Lecture Notes
October 26, 2017

Implementation team assignment due tonight (submit even if you haven't demo'd yet)

Midterm 360-degree Feedback due Friday

Black-box unit testing and completion of first iteration due November 9 (another demo)

Unit Testing

The simple testing done by novice programmers, if they test at all, is typically system testing - the program is run from the command line or GUI with a few sample inputs

> These are usually test-to-pass

Focus of professional testing is test-to-fail, at all levels

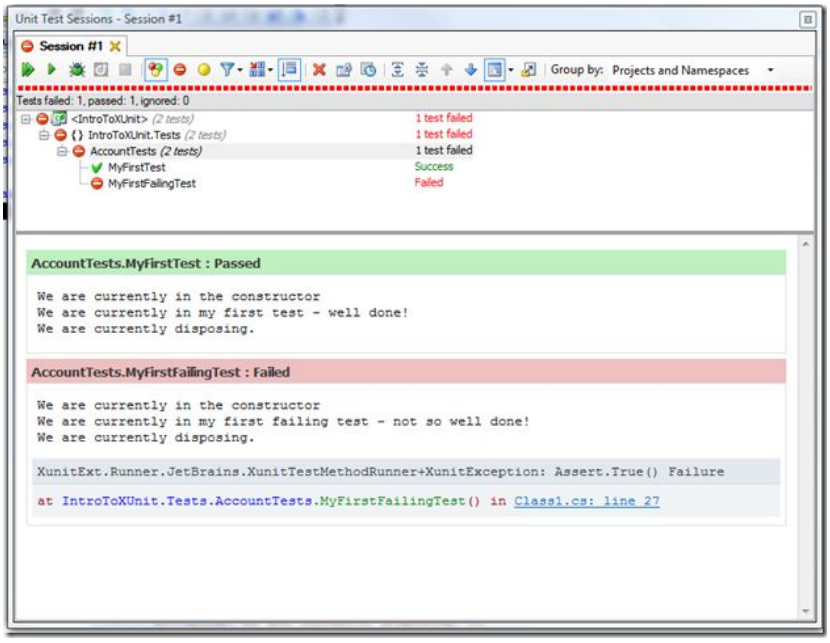
> If units (that are actually used by the full system) misbehave, it is unlikely that the full system behaves properly even if it appears to do so from a few system tests

> The initial unit tests are usually written by the developers who wrote or changed the unit, as part of completing the user story, but more extensive unit tests may be written by separate testers (e.g., to achieve coverage)

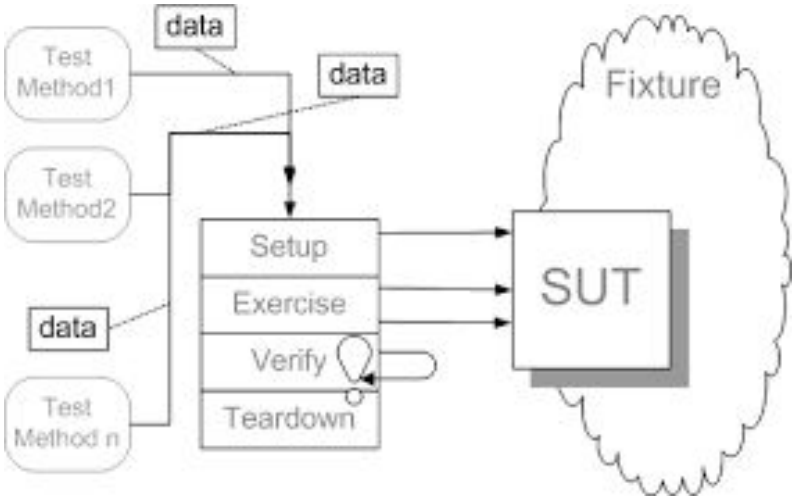
What is a “unit”? When I say unit, I’m usually thinking of a method, function, procedure, subroutine, but it is not unusual for software engineers to treat a class, module, or other program component as a unit for unit testing purposes

> There are almost always *multiple* test cases for each unit, not just one

Unit testing tools, often named “xUnit” for some language-specific x, enable running a set of unit tests on demand



Usually four phases for each test case: setup, exercise, verify, teardown



Exercise means actually running the test subject (“code under test”, “method under test”)

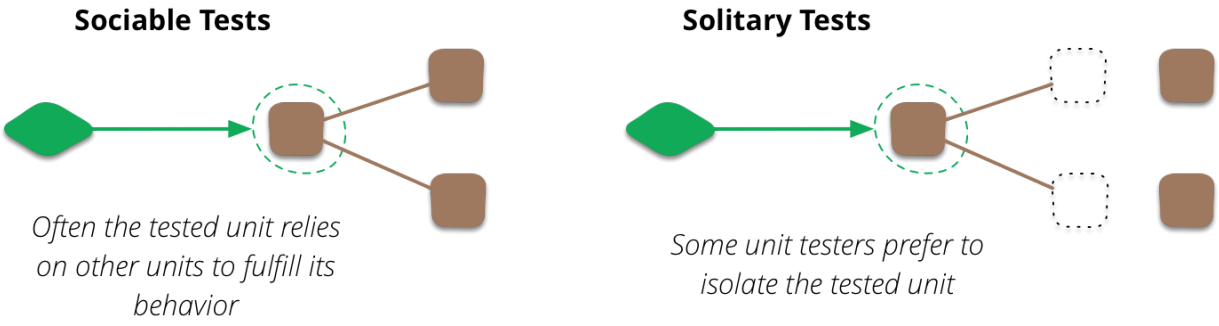
Verify means checking that the actual result matches the expected result (need a “test oracle” to know what is the expected result - for the kinds of applications considered in this class, the test oracle is usually the human tester)

The “result” is not necessarily a return value, may be a change to state that persists beyond the unit under test - State verification

(see Mocks Aren’t Stubs for discussion of mock-based Behavior verification)

Mocks and Stubs

"Sociable" tests vs. "Solitary" tests



For the purposes of this course, unit tests are solitary tests

Fowler's explanation of sociable tests (in https://martinfowler.com/bliki/UnitTest.html, assigned reading) is incomplete/misleading, because to truly test a unit in the context of the whole program, you need the code/context "above" that unit not just the code/context "below"

> I refer to full-context unit tests as "in-vivo" tests, vs. solitary tests as "in-vitro" tests, but you do not need to be concerned with in-vivo testing for this course

How do you *isolate* the tested unit, for solitary testing, if it calls other methods and/or reads state besides local variables and parameters and/or changes the values of parameters and other non-local state that persists after the method returns?

> And if you do manage to isolate the unit under test, how do you verify state changes to state outside that unit?

Fowler refers to "TestDoubles", which includes full range of fake non-local objects and methods that might be used during isolated unit testing

TestDoubles are needed for two main purposes:

1. Make the compiler/runtime happy, because the referenced objects and methods need to exist even if they are never used by a given test
2. Standing in for the relevant parts of the rest of the program (or external resources) that indeed are actually used during one or more unit tests, so need to be/do <something> useful for the given unit test

First consider “never used” case:

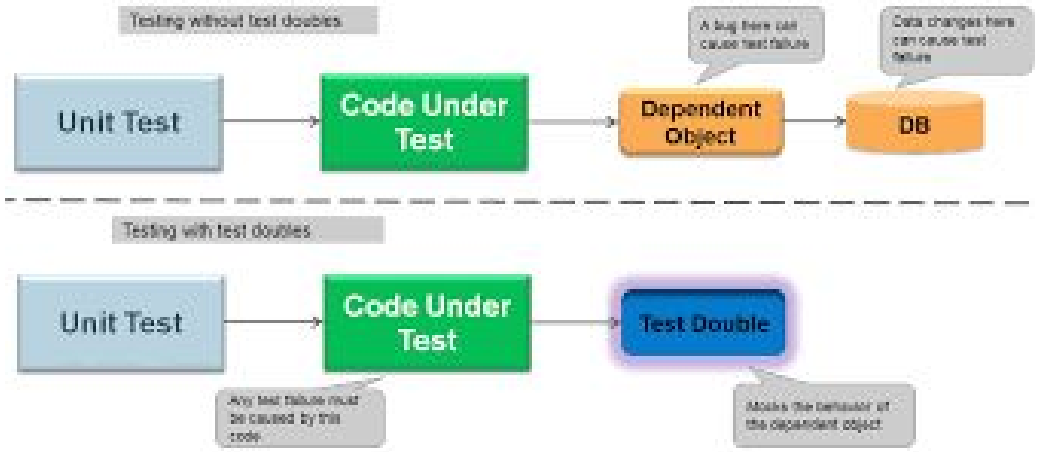
Any other methods that can be called anywhere during the method under test have to exist, even if never actually called during the given test case

> fakeMethod (same parameter list as real method) : same return type as real method
> { return fakeObject of correct return type }

Need to call the method under test with all parameters required by the compiler/runtime, so need to conjure up fake parameters during setup before the actual unit test begins and then get rid of these fake parameters during teardown after the unit test ends

If any part of a method accesses non-local state, even if that part is not actually executed during the given unit test, still usually need setup/teardown for this state

Now consider “actually used” case:



Stubs hardcode a specific state, operation and/or return value

Mocks can be configured to produce varying states and behaviors for testing purposes

Example with stubs:

We want to take an order object and fill it from a warehouse object. The order is very simple, with only one product and a quantity. The warehouse holds inventories of different products. When we ask an order to fill itself from a warehouse there are two possible responses. If there's enough product in the warehouse to fill the order, the order becomes filled and the warehouse's amount of the product is reduced by the appropriate amount. If there isn't enough product in the warehouse then the order isn't filled and nothing happens in the warehouse.

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse1() {
    Order order = new Order(TALISKER, 25);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(25, warehouse.getInventory2(TALISKER));
  }
 public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 25);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }


  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 26);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
```

Do you see any problems with these tests?

Same example with mocks (using record/replay, there are other approaches to configuring):

```
public class OrderEasyTester extends TestCase {
  private static String TALISKER = "Talisker";

  private MockControl warehouseControl;
  private Warehouse warehouseMock;

  public void setUp() {
    warehouseControl = MockControl.createControl(Warehouse.class);
    warehouseMock = (Warehouse) warehouseControl.getMock();
  }

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);

    //setup - expectations
    warehouseMock.hasInventory(TALISKER, 50);
    warehouseControl.setReturnValue(true);
    warehouseMock.remove(TALISKER, 50);
    warehouseControl.replay();

    //exercise
    order.fill(warehouseMock);

    //verify
    warehouseControl.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);

    warehouseMock.hasInventory(TALISKER, 51);
    warehouseControl.setReturnValue(false);
    warehouseControl.replay();

    order.fill((Warehouse) warehouseMock);

    assertFalse(order.isFilled());
    warehouseControl.verify();
  }
}
```

It is ok to use only stubs, not mocks, for this course - but if you would like to use mocks, find out which mock tool(s) play nicely with your team’s choice of implementation stack

Black Box Testing

Black box testing can be static or dynamic

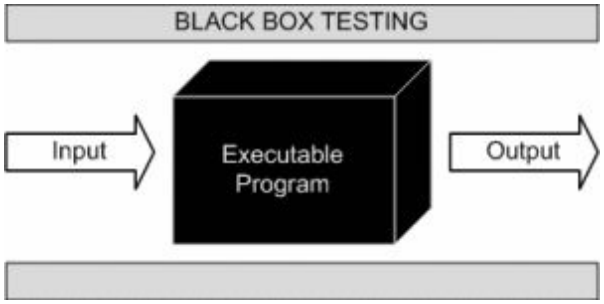
What does static mean here? Since its black box, we can't examine the code

> At full system level, we can review requirements and at unit level, we can review design
> ➔ Find bugs before code is written

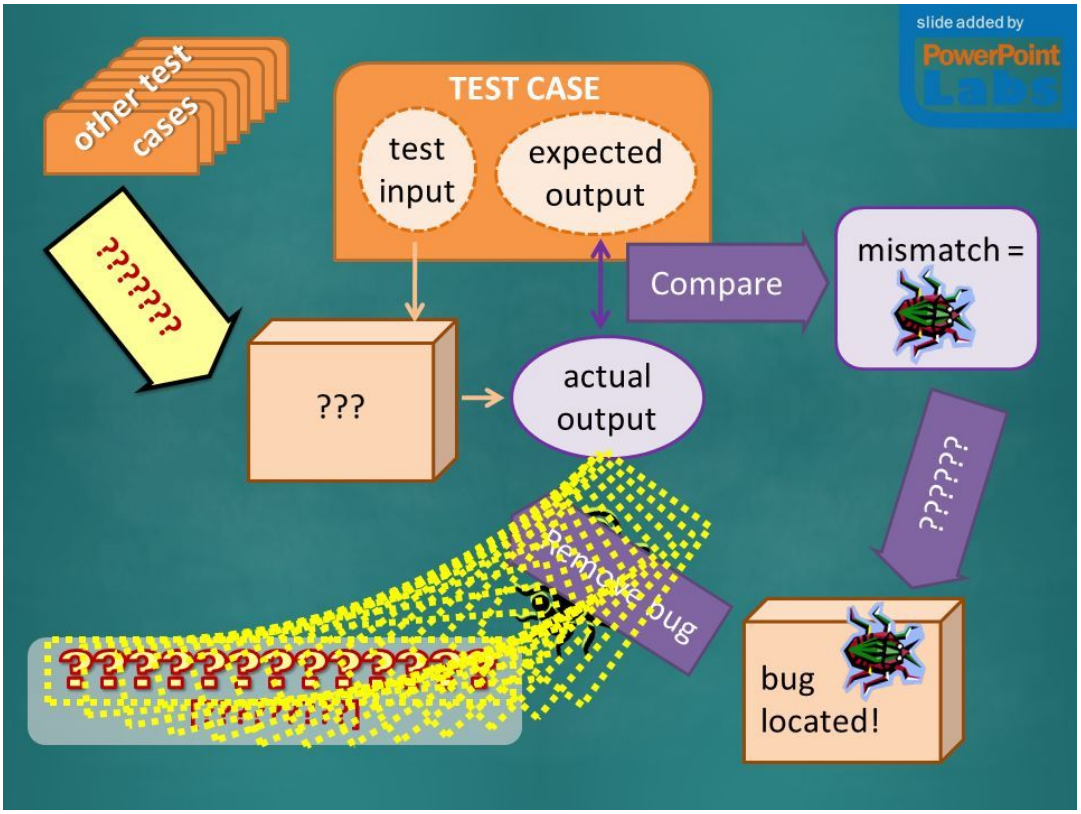
What does dynamic mean here?

> Running the code with sample workload (inputs) and checking the results
> ➔ What we normally think of as "testing"

Commonality of blackbox testing, regardless of the level = focus on inputs and outputs



For a given input, is the actual output *consistent with* the expected output?



Does not necessarily have to be identical, depends on semantics and inherent imprecisions (e.g., floating point computations), but for the purposes of this course will discuss as if “equal”

The expected output does not need to be known in advance if there is some way to check that the actual output is correct for the given input, but for the purposes of this course we will usually refer to expected output as if we know a priori what it should be

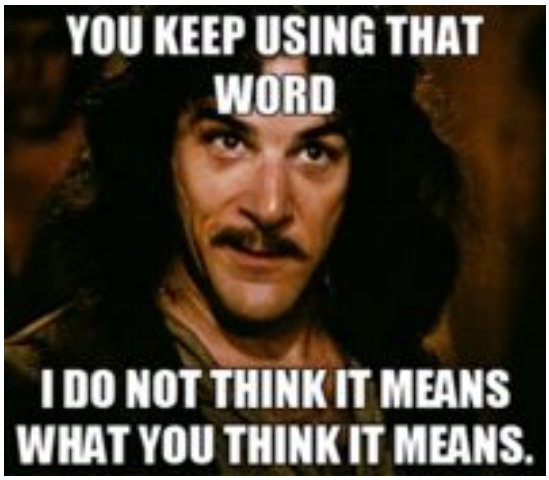
Choose set of inputs likely to uncover defects, where actual != expected

Cover all input points and all output points

Covering all input points does not necessarily cover all output points, there can be multiple distinct output points for the same input point and vice versa

For example, one input value entered at a given input point might result in the software returning an output value whereas another input value entered at the same input point might result in the software raising an exception and not returning any output value (consider division by zero)

What is an “input”?



Most inputs and outputs determined from the requirements (or, at lower levels, design) not the code - which is why test driven-development is possible, write the tests before writing the code

> For a method, the I/O is at least the parameters and return values
>
> For a GUI application, the I/O is at least the input provided to the GUI by the user and the output displayed by the GUI to the user

But some inputs and outputs may be implementation-specific, so usually considered only during graybox testing not blackbox (the tester needs to peek to find these I/O points)

> Non-local variable reads (inputs) and writes (outputs)
>
> Parameters to library/system API calls within the method (outputs)
>
> Return values from the external APIs (inputs)

Files, databases, network, devices, etc. might or might not be user-visible

> The word processing file being edited by the user is visible,
> the system log file probably isn’t

How do we know what the expected output is for a given input? Or how do we check that the output is correct? → Need a *test oracle*

Possible test oracles:

- Human tester
- Naive - if it doesn't crash, hang, or produce obviously nonsensical output, it works!
- Formal specification (mandatory for safety-critical systems, unlikely for business/consumer applications)

Sometimes tester does not know what the software is supposed to do = "exploratory testing", so only naive oracle is possible, or the software domain is too complex for practical human oracle

> *Pseudo-oracles* may be available even when there is no formal specification, e.g., another independently developed implementation of the same functionality
>
> If they disagree, one or both is wrong

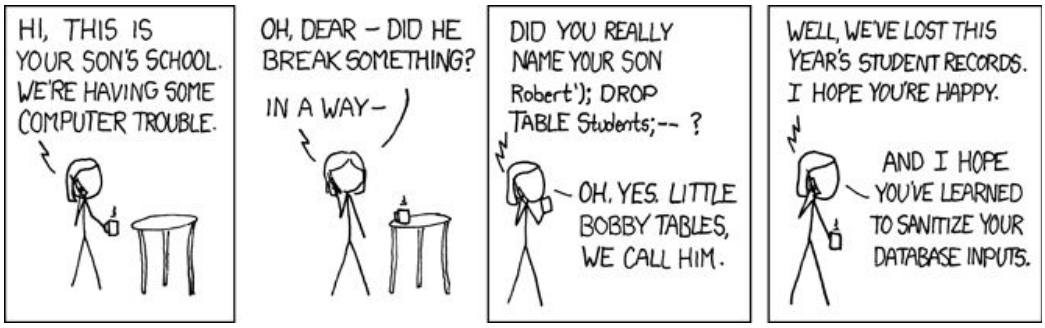
For the purposes of this course, we assume the human tester knows what the expected output should be for a given input and/or can check that the actual output is correct (beyond naive)

The test oracle, whether from human tester or otherwise, typically expressed in test cases as assertions (using a special assertion notation) or regular code

> Test oracles can themselves be buggy!

Oracle may be able to produce expected outputs (or check actual outputs) only for *valid* inputs, so we also need to make sure the software rejects all *invalid* inputs (or “sanitizes” = converts invalid inputs to valid inputs)

> Example invalid inputs for web application: accessing page via url hacking, back button, cache, browser history
>
> Most security exploits occur through carefully crafted invalid inputs that should have been rejected, but weren’t

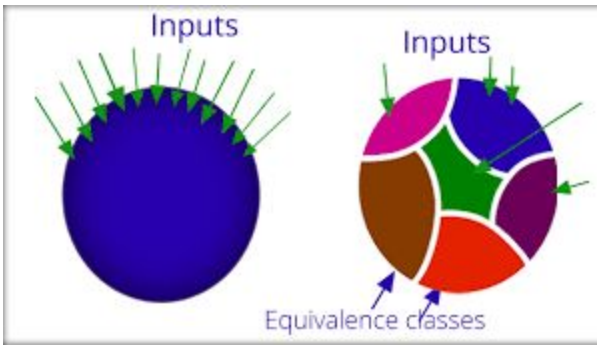


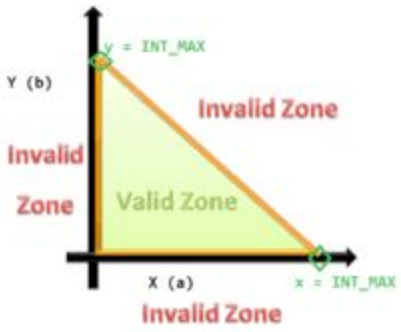
How do we put together a “good” set of test inputs, both valid and invalid?

Continue here

Equivalence Partitions

An “equivalence partition” (or equivalence class) is set of input values where we expect the software to behave in the same way - this does not necessarily mean literally the same value, but instead an analogous result





For calculator, we might expect the “+” operation to behave similarly on all positive integers (up to near maxInt)

> So we do not check 1+1, 2+1, 3+1, 4+1, … 1+2, 2+2, 3+2, 4+2, … but instead choose a few different values for both first and second parameters

But the calculator might behave differently for zero or negative integers, or nearing MaxInt and MinInt

> So positive, zero, negative, MaxInt, MinInt are different equivalence classes
>
> Maximum and minimum might be specific to the calculator implementation, rather than language-defined integer overflow

Floating point provides analogous equivalence classes, to some degree of precision (number of decimal points, exponent range)

But alphabetic characters are not valid inputs - if possible to input at all, alphabetic characters form at least one more equivalence class

What else may be possible to provide as input, assuming we have a full qwerty keyboard, with special function keys, and a multi-button mouse? For example, see https://www.alt-codes.net/

Say we have a calendar program that represents months as integers 1 to 12

Range 1..12 is one equivalence class, all valid inputs are in this class

Integers below range - zero and negative - is another equivalence class

Integers above range - 13 and higher - is yet another equivalence class

Non-integers represent one or more additional equivalence classes for invalid inputs

What would be some good test inputs for test to pass?

What would be some good test inputs for test to fail?

Say we have a different calendar program that represents months as strings, “January”, “February”, and so on

Instead of a *range* of valid inputs, we have a *set* of valid inputs

Instead of below and above range as invalid inputs, we have non-members in the set as invalid inputs - but some may be validly formatted as strings whereas others may not, so at least two different equivalence classes of invalid inputs

More generally for strings, we could check printing vs. non-printing characters, upper vs. lower case, space/tab characters, any characters in string with special meaning to the application (or to the implementation, e.g., shell or SQL), null string, max buffer size, buffer overflow

What would be some good test inputs for test to pass?

What would be some good test inputs for test to fail?

Say we have the same calendar program that represents months as strings, “January”, “February”, and so on

But now it allows abbreviations, so we have range of string *lengths* that are valid as well as the *values* of these strings

What would be some good test inputs for test to pass?

What would be some good test inputs for test to fail?

At the full system level, inputs/outputs are generally going to be numbers and strings, which inform the possible equivalence classes

> Possibly specially formatted strings such as network packets or JSON
>
> Possibly mouse movements, locations or clicks, possibly gestures - probably represented internally as tuples of numbers and strings

Files can be inputs/outputs at both full system and unit level

- Exists or not, min and max size
- Readable, writeable, executable (permissions)
- Correct format
- Images, audio, video, … - very complex collections of numbers, accompanied by metadata

Files containing “data sets” often correspond to a table (rows and columns) - with or without missing or repeated values, sorted or not sorted by a “key” column or set of columns

U.S. Department of Commerce
National Oceanic & Atmospheric Administration
National Environmental Satellite, Data, and Information Service

Record of Climatological Observations
These data are quality controlled and may not be identical to the original observations.

Station: **OLD FAITHFUL, WY US**
Observation Time Temperature: Unknown Observation Time Precipitation: 0800

Preliminary	Year	Month	Day	Temperature (F)			Precipitation(see **)					
				24 hrs. ending at observation time		at Observation	24 Hour Amounts ending at observation time				At Obs Time	
				Max.	Min.		Rain, melted snow, etc. (in)	Flag	Snow, ice pellets, hail (in)	Flag	Snow, ice pellets, hail, ice on ground (in)	M
	2015	1	1	17	-30		0.00		0.0		19	
	2015	1	2	30	-30		0.00		0.0		19	
	2015	1	3	19	-1		0.02		1.0		19	
	2015	1	4									
	2015	1	5									
	2015	1	6									
	2015	1	7	36	16		0.00		0.0		24	
	2015	1	8	39	13		0.00		0.0		23	
	2015	1	9	41	-4		0.00		0.0		23	
	2015	1	10	33	-3		0.00		0.0		23	
	2015	1	11									
	2015	1	12	33	23		0.02		0.0		24	
	2015	1	13	32	-7		0.00		0.0		24	
	2015	1	14	33	-12		0.00		0.0		24	

https://youtu.be/Qxf3xzirBrs

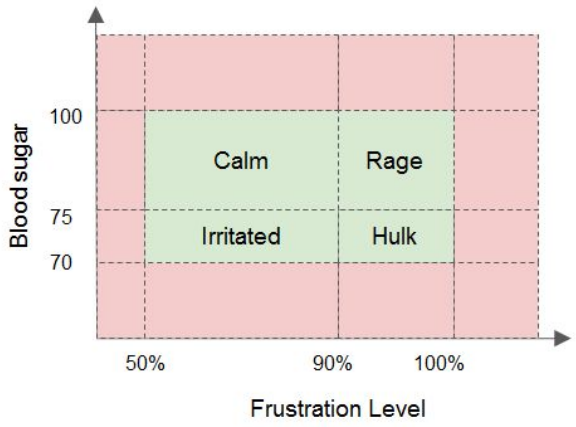
Equivalence classes also need to address required vs. optional inputs, e.g., on a web form

Equivalence classes may cut across multiple inputs:

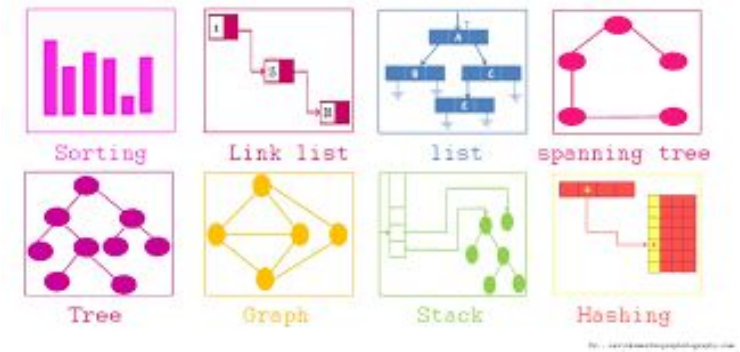
- Sometimes the value of one input determines appropriate values for other inputs
- For example, date of birth and age should be consistent (valid), or not (invalid)

Distinct parameters may be related to each other in forming equivalence classes

Let's say we had a function that took two variables (blood sugar level and level of frustration) that calculated "mood".

Assume valid inputs are ●Blood sugar 70-100 ●Frustration is 50% .. 100% For 50 <= F <= 90 75 <= BS <= 100: Calm BS < 75: Irritated For F>90 75 <= BS <= 100: Rage BS < 75: Hulk Smash	

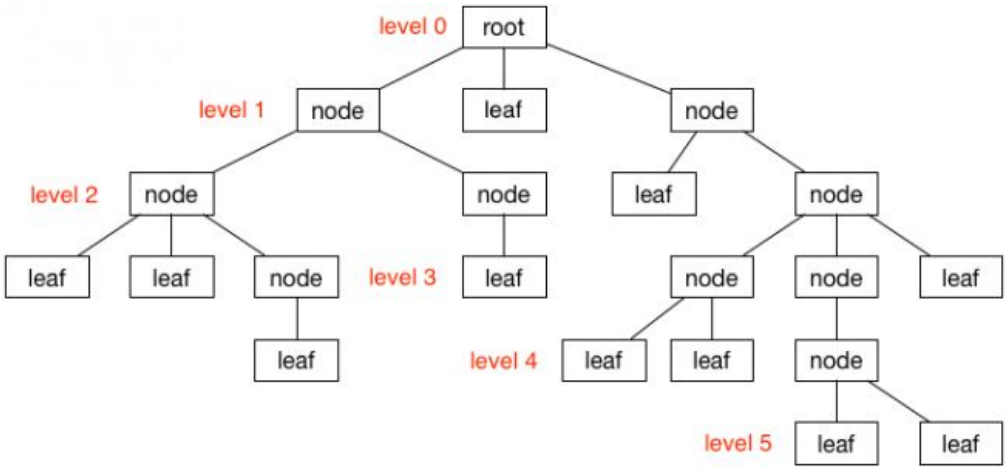
At the unit level, inputs/outputs can be arbitrary data structures as well as primitives, strings and tuples, so equivalence classes can be more complex - and inputs more complicated to construct



Container data structures (or file) with set of content elements

- In or not in container
- Empty or full container (or min/max number of elements)
- Duplicates may or may not be allowed
- Must contained elements be in some order or other organization?

Specific kinds of containers, e.g., tree - root node, interior node, leaf node, null tree, tree with exactly one node, “full” tree, balanced vs. unbalanced, various specialty trees - and graphs



How would test cases construct root, node and leaf objects to use as input?

It is possible that some of the sample inputs constructed at unit level could never actually be supplied when the unit is executed within the full program

> If those inputs find bugs, are these false positives or true positives?

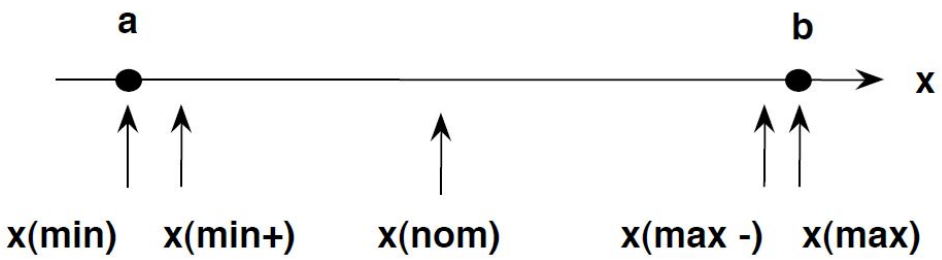
Boundary Analysis

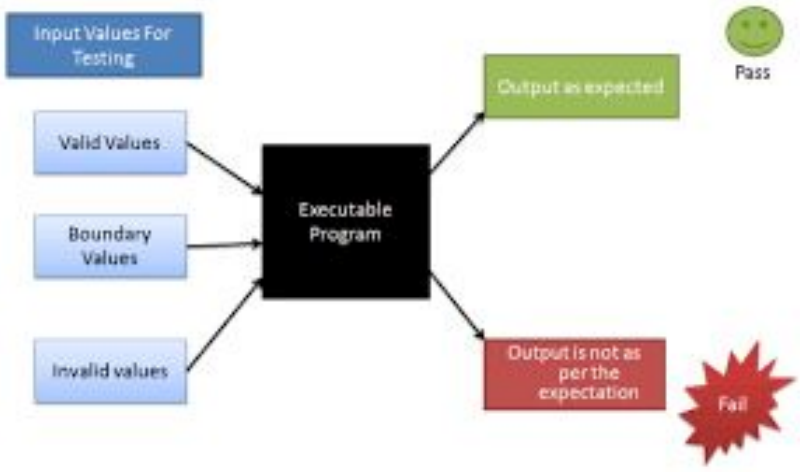
For a given input point, need to test at least one value from every equivalence class, but which one(s)?

Errors often occur at the *boundaries* of ranges: off-by-one error, fencepost error, corner case (e.g., some condition in the code checks < but should check <= or vice versa)

Do in addition to checking a value “in the middle” of each range equivalence class, also need to check at both boundaries of each range partition

- min, min+1, max, max-1 → what about min-1 and max+1?
- 2^N-1, 2^N, 2^N+1





Consider also how data is represented in memory, may be hidden boundaries when going from one byte to one word, from single to double

Consider the valid/invalid ranges for ascii (and unicode)

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Wow, that's a LOT of test cases!! Production-quality code often has three times as many lines of test code as application code

Issue Tracking

Where do bug reports go?

Email is a *bad idea*: hidden, not available to later developers, often disorganized

Task board works ok within an iteration, but generally disappears thereafter (replaced with new task board for next iteration)

Issue tracking intended to address the problem, maintains permanent records of “issues” and how they were dealt with (and by whom)

Workflow: create/open -> in-progress -> resolved/closed -> possibly reopened

Reported by someone, assigned to someone (answers these questions and more)

> Can also record time spent
>
> Can link to other issues and related materials
>
> Can reference commit records in version control repository and vice versa

What does a good bug report look like? See https://marker.io/blog/bug-report-template/

Example (most issues have been closed):
https://github.com/Programming-Systems-Lab/phosphor/issues?q=is%3Aissue+is%3Aclosed

Example (many issues remain open)
https://github.com/randoop/randoop/issues

If your team is using github, use their “issue tracker” to record bugs and other problems as they arise

If not using github, find some other issue tracker if there isn’t one already provided by your team’s git host