

# CS1632, LECTURE 2: TESTING THEORY AND TERMINOLOGY

Wonsun Ahn

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

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

n I send in the value  $a = 42$ , then I expect to be returned the value 074069841.

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

n 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 if there are a 1000 method calls?

- Assume each method call accepts one 32-bit int argument
- Remember that methods in a Java-like language can have side-effects (in other words, they are not pure functions)
  - Can access / modify global variables, global data structures
  - Means a method call is affected by all previous method calls

4,294,967,296 ^ 1000



# THAT’S EQUAL TO...

9117195078527900250972333896757874547991584670416109326631685858659056189397189966961858574196992852338720774576146913280019798175188631443936755178137562235518249413278412224206590457192312529626797715663223116251324543070355970530225215612468985466808580243659546869714311919061163821263834276348005068718825821539072017198844144188664879121575976272538936621655744682432287003146859158274839058210769233155307829795617096559227269323094215379537490643464429470932033797728358190898332940657131789926469772281206101938089923628772175396203549523069277892866886240474882457078218830750518210778400703003844510044466069981269617781194788757656025565590839684527043545238966575573403559687951326963892838818040660850711941690965464303296715997735315114393298314639765906325716916902265058485269112748232055283264916137426095401499226015801097484231501236124348080603774582483094786501010411491640990719808884284213685461771794474955271321378577110896128881183123766637948676781354618800664352404863944529565297885296338373785076024116351438271665187537455569914006769145607174827108759036217371186004470408462533609154350069171432391905236698147627643706567883856399887838188577300034105386260275801457249934804175739466067697350928216693063295588063141404637171376566737984126610669029129338339508145694851201748679819045973059635627618804971155841944169287806872340397299377605190620469299082702993326951264149037879410723677767085085619444240438253667000617814806792353740813593652081830044375906970007383971690175067381164968560489863912075463099176187624449475500457336543128916553631797587532936361060468731541106058332868386227534575757546324055632471212354205030687059326128124237306046177074749082672024398962791509181982441303121535821211536536703122879836190043609803359393931710850772544946057638672853757997423297492037562477818092323134340393692108391113953465946580012151301041990112340561101313016948710400033730407136415366537540139239157194155951689480400999761684077927466992190189673014569128322473733053895949364134343808332499059987264782814655949125056590781796556422804746351626979859227647826549850228866744876451447751151748843299619897972810381849869432292156102496685031462590708211491471479296713186360939930552616462546688521522562239741865470338206643246354269782985821882855897122309756003246254175934180385723179119775837176489887635751384552797529798395909029794651352218661771929377469552905704066393530933589229919879152423770199508041814825322272670194314379104030845758340779939556947096812250096343981167557247595961179331158114499843172888876422963195914969870701001192091500348562249270525324932249759835996854932467548669022136744548330632615545175351012218282500250610087479650864554497383187477097935862023421473171711484641968407993497083283831690153260491677702856729802838886234968356429597050907304768545315672469515022000434336650702482984461859711552244586736954362849354910430784086001137574524089776654393022315588593430316126939209496539832496535405867384177540225223711755360719671909215017724069262098002399293049541483064141164445566433084336365728781197999649274374069115041210530177145822340279705684452995471590064623375426786194425691105989123442653367209926836954315801613403893667175853502940802083873392270153115211059806226605339810254098811434005506705660238583972530934203626456667447050280426434407146084463844557676231376145368465076041733818936624509202579048128118406191307095564038850600579284504917698986384288769207486649326976202083059024913398259310137950925103411721183724586165347360741236024391227029422419550808057385315286510802421461538958166212637699038671974841781186417185583292599826273157124473664279204145614335673945730811403678069767978334196418139877986387102541577052705533642619524193522103705881975729399504837455081989422233431968334022138589080707248423554599048178031610185626620086056403468650129710989806595352943812558894932422584722634890931081243356161736189915233968129907439049015691222602274432246111150080709815183680219437123315744893585984196269623868667870456619797815214249387888730436839154265229066431013276550714550546307629571650128910962661940478119939159806757686024198773028092601855291898050622497661356201163684378850172037399421215068245561343325283565920125913117845503931374153590429155784531536202234916055276646438751180154328328060531981111512325152296537134540074768502466983752091404584861909367458874566508760789160702913223927728007529737526074388041982638528921416864247711572881283538001106835446598432243170893021429378896556776459613920903704732742879380906839175403579507438487492173508180824177776941630548235285428661242332197917303611620208937410064174567210975789006255852719174148137112432853653607132409351125376875023536625491435657405332431015194624386449532806906632881021058375409313310328828052299797160165646751011243872121225007265278212085985036217264423016184214285772212338048617448653777852935482168326246138678834903021147243678558424847547917888218371743369396014783628533549260842862242006394831929589376558734119082694146131929973825734930816757160607652858999523782850359528013781037135707033245059781779843643418441051467371181772423410381605071312319605959717198055966289591952347367442951661043611593606830361632702026345522135133185664848357194038802205540334117963493724958755861186562074355439961631871585269684037324711051390391856426740881383966526243258679881109313149829140840519784572581938192030868424048448489544723770504285296889152706313281183336545181993631850875416823300955401773995018507773585105419125331107410218042198964198023582750687672626691726643846467559433378486748034355735894735448732740215206929436336288047949817199533697551059594704233217150110001318775474062607992691664218910750509173361068365315450114841224548170357095043100966938379009438490468208206471876118259240141006749939528679101009642068826804739081392522992863894126858268721183661798903553001129685745033531325112429594472688307450860956162014281645160068425866751715879154129648113416699664100736806366615734402513769149128665561289906831190324490341764631114427994928700621776476627566933033662558631048914844785895775690812421576647529400247906293825945585169913923273890731266736364918635663674009747192507598364366197907583956098529262550066608744973038225506048304984084392525709414648686234392626319687626733205354843055202406699167437367835008362492667126547274623848872083424588918461360289531413558186883755973529609456498461553569091675076360406777996909483738743390919096780202382341491349958690903664927400588857090013645195142447218337339343001522596964326681440710596503992942213517556382772756747598017982013325874005680529870961701410431988078059862437989037819372549708563787372686046463074541443366212534369684813551693564936687254941994522934018195017525549444700868823993115636034505529461045301060444933263568782897349984594929215614636143283986198334944535415943867803617892583594295786574422412787501298212425098534552540823439766763227848153057333499357320813981479365688083368944747019143438124131845882652947130701837372024926901547452088765873433929233115894884953563621355080544011327838427191656226376352039722684020150152597638897706071847162525995050001361251800596808445848537576364111221755369986030360621366345532458637615648908140533087938693489508859067427442899832936297018998228449831175764807232863648922187671016113511956192598056441691913180793410205848151380066011457637056370685466178001877721777964585711894880734908956352709264672859611278224779278539925717788477311326910476981906648290017893205951288411887925068770625790971747969307882731161266884475424295600458204899268007845310782720071116397207300523582902653610102579928775995920867339265599343512486620119083979877907138001894289903312329699673216448527968615657988693515760373547767688171833634237729506529854937067332274928489685775067483177413099217325224086241646897700614729689569842886660014965235266326711701801800504856531587132403539956254412409180806602503647169822167767627131484831902013306071075709755617282913181677160254138644726949616404340448339909493558407040127968392448340867373837709208024468306602783309523986267881709073793534395906001155623195797517940306798159488933037116390502290223739627640910755195660574994249037253061953283743080917918101650526535030771660908223164471043406263905344031520182132013000761830308833178382109642733923978710854460357091072928739692277111756018614686106634321952938137996608194269116837528695547314498847939438988892172669177138946201684065003046090134922482829326394901367792896906172512634504104129260934135705385105010570411269933278405050148904762945084018233746533674326531779889523078039231457620050342359796836737280933673599690714097366767142822600738069576618516425065001621677166816257640842664690543274309036395271974904731268728621503370319375930779994574095357126317236912267575371995190901805593078496995016587764063047677076735739317412902591868417203837353116733201790299528625727536570678189495258646192458199440095291591283969428406912665432093120904443424706047907258156099896705642517840180529560725785258711059566856142051306707352558401612588999519443731178056039984753216078562003028506800494820623828617129028666264555632384902930091829247278449773030511674842151607851025821090742941514431736054731493262606532736024386291288197779696647358206451825116988631320595390442997608759132222017057289162026843895914872879450614315157356794604517875312204621626540978370154409519448837542811422945593310556603778896664452947773947257989782411129443834386341569639844449666759577829376

## TEST CASES!

# What if the argument is an object reference?

- That object could be a list data structure, a tree, a graph, ...
- How many shapes can a tree take?

Would testing all the combinations of arguments guarantee that there are no problems?

# LOL NOPE

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

# LOL NOPE

- Compiler issues

- Your source code does not run on the computer, the compiled binary does
- Depending on compiler and compile options, many different binaries can be produced! Besides the binary you used for testing.
- What if the compiler has a bug? (Rare)
- What if the compiler *exposes* a bug in your program? (More frequent)

```
int add_up_to (int count) {  
    int sum, i;  /* some C compilers will init sum to 0, some will not */  
    for(i = 0; i <= count; i++) sum = sum + i;  
    return sum;  
}
```

- Big problem for C/C++, less of a problem for Java since all bytecode is run on JVM (In above example, Java Virtual Machine always initializes all variables to 0)

# LOL NOPE

- Compiler issues

- Parallel programming issues

- If there is a data race, the result of your program is undefined
  - Doesn't matter whether you are using C/C++ or Java
  - Worst part: for many data races, result is correct 99% of time, masking the bug
- Even with no data race, the result of your program is often nondeterministic (Depending upon the respective speed of each thread)
  - To thoroughly test, you have to vary the speed of each thread, even for same input

# LOL NOPE

- Compiler issues
- Parallel programming 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.
- Defining what “sufficient test coverage” means is subjective.
- We must rely on domain knowledge to decide.



# 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 belong to multiple equivalence classes, you probably need another equivalence class
- Example:
  - Right handed people -> writes with right hand
  - Left handed people -> writes with left hand
  - Jane can write with both hands. Which equivalence class does she belong to?*
  - Solution: add “Ambidextrous people -> writes with both hands”

# Multiple partitionings

- Assume in the previous square root method, if the result contains a decimal point (e.g.  $1.3$  or  $2.23i$ ), it prints in **red**, otherwise in black.
- Now we have two partitionings:
  - The positive / 0 / negative partitioning on the previous slide
  - The decimal / non-decimal partitioning on this slide:
    - Number contains decimal -> output printed in **red**
    - Number does not contain decimal -> output printed in black
- A value belongs to two equivalence classes, but in different partitionings
  - Value  $1.3$  belongs to “positive” and “decimal” equivalence classes
  - But within a single partitioning,  $1.3$  belongs to a single equivalence class

# Values do not have to be numeric

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

# Values do not have to be numeric

Since 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.

Prevalence classes (NV = number of valid chars, NI = number of invalid chars):

$NV \geq 1, NI == 0$  ) -> Post the tweet

$NV == 0, NI == 0$ ) -> DO NOT post the tweet

$NV < 1$ ) -> 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?

- Suppose expected behavior is:
  - Method shall take citizenship and age as arguments
  - Method shall determine whether a person can be US president according to a set of rules
  - Rule 1: Person must be a US citizen to be US president
  - Rule 2: Person must be 35 years or older to be US president
- Suppose implementation is:

```
boolean canBePresident(int age, boolean citizen) {  
    return age > 35 && citizen;  
}
```
- Is observed behavior the same as expected behavior?



# Equivalence class partitioning

```
CANNOT_BE_PRESIDENT = [...  
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...]
```

Try to test both boundary and interior values

```
CANNOT_BE_PRESIDENT = [...  
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...]
```

- At the boundary values (shown in red)
- In fact, there is a bug at: `age > 35`

Try to test both boundary and interior values

```
CANNOT_BE_PRESIDENT = [...  
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...]
```

- Testing interior values is also important to see behavior in interior

Try to test both boundary and interior values

```
CANNOT_BE_PRESIDENT = [...  
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...]
```

• Are we done?

# “Hidden” (IMPLICIT) boundary values

- The boundary values we have gone over already are explicit – that is, they are defined by 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.)
  - Physical world boundaries (weight can't be negative, Y2K won't happen, etc.)
    - Side note: Y2K did happen and anti-gravity may yet happen

# Add implicit boundary values

CANNOT\_BE\_PRESIDENT =

[**MININT**, ..., -1, **0**, 1, ... 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**, ..., **MAXINT**]

- **MININT**, **MAXINT**: hardware boundaries
- **0**: physical world boundaries (age cannot be negative)

# 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 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
- Tests are performed at the code-level (e.g. testing individual methods or classes)

- **Grey-box testing:**

- Testing with some knowledge of the interior structure and codebase
- Knowledge may come from partial inspection of code or a design document
- Tests are performed from the user's perspective, but informed by tester's knowledge



# 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 works

# 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
- Checking that exceptions are properly caught and handled

# 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* in a web app and noticing user input is not properly sanitized. Then write a *user-facing test* which attempts SQL code injection or cross site scripting.
- *Reading a design document* and noticing a critical network connection through which a lot of data passes through. Then write a *user-facing test* that stresses that network connection.

# Static vs dynamic testing

- Dynamic testing = code is executed (at least the part that is exercised in that test run)
- Static testing = code is not executed

# Dynamic testing

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

# Static testing

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

Now Please Read Textbook Chapters 2-4