Lecture Notes
October 31, 2017

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

Second Iteration Plan due November 14

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

Choose one representative - assume if one works, all will work

In white box terms, the same execution path - but using black box you do not (explicitily) know the conditionals that determine execution paths, instead consider the criteria that distinguish alternative/exceptional flows from basic flow in use cases

Equivalence Partitioning Examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, $13} Empty value
Phone Number Area code: [200, 999]	555-5555 (555)555-5555 555-555-5555 200 <= Area code <= 999	Invalid format 5555555, (555)(555)5555, etc. Area code < 200 or > 999 Area code with non-numeric characters

Example of Equivalence Partitions

- Windows filename can contain any characters except \/ : * ? " < > |.
- Filenames can have from 1 to 255 characters.
- Creating test cases for filenames:
  - Will have equivalence partitions for valid characters, invalid characters, valid length names, names that are too short and names that are too long.

For calculator, we might expect the “+” operation to behave similarly on all positive integers (up to at or 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 whereas others may not, so at least two different equivalence classes of invalid inputs

What does formatting mean here? 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 are the equivalence partitions?

Design-your-burger

Please make your selection

Burger: beef

Cooked: rare medium well

Cheese:

Lettuce:

Tomato:

Onion:

Ketchup:

Mustard:

Mayo:

Secret sauce:

Nutrition facts | Reset | Submit | About the chef

What are the equivalence partitions?

At the full system level, inputs/outputs are generally going to be numbers, strings/text, and (for system-to-system inputs) tuples consisting of numbers and strings and - more generally - bits and bytes, which inform the possible equivalence classes

GUI mouse movements, locations or clicks, or (touch-sensitive) gestures

System-to-system inputs typically specially formatted, e.g., network packets and JSON

```
{
    "Rail Booking": {
        "reservation": {
            "ref_no": 1234567,
            "time_stamp": "2016-06-24T14:26:59.125",
            "confirmed": true
        },
        "train": {
            "date": "07/04/2016",
            "time": "09:30",
            "from": "New York",
            "to": "Chicago",
            "seat": "57B"
        },
        "passenger": {
            "name": "John Smith"
        },
        "price": 1234.25,
        "comments": ["Lunch & dinner incl.", "\"Have a nice day!\""]
    }
}
```

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

- Exists or not
- Min and max size
- Readable, writeable, executable (permissions)
- Correct data and metadata formats - images, audio, video, …

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)
	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 (with or without default values), e.g., on a web form

{ NUL | 👹 | … }

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 (or ordered sets): off-by-one error, fencepost error, corner case (e.g., some condition in the code checks < but should check <= or vice versa)

Check the “borders” of an equivalence class in addition to checking a value “in the middle”

min, min+1, max, max-1 → what about min-1 and max+1?

Three test cases result from every boundary

2^N-1, 2^N, 2^N+1

Consider how data is represented in memory, may be hidden boundaries when going from one byte to one word, from single to double (called “sub-boundary conditions”)

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

> 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 arise

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

Debugging

First step: get over your belief that your code is right and the computer is wrong

> Bugs are very rare in hardware, compilers, interpreters, runtime environments, standard and widely-used libraries - except perhaps in newly released versions, but large numbers of (other) users quickly shake them out
>
> What you told the computer to do was wrong

Second step: use an [interactive debugger](), log analyzers, assertions - not print statements

> In some languages, introducing new print statements can perturb memory state in such a way that the bug is disguised
>
> In any language, introducing print statements changes the code - modern languages are designed to prevent semantic changes due to built-in (or standard library) debugging statements, but cannot help if you introduce new code that cannot be distinguished from application code
>
> In any application with multithreading/multiprocessing, introducing *anything* new to runtime (including an interactive debugger or logging) can perturb timing and scheduling in such a way that the bug is disguised - need special techniques

Best case: Your unit test discovered the bug, before commit, so you know the bug *manifests* in that unit - and if your unit tests are properly isolated and independent, then the root cause (the underlying programming defect) *is also in that unit*

Medium case: Integration or systems testing discovered the bug, so the unit where the bug manifests is **not** necessarily the same unit where the root cause resides - and in some cases the root cause may be spread among multiple units

Worst case: Your customer detects the bug after deployment, which usually means your test suite is inadequate to uncover that bug - so you need to determine how to reproduce the bug, and then introduce a new test case that reliably detects that bug, *before* trying to fix the bug

> Try to find the smallest, simplest test case the detects the causes the same behavior
>
> When inputs are “large”, e.g., words in word processor, HTML in browser, cells in spreadsheet, number of shots in shooting game, search for the “smallest” input by systematically removing parts and checking whether remainder still causes bugs (called delta debugging)

Finding a bug is a process of checking your beliefs about the program

> You might believe that a certain variable has a certain value at a certain place
>
> You might believe that a function was called with particular parameters
>
> You might believe that a particular execution path is taken through conditionals

You can use assertions and interactive debugging to check specific beliefs, but you cannot check everything

Crashing bugs are typically relatively easy to find, if repeatable - run application in debugger, which is automatically triggered at point of crash so you can check memory state

> Or “post-mortem debugging” by attaching debugger to logged trace or memory dump, works best for dynamic languages (see http://www.almarklein.org/pm-debugging.html)

To “check” memory state, you need to know what you are looking for, what are your beliefs at this point in the program execution?

> This is the main value of including assertions with every method, because at the time you are writing the method you usually know what your beliefs are regarding the parameters it is passed, global state accessed, and return values of system/library calls

Non-crashing bugs may not announce themselves until much later, with user-visible output, so you need to work backwards along execution path

> “Binary search” between known buggy location (that produced the output) and known good location (analogous to delta debugging)
>
> How did the execution get here? Through control branches, through function calls, through “slices”

Hardest to find bugs where the root cause is *missing* code - where should this code be?