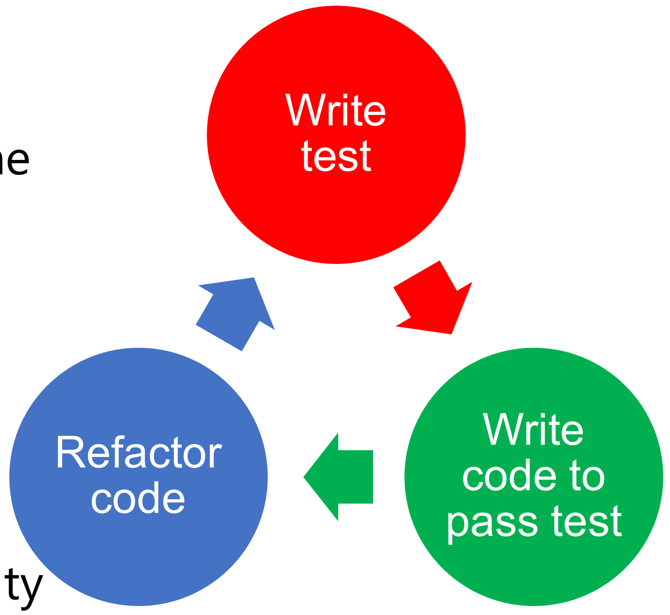
- What:
  - Test based on the spec and developed **before** the code is written
  - Will fail initially
  - Write just enough code to make it pass!
- Why?



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?
  - 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 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:
  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

Static code analysis is one type of white-box testing
(see "build" lecture)

Test suite code coverage is another
(coming Monday)

A motivating example for white box 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`

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?

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

- 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

Ultimate goal:
Maximize a measurement of the test suite

So, code coverage testing

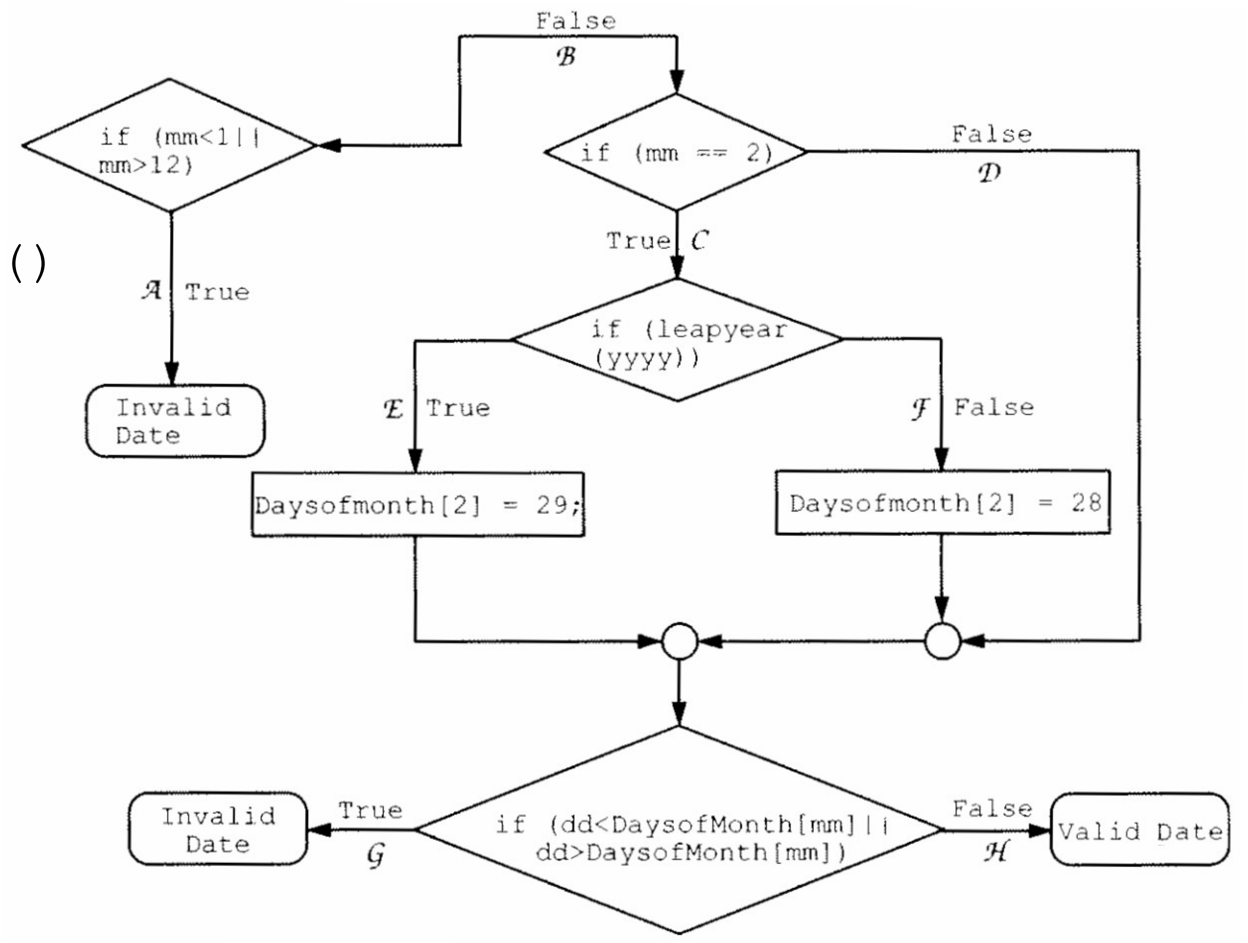
Code coverage testing: examines what fraction of the code under test is reached by existing unit tests

- Statement coverage - tries to reach every line (practical?)
- Branch 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?

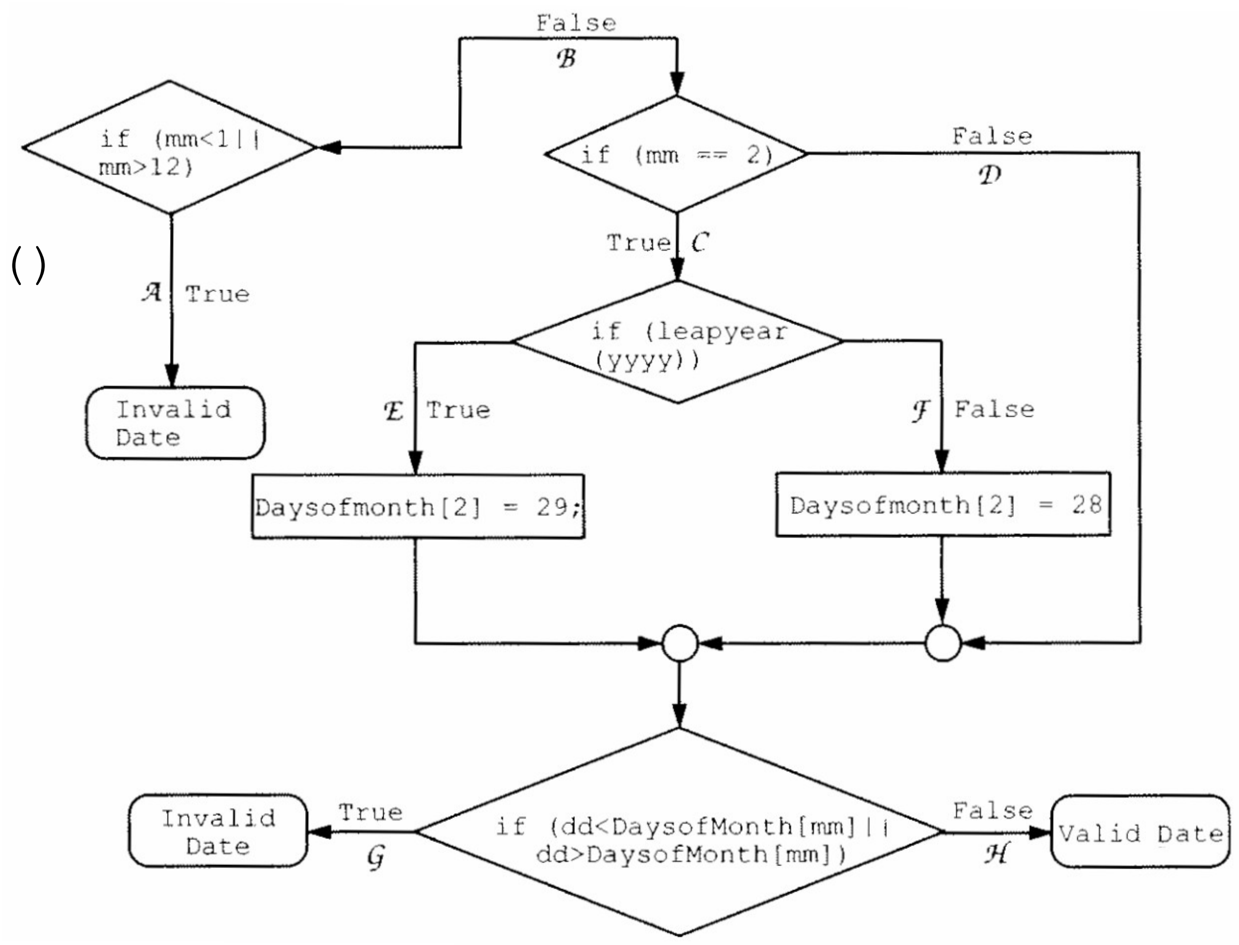
Consider tests to cover all paths for a `Date` class

`Date::`
`isValidDate()`



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

`Date::isValidDate()`



Try using a code coverage tool as part of your project testing

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