



CS1632, LECTURE 2: TESTING THEORY AND TERMINOLOGY



KEY () CONCEPT TO THE COURSE

EXPECTED BEHAVIOR VS OBSERVED BEHAVIOR



EXPECTED BEHAVIOR VS OBSERVED BEHAVIOR

You need to know what “should” happen under some circumstances, then check to see if that behavior actually occurred.

For example, assume I have a function `foo`, which accepts an integer, `a`, and returns a float. What should happen if I send in the value `a = 42`?

This is a simple idea, but it’s the “Fundamental Theorem of Testing” (although note that we may violate it later...)

EXAMPLE

Assume `foo` is supposed to return the square root of the passed in value `a`.

When I send in the value `a = 42`, then I expect to be returned the value `6.48074069841`.

When I send in the value `a = 9`, then I expect to be returned the value `3`.

When I send in the value `a = -1`, then I expect....

THE IMPOSSIBILITY OF EXHAUSTIVE TESTING

- Let's say we want to ensure that our square root method will never fail, no matter what we send in. Assume we are using a standard Java int (signed 32-bit integer)
- How many values do we have to test?

4,294,967,296

WHAT ABOUT A MEDIUM-SIZED, 1 000-METHOD JAVA PROGRAM?

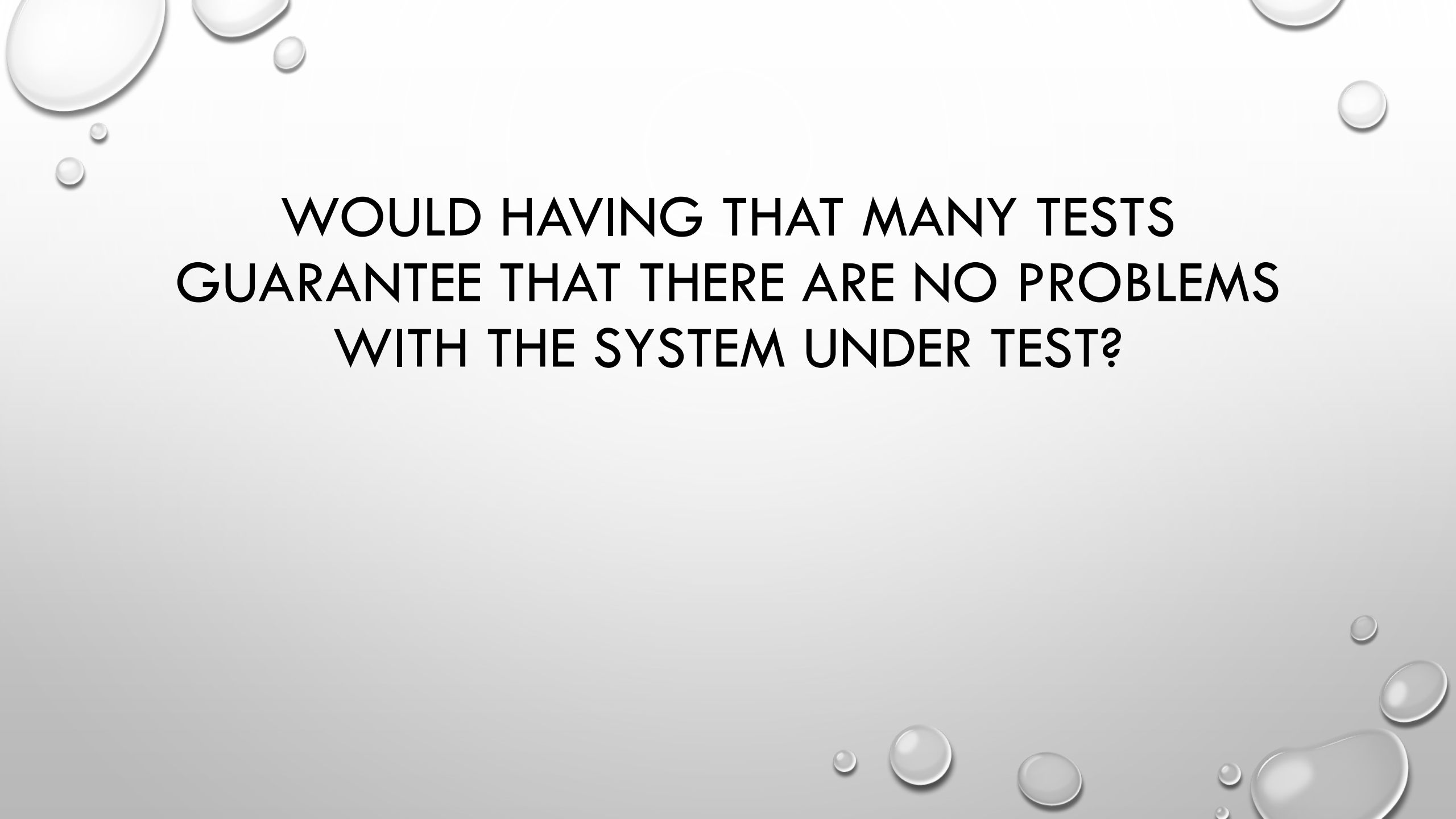
- Assume that each method accepts one 32-bit int argument and returns one primitive value.
- If we have references to objects, or multiple arguments, etc., then the program has even more possibilities to test.
- Remember that methods in a java-like language could theoretically influence other methods (e.g., setting global variables, calling other methods, mutating objects, etc.)

4,294,967,296 ^ 1000

THAT'S EQUAL TO...

4307035597053022521561246898546680858024365954586897143119190611638212638342763480050687188258215390720171988441441886648791575297627253893662165574468243228700314685915827483050821076923315530782979561709655922726932
309421537953749064364429040393033772835819089833294065713178992646977228120610193808992368287721753962035495230692778928668824704488245700338445100446606998126961778119374887575650255655908
396845270435452389665755740352688795123696892838818040660850711941690965464303296715997735315114393298314639765906325716916902260584852691127482320528326491613742609540149922601580190974842315012367812348080603774582
48309478650101041149164099071980888428421368546177179447495527132137857711089612888118312376663794867678135461880066435240486394452956529788529633837378507602411635143827166518753745556991400676914560717482710875903621
7371186004470408462533609154350069171432391905236698147627643706567883856399887838188577300034105386260275801457249934804175739466067697350928216693063295588063141404637171375667379841266106690291293383395081456948512
017486798190459730596356271880497115584194416928780687234039729937760519062604992990827029933269512641490387941707236776708508619444204382536670006178148067923537408135936520818300444573906970007383971690175067381164
9685649896391207546309917618762444947550045733654312891655363197987532936361064687315411060583286838622753457575546324055632471235420306870593261281242373060461770747490826702439896279150918198244130312153582
1211536536703122879836190043609803359393931710850772549460576386728537579974232974920375624778180923231343430936921083911139534659465800121513010419901123405611013130169487104000337304071364153665375401392391571914559
51689480400999761684077927466992190189673014569128322473733053895949364134343808332499059987264782814655949125056590781796556422804746351626979859227647826549850228866744876451447751151748843299619897972810381849869432
29215610249668503146259070821149147147929671318636093993055261646254668852152256223974186547033820664324635426978298582188285589712230975600324625417593418038572317911977583717648988763575138455279752979839590902979465
13522186617719293774695529057046639305389229919879152423701995088181482532272670194314379104038045758340779939556947096812250096343981167557247595611793311581144998431728888764229631959149698707010011920915003
4856224927052532493224975983599685492367548669022136744554833063261554517353101221828250025061008914765084554497383187477079793586203242147317171148464196840793949708328383169015326049167770286567980283888624968356429
59705090730476854531567246951502200043433665070248298446185971155224458673695436284935491043078408600113757452408977665439302231558859343031612693920949653983249653540586738417754022522371175536071967190921501772406926
209800239929304451483064141164445566433084336365728781197999649274374069115041210530177145822340279056844529954715900646233754267861944256911059891234426533672099268369543158016134038936671758535029480280838733922701
53115211059806246650339810254098811434005506705660238583972530934203626456674470502804264344071460846384555766231376145368465076041733818936624509202579048128118406191307095504038850600579284504917698983684288769207
4864943269762020830590249133982593101379509501033117211837245861653473607412360243912202942241955088005738531528651080242146153895816621263769903867197484181186417185583295969826273157124473654972920414561433567394573
08114036780697679783341964181398779863871025415770527055336426195241935221037058819757293995048374550819894222334319683340221385980807072484235545990481780316101856266200860564034686501297109898065953529438125588949324
2258472263489093108124335616173618991523396812990743904901569122226022744322461111500807098151836802194371233157448935859841962696238686678704566197978152142493878887304368391542652290664310132765507145505463076295716
50128910962661940478119939159806757686024198773028092601855291898050622497661356201163684378850172037399421215068245561343252835659201259131784550393137415359042915578453153620234916055276644643875

TEST CASES!

The image features a light gray background with a subtle gradient. In the top-left and bottom-right corners, there are several realistic-looking water droplets of various sizes, some overlapping. The text is centered in the upper half of the image.


**WOULD HAVING THAT MANY TESTS
GUARANTEE THAT THERE ARE NO PROBLEMS
WITH THE SYSTEM UNDER TEST?**

LOL NOPE

- Data races?
- Compiler issues?
- Non-functional issues (performance, usability, etc.)?
- Floating-point issues?
- Integration issues?
- Systems-level issues?
- Ambiguous or misunderstood requirements?



TESTING = ART + SCIENCE

- There are techniques for testing which can reduce the number of tests necessary for sufficient test coverage.
 - We will need to define what we mean by “sufficient test coverage”.
 - We will also require domain knowledge.
- 

EQUIVALENCE CLASS PARTITIONING

- We can partition the testing parameters into “equivalence classes”
 - Equivalence class = a natural grouping of values with similar behavior
- For example, in our square root method:
 - Negative numbers (input) -> Imaginary numbers (output)
 - 0 -> 0
 - Positive numbers -> Positive numbers

EQUIVALENCE CLASSES ARE STRICTLY PARTITIONED

- For any given input value, it must belong to one and **ONLY** one equivalence class (strictly partitioned)
 - If there are values that seem like they belong in multiple equivalence classes, you either need:
 - Multiple partitionings
 - Another equivalence class

EXAMPLE

- Assume you have a program which will return the square root of an int, and if the number is whole (e.g., 1 or 2, but not 1.342), it should print it out in **red**, otherwise it will print it out in black.
- You can have two partitionings:
 - (the positive/0/negative partitioning on the previous slide)
 - Another partitioning:
 - Number is whole -> output printed in **red**
 - Number is not whole -> output printed in black
- Therefore, for every value, there are multiple partitionings to check

THEY DO NOT HAVE TO BE NUMERIC

- On Twitter, if you follow somebody, you see all of their tweets, unless they are writing directly to somebody you do not follow.
- Equivalence classes:
 - You do not follow person A -> DO NOT see the tweet
 - You do follow person A, they are not writing directly to somebody -> see the tweet
 - You do follow person A, they are writing directly to person B, whom you also follow -> see the tweet
 - You do follow person A, they are writing directly to person B, whom you do not follow -> DO NOT see tweet

THEY DO NOT HAVE TO BE NUMERIC

- Suppose Twitter only allows alphanumeric `[A-Za-z0-9]` characters, and tweets must contain at least one character. Tweets that contain any invalid characters are not posted.
- Equivalence classes (NV = number of valid characters, NI = number of invalid characters):
 - $(NV \geq 1, NI == 0)$ -> Post the tweet
 - $(NV == 0, NI == 0)$ -> DO NOT post the tweet
 - $(NI \geq 1)$ -> DO NOT post the tweet (note NV is irrelevant here)

TEST EACH EQUIVALENCE CLASS

- Pick at least one value from each equivalence class
- This will ensure you capture behavior from each “class” of possible behavior
- Will find a good percentage of defects without exhaustive testing!
- We reduced the problem something a human can do! Woo-hoo!
- How to pick the input? Well, that is part of the art.
 - However, there are some good guidelines!



INTERIOR AND BOUNDARY VALUES

- Theory: Problems are more prevalent on the boundaries of equivalence classes than in the middle.
- 

WHY?

```
public boolean canBePresidentOfUnitedStates(int age) {  
    return age > 35;  
}
```

EQUIVALENCE CLASS PARTITIONING

CANNOT_BE_PRESIDENT =

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34]

CAN_BE_PRESIDENT =

[35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64..
..INFINITY]

WHERE ARE PROBLEMS LIKELY?

CANNOT_BE_PRESIDENT =

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,
34]

CAN_BE_PRESIDENT =

[35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64..
..INFINITY]

TRY TO ENSURE THAT YOU TEST BOUNDARY AND INTERIOR VALUES

CANNOT_BE_PRESIDENT =

[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34]

CAN_BE_PRESIDENT =

[35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64..
..INFINITY]

- Are we missing anything?

“HIDDEN” (IMPLICIT) BOUNDARY VALUES

- The boundary values we have gone over already are explicit – that is, they are defined, or at least able to be deduced from, the requirements of the problem itself.
- Some boundaries are implicit – they are generated from the domain, architecture, hardware, or other elements:
 - MAXINT, MININT
 - Maximum precision of a floating point value
 - Allocation limitation (memory, hard drive space, network bandwidth, etc.)
 - Undefined values

BASE, EDGE, AND CORNER CASES

- **Base case** – An element in an equivalence class that is not around a boundary (interior value), OR an expected use case.
- **Edge case** – An element in an equivalence class that is next to a boundary (boundary value), OR an unexpected use case.
- **Corner case** (or **pathological case**) – A case which can only occur outside of normal operating parameters, or a combination of multiple edge cases.

BLACK-, WHITE, AND GREY-BOX TESTING

- **Black-box testing:** Testing with no knowledge of the interior structure or code of the application. Tests are often performed from the user's perspective, looking at the system as a whole.
- **White-box testing:** Testing with explicit knowledge of the interior structure and codebase, and directly testing that code. Tests are often at a lower level (e.g., testing individual methods or classes)
- **Grey-box testing:** Testing with knowledge of the interior structure and codebase of the system under test, but not directly testing the code. Tests are similar to black-box tests, but are informed by the tester's knowledge of the codebase.

BLACK-BOX TESTING EXAMPLES

- Accessing a website, using a browser, to look for flaws
- Running a script against an API endpoint
- Checking to see that changing fonts in a word processor shows the correct font

WHITE-BOX TESTING EXAMPLES

- Testing that a function returns the correct result
- Testing that instantiating an object creates a valid object
- Checking that there are no unused variables in a method

GREY-BOX TESTING EXAMPLES

- Reviewing code, and noticing that bubble sort is used. Then write a user-facing test involving a large input size.
- Reviewing code and noticing an off-by-one error. Then write a user-facing test which checks that boundary value.

STATIC VS DYNAMIC TESTING

- Dynamic testing = code is executed (at least some of it)
- Static testing = code is not executed

DYNAMIC TESTING

- If you're thinking about testing, this is probably what you are thinking about.
 - Code is executed under certain circumstances (e.g. input values, environment variables, etc.)
 - **Observed results** are then compared with **expected results**
- The majority of the class will consist of dynamic testing
- Much more commonly used in industry

STATIC TESTING

- The code is reviewed by a person or external program, without being executed
- Examples:
 - Code walkthroughs and reviews
 - Requirements analysis
 - Source Code Analysis
 - Linting
 - Model checking
 - Complexity analysis
 - Code coverage
 - Finite state analysis
 - ... COMPILING!