CSE 403
Lecture 9

Testing

Thanks to Michael Ernst and other past instructors of CSE 403 and CSE 331

http://www.cs.washington.edu/403/

Testing summary

- Testing matters
  - You need to convince others that module works
- Catch problems earlier
  - Bugs become obscure beyond the unit they occur in
- Don't confuse volume with quality of test data
  - Can lose relevant cases in mass of irrelevant ones
  - Look for revealing subdomains
- Choose test data to cover
  - Specification (black box testing)
  - Code (glass box testing)
- Testing can't generally prove absence of bugs
  - But can increase quality and confidence

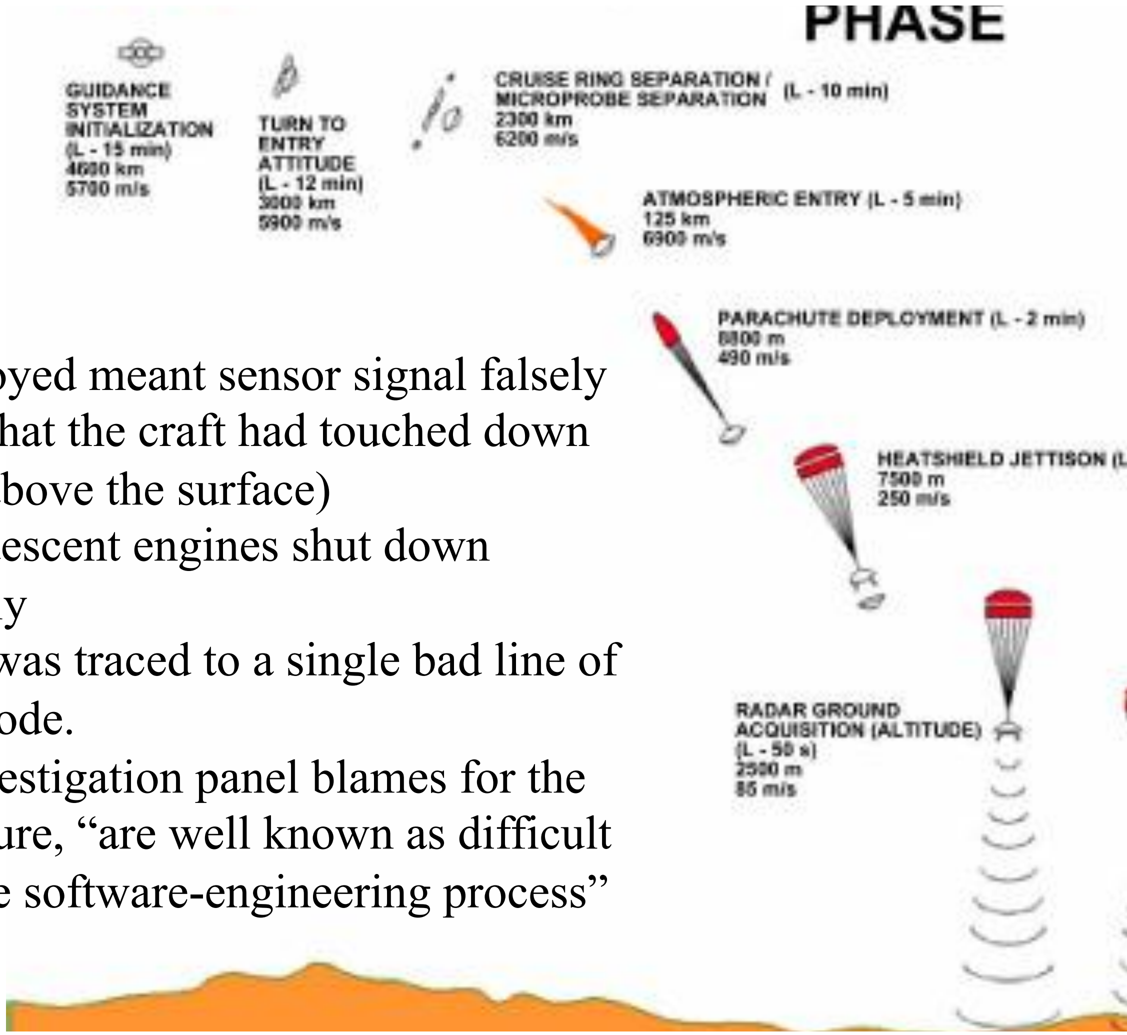
Ariane 5 rocket

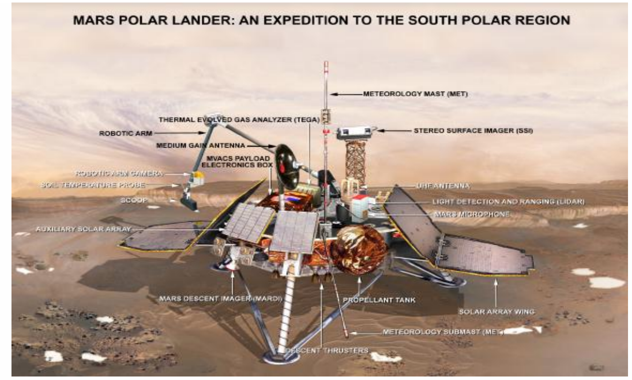
- The rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
  - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
  - The floating point number was larger than 32767 (max 16-bit signed integer)
  - Efficiency considerations had led to the disabling of the exception handler.
- Program crashed means the rocket crashed
- Total Cost: over $1 billion

Therac-25 radiation therapy machine

- Excessive radiation killed patients (1985-87)
- New design removed hardware interlocks that prevent the electron-beam from operating in its high-energy mode. Now all the safety checks are done in software.
- 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.
- This was missed during testing, since it took practice before operators were able to work quickly enough for the problem to occur.
- Panama, 2000: At least 8 dead
- Many more! (NYT 12/28/2010)

Mars Polar Lander





- Legs deployed meant sensor signal falsely indicated that the craft had touched down (130 feet above the surface)
- Then the descent engines shut down prematurely
- The error was traced to a single bad line of software code.
- NASA investigation panel blames for the lander failure, “are well known as difficult parts of the software-engineering process”

More examples

- Microsoft Zune's New Year Crash (2008)
  - iPhone alarm (2011)
- Air-Traffic Control System in LA Airport (2004)
- Northeast Blackout (2003)
- USS Yorktown Incapacitated (1997)
- Denver Airport Baggage-handling System (1994)
- Mariner I space probe (1962)
- AT&T Network Outage (1990)
- Intel Pentium floating point divide (1993)
- Prius brakes and engine stalling (2005)
- Soviet gas pipeline (1982)
  - Iran centrifuges (2009)

Testing is for *every* system

- Every little error adds up
- Inadequate infrastructure for software testing costs the U.S. $22-$60 billion per year
- Testing accounts for about half of software development costs.
- Program understanding and debugging account for up to 70% of time to ship a software product
- Improvements in software testing infrastructure might save one-third of the cost

- Source: NIST Planning Report 02-3, 2002

Building Quality Software

- What impacts software quality?
- External
  - Correctness *Does it do what it supposed to do?*
  - Reliability *Does it do it accurately all the time?*
  - Efficiency *Does it do with minimum use of resources?*
  - Integrity *Is it secure?*
- Internal
  - Portability *Can I use it under different conditions?*
  - Maintainability *Can I fix it?*
  - Flexibility *Can I change it or extend it or reuse it?*
- Quality Assurance
  - The process of uncovering problems and improving the quality of software.
  - Testing is a major part of QA.

What Is Testing For?

- Validation = reasoning + testing
  - Make sure module does what it is specified to do
  - Uncover problems, increase confidence

- Two rules:
- 1. Do it early and do it often
  - Catch bugs quickly, before they have a chance to hide
  - Automate the process if you can
- 2. Be systematic
  - If you thrash about randomly, the bugs will hide in the corner until you're gone

Phases of Testing

- Unit Testing
  - Does each module do what it supposed to do?
- Integration Testing
  - Do you get the expected results when the parts are put together?
- Validation Testing
  - Does the program satisfy the requirements?
- System Testing
  - Does it work within the overall system?

Unit Testing

- A test is at the level of a method/class/interface
  Check if the implementation matches the specification.

- Black box testing
  - Choose test data *without* looking at implementation

- Glass box (white box) testing
  - Choose test data *with* knowledge of implementation

How is testing done?

- Basic steps of a test
  1) Choose input data/configuration
  2) Define the expected outcome
  3) Run program/method against the input and record the results
  4) Examine results against the expected outcome

- Testing can't generally prove absence of bugs
  - But can increase quality and confidence

What's So Hard About Testing?

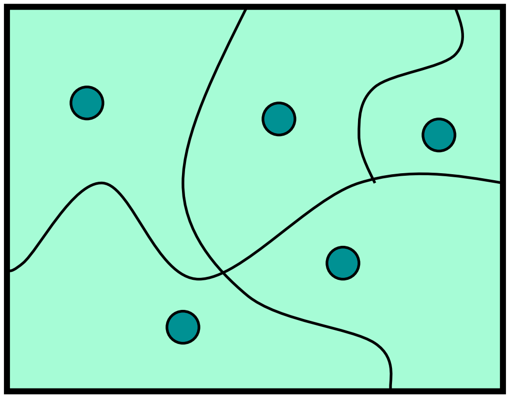
- "just try it and see if it works..."
- `// requires: 1 ≤ x,y,z ≤ 10000`
- `// effects:  computes some f(x,y,z)`
- `int proc1(int x, int y, int z)`
-
- Exhaustive testing would require 1 trillion runs!
  - Sounds totally impractical – and this is a trivially small problem
- Key problem: choosing test suite (set of partitions of inputs)
  - Small enough to finish quickly
  - Large enough to validate the program

sqrt example

- ```
  // throws: IllegalArgumentException if x<0
  // returns: approximation to square root of x
  public double sqrt(double x)
  ```
- What are some values or ranges of x that might be worth probing?
  - $x < 0$ (exception thrown)
  - $x \geq 0$ (returns normally)
  - around $x == 0$ (boundary condition)
  - perfect squares (sqrt(x) an integer), non-perfect squares
  - $x <$ sqrt(x) and $x >$ sqrt(x) – that's $x < 1$ and $x > 1$ (and $x == 1$)
  - *Specific tests: say x = -1, 0, 0.5, 1, 4*
  ```

Approach: Partition the Input Space

- Ideal test suite:
  - Identify sets with same behavior
  - Try one input from each set



- Two problems
  - 1. Notion of the same behavior is subtle
    - Naive approach: execution equivalence
    - Better approach: revealing subdomains
  - 2. Discovering the sets requires perfect knowledge
    - Use heuristics to approximate cheaply

Naive Approach: Execution Equivalence

```
// returns:  x < 0      => returns -x
//           otherwise => returns x
int abs(int x) {

   if (x < 0) return -x;
   else       return x;
 }
```

All x < 0 are execution equivalent:

program takes same sequence of steps for any x < 0

All x ≥ 0 are execution equivalent

Suggests that {-3, 3}, for example, is a good test suite

Why Execution Equivalence Doesn't Work

Consider the following buggy code:

```
// returns:  x < 0      => returns -x
//           otherwise => returns x
int abs(int x) {

    if (x < -2) return -x;
    else        return x;
 }
```

Two executions:

$x < -2$ $\quad$ $x \geq -2$

Three behaviors:

$x < -2$ (OK) $\quad$ $x == -2$ or -1 (bad) $\quad$ $x \geq 0$ (OK)

{-3, 3} does not reveal the error!

Heuristic: Revealing Subdomains

- A subdomain is a subset of possible inputs

- A subdomain is *revealing* for error E if either:
  - *Every* input in that subdomain triggers error E, or
  - *No* input in that subdomain triggers error E

- Need test only one input from a given subdomain
  - If subdomains cover the entire input space, then we are <u>guaranteed</u> to detect the error if it is present

- The trick is to guess these revealing subdomains

Heuristics for Designing Test Suites

A good heuristic gives:

- few subdomains
- ∀ errors E in some class of errors,
  - high probability that some subdomain is revealing for E

- Different heuristics target different classes of errors
  - In practice, combine multiple heuristics

Black Box Testing

- Heuristic: Explore alternate paths through specification
  - Procedure is a black box: interface visible, internals hidden
- Example
  -
```
int max(int a, int b)
  // effects:  a > b => returns a
  //           a < b => returns b
  //           a == b => returns a
```
  - 3 paths, so 3 test cases:

    (4, 3) => 4 *(i.e. any input in the subdomain a > b)*

    (3, 4) => 4 *(i.e. any input in the subdomain a < b)*

    (3, 3) => 3 *(i.e. any input in the subdomain a == b)*

Black Box Testing: Advantages

- Process is not influenced by component being tested
  - Assumptions embodied in code not propagated to test data.
- Robust with respect to changes in implementation
  - Test data need not be changed when code is changed
- Allows for independent testers
  - Testers need not be familiar with code

More Complex Example

- 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 is not in a
    ```

- Two obvious tests:

  ([4, 5, 6], 5) => 1
  ([4, 5, 6], 7) => throw Missing

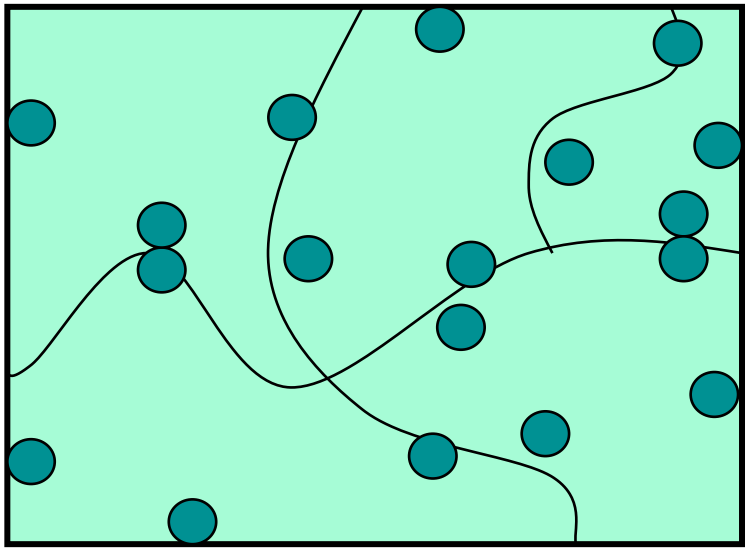
- Have I captured all the paths?

  ([4, 5, 5], 5) => 1

- Must hunt for multiple cases in effects or requires
    ```

Heuristic: Boundary Testing

- Create tests at the edges of subdomains



- Why do this?
  - off-by-one bugs
  - forgot to handle empty container
  - overflow errors in arithmetic
  - aliasing

- Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors

- Also, you might have misdrawn the boundaries

Boundary Testing

- To define the boundary, need a distance metric
  - Define adjacent points
- One approach:
  - Identify basic operations on input points
  - Two points are adjacent if one basic operation apart
- Point is on a boundary if either:
  - There exists an adjacent point in a different subdomain
  - Some basic operation cannot be applied to the point
- Example: list of integers
  - Basic operations: create, append, remove
  - Adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>
  - Boundary point: [] (can't apply remove integer)

Other Boundary Cases

- Arithmetic
  - Smallest/largest values
  - Zero
- Objects
  - Null
  - Circular list
  - Same object passed to multiple arguments (aliasing)

Boundary Cases: Arithmetic Overflow

- ***public int abs(int x)***
- ***// returns: |x|***
- Tests for abs
  - what are some values or ranges of x that might be worth probing?
    - $x < 0$ (flips sign) or $x \geq 0$ (returns unchanged)
    - around $x == 0$ (boundary condition)
    - *Specific tests: say x == -1, 0, 1*
- *How about…*

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

- From Javadoc for Math.abs:
  - Note that 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

Boundary Cases: Duplicates & 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

    while (src.size()>0) {
        E elt = src.remove(src.size()-1);
        dest.add(elt)
    }
}
```

- What happens if src and dest refer to the same thing?
  - This is *aliasing*
  - It's easy to forget!
  - Watch out for shared references in inputs

Clear (glass, white)-box testing

- Goals:
  - Ensure test suite covers (executes) all of the program
  - Measure quality of test suite with % coverage
- Assumption:
  - high coverage → few mistakes in the program
  - (Assuming no errors in test suite oracle (expected output).)
- Focus: features not described by specification
  - Control-flow details
  - Performance optimizations
  - Alternate algorithms for different cases

Glass-box Motivation

- There are some subdomains that black-box testing won't give:

```
•     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];
•          }
•     }
```

- Important transition around x == CACHE_SIZE

Glass Box Testing: Advantages

- Finds an important class of boundaries
  - Yields useful test cases
- Consider **`CACHE_SIZE`** in **`isPrime`** example
  - 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

- **Disadvantages?**
  - **Tests may have same bugs as implementation**

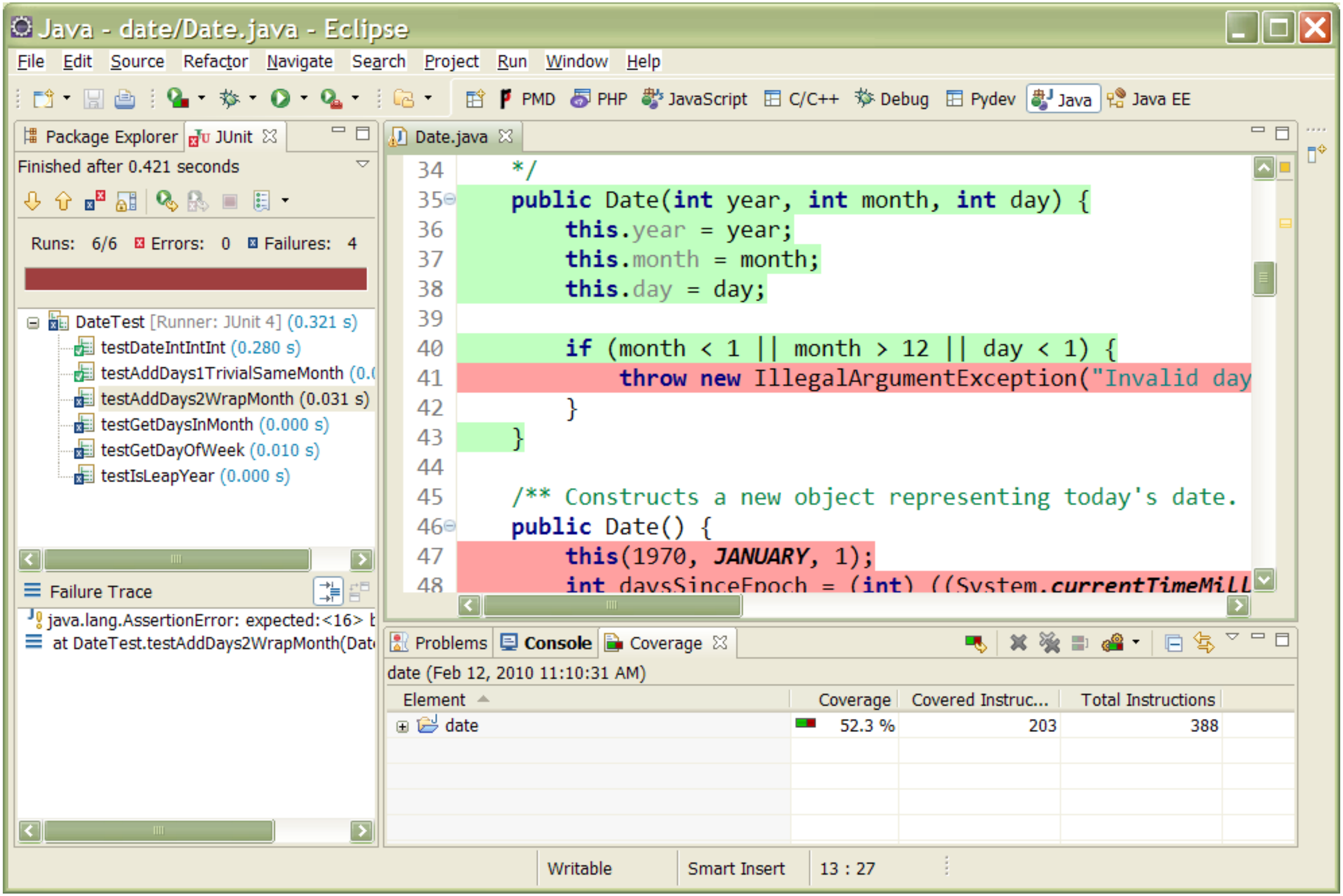
What is full coverage?

- ```
  static int min (int a, int b) {
      int r = a;
      if (a <= b) {
         r = a;
      }
      return r;
  }
  ```

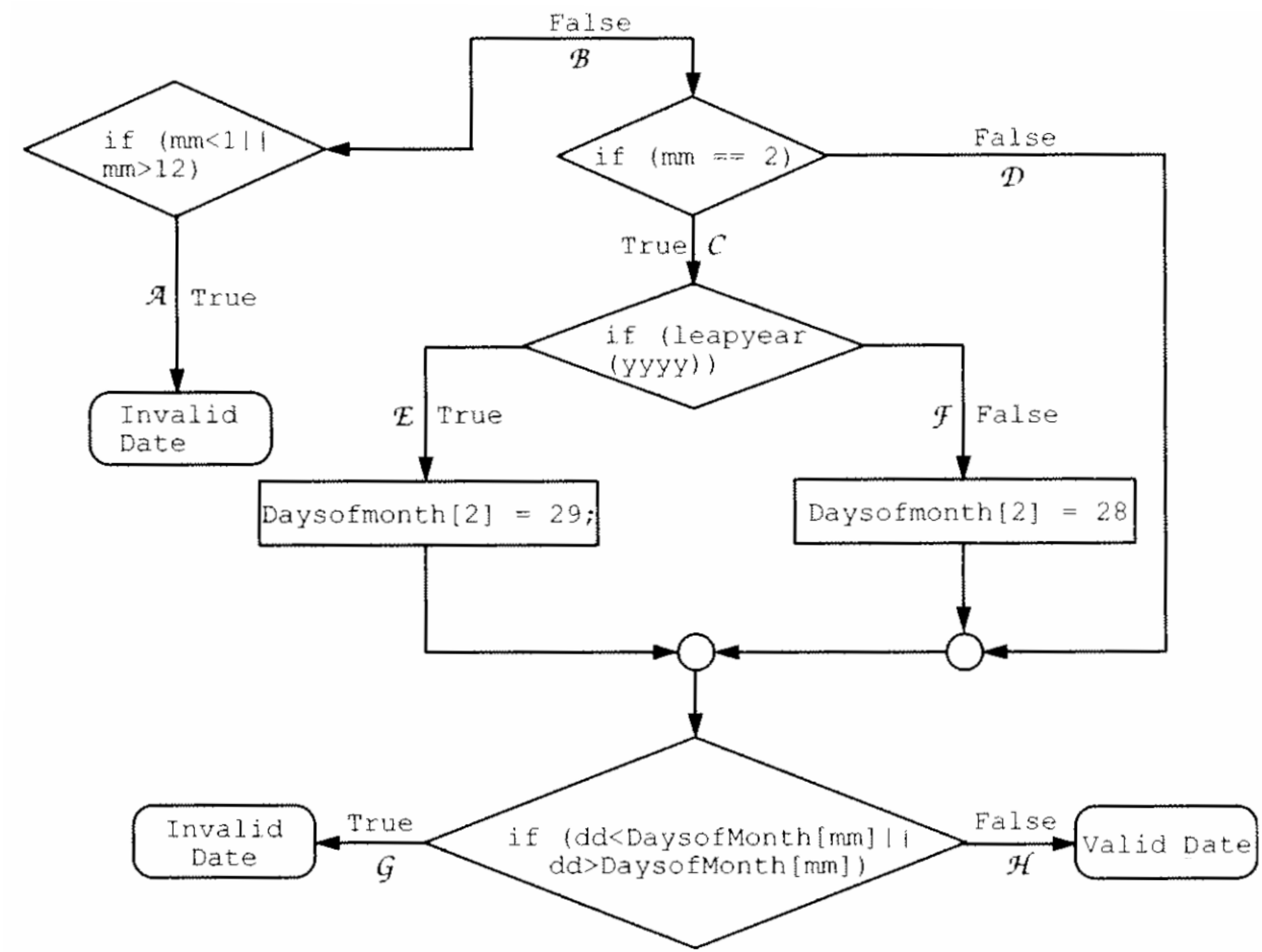
- Consider any test with a $\leq$ b (e.g., `min(1,2)`)
  - It executes every instruction
  - It misses the bug

- *Statement* coverage is not enough
  ```

Code coverage example



Path coverage example



Varieties of coverage

- Covering all of the program
  - Statement coverage
  - Branch coverage
  - Decision coverage
  - Loop coverage
  - Condition/Decision coverage
  - Path coverage

increasing number of test cases (more or less)

- Limitations of coverage:
  1. 100% coverage is not always a reasonable target
     100% may be unattainable (dead code)
     High cost to approach the limit
  2. Coverage is just a heuristic
     We really want the revealing subdomains

Regression Testing

- Whenever you find a bug
  - Store the input that elicited that bug, plus the correct output
  - Add these to the test suite
  - Verify that the test suite fails
  - Fix the bug
  - Verify the fix
- Why is this a good idea?
- Ensures that your fix solves the problem
  - Don’t add a test that succeeded to begin with!
- Helps to populate test suite with good tests
- Protects against reversions that reintroduce bug
  - It happened at least once, and it might happen again

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 the process if you can
  - Regression testing will save time.

- Second rule of testing: ***Be systematic***
  - If you randomly thrash, bugs will hide in the corner 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
    - Incorrect, incomplete, ambiguous, and missing corner cases
  - When you find a bug → write a test for it first and then fix it

Testing summary

- Testing matters
  - You need to convince others that module works
- Catch problems earlier
  - Bugs become obscure beyond the unit they occur in
- Don't confuse volume with quality of test data
  - Can lose relevant cases in mass of irrelevant ones
  - Look for revealing subdomains
- Choose test data to cover
  - Specification (black box testing)
  - Code (glass box testing)
- Testing can't generally prove absence of bugs
  - But can increase quality and confidence