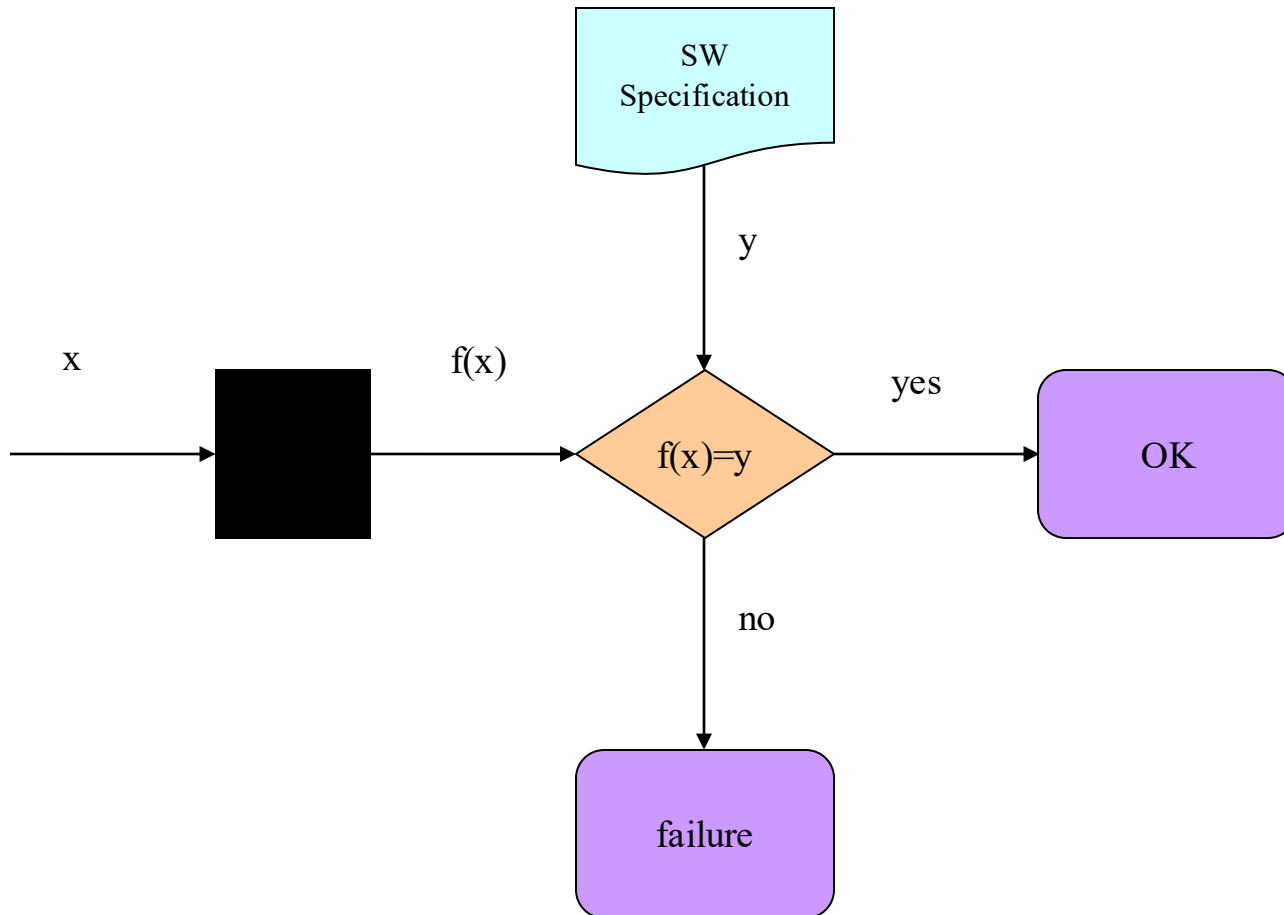


BLACKBOX TESTING

Week 9

Black Box Testing



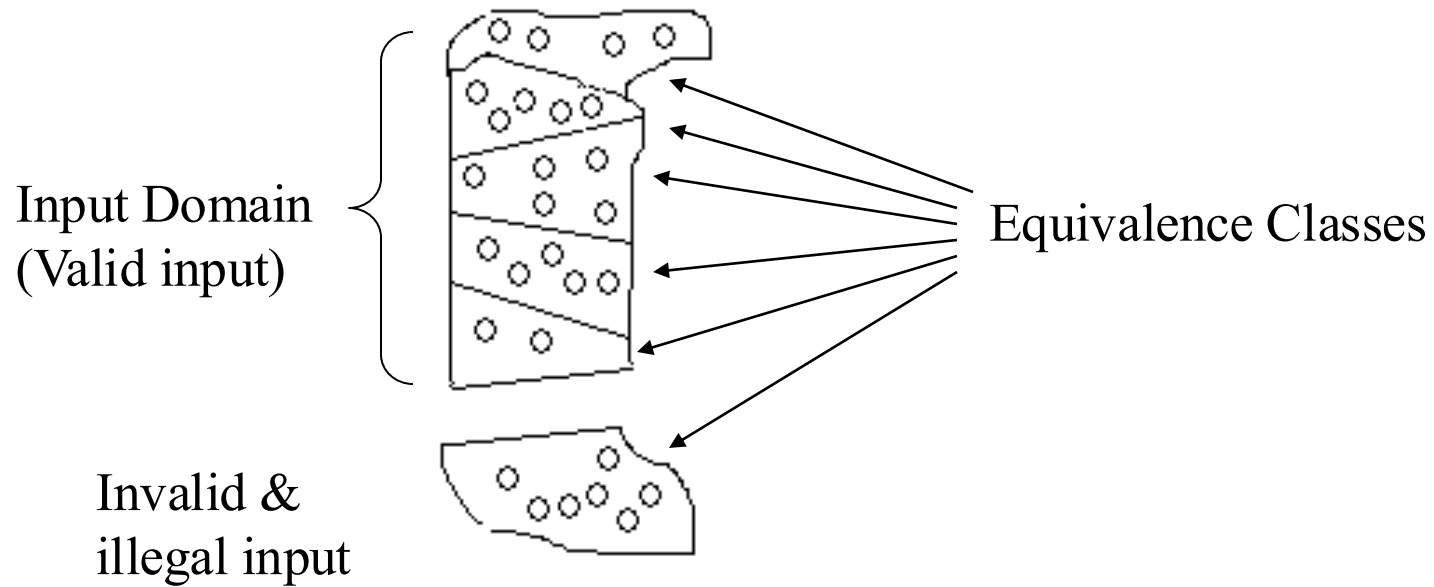
Principles

- Based on *specifications* and *documents*
 - *requirements*
 - *technical plans, architectures*
 - *user manuals*
- *Code not necessarily needed* (while it certainly helps)
- General strategy; applies especially to
 - *integration testing, system testing, acceptance testing*
- Can be assisted by a post-white-box testing phase, to obtain code coverage measures as indicators of testing quality

Domain partitioning: Equivalence classes

- *System domain*: set of *all input values*
- *Equivalence class*: *certain set of input values* (subset of domain, *subdomain*)

Equivalence Classes (ECs)



Equivalence Classes

- Each EC *represents a central property* of system
- each value in an EC makes system *behave “in the same manner”*
 - in testing, *each* value reveals a *failure* or makes system behave *ok*
- each value activates (almost) the same execution path through the system
- based on
 - *system’s specification* and
 - *experience / intuition of tester*

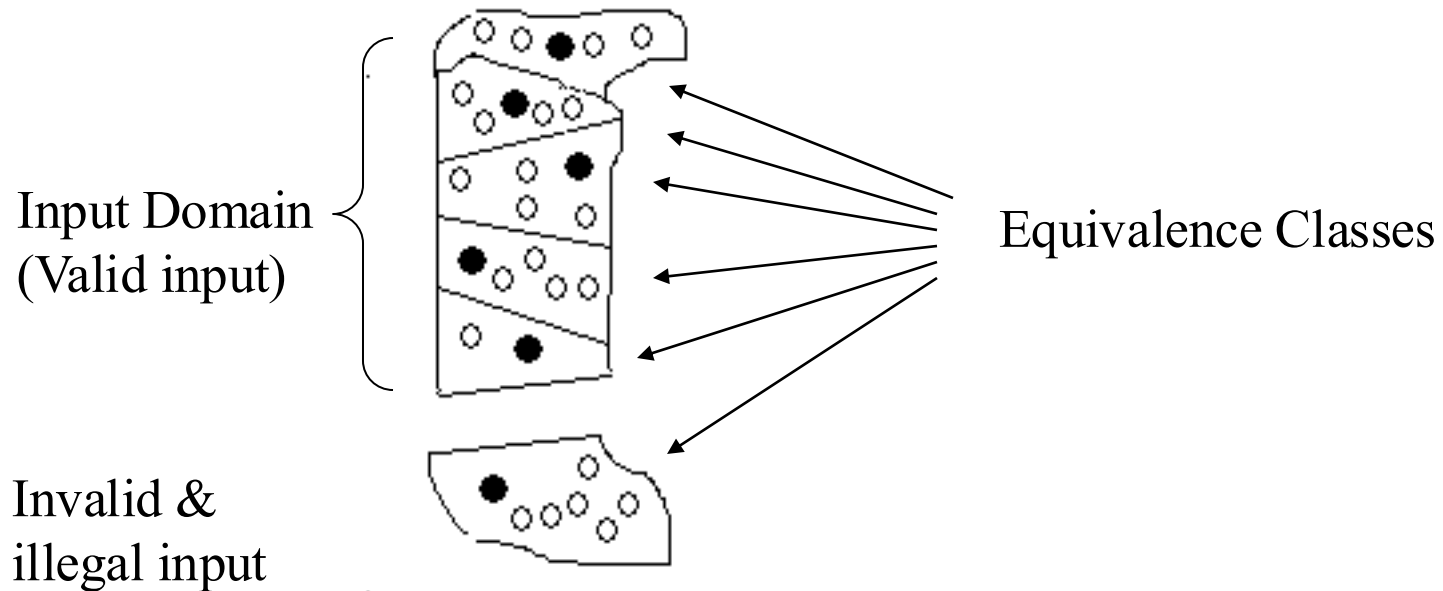
Black-box testing hypothesis

- each value in an EC results in
 - *correct execution*, or
 - *failure*when used as input to system
- *for testing purposes*, one representative input value from each EC is enough!
- *in practice*, the hypothesis does not hold universally, so *system shall be tested with several input values from each EC*.

Equivalence Classes (ECs)

- Each “*black dot*” represents the equivalence class it is in.
- Testing the code using a black dot will result either
 - in a failure or
 - OK

and represent the entire equivalence class.



Forming equivalence classes (ECs)

- To specify: a *range of values*
- Corresponding ECs: *one valid and two invalid classes*
- **Example 1:** “ $a \leq x \leq b$, x an integer”
 - Valid EC: $\{\text{integer } x \mid a \leq x \leq b\}$,
 - Invalid EC: $\{\text{integer } x \mid x < a\}$,
 - Invalid EC: $\{\text{integer } x \mid x > b\}$
- To specify: a *specific value within a range*
- Corresponding ECs: *one valid and two invalid classes*
- **Example 2:** “value of integer x shall be t ”
- Valid EC: $\{\text{integer } x \mid x = t\}$,
- Invalid EC: $\{\text{integer } x \mid x < t\}$,
- Invalid EC: $\{\text{integer } x \mid x > t\}$

Forming equivalence classes (ECs)

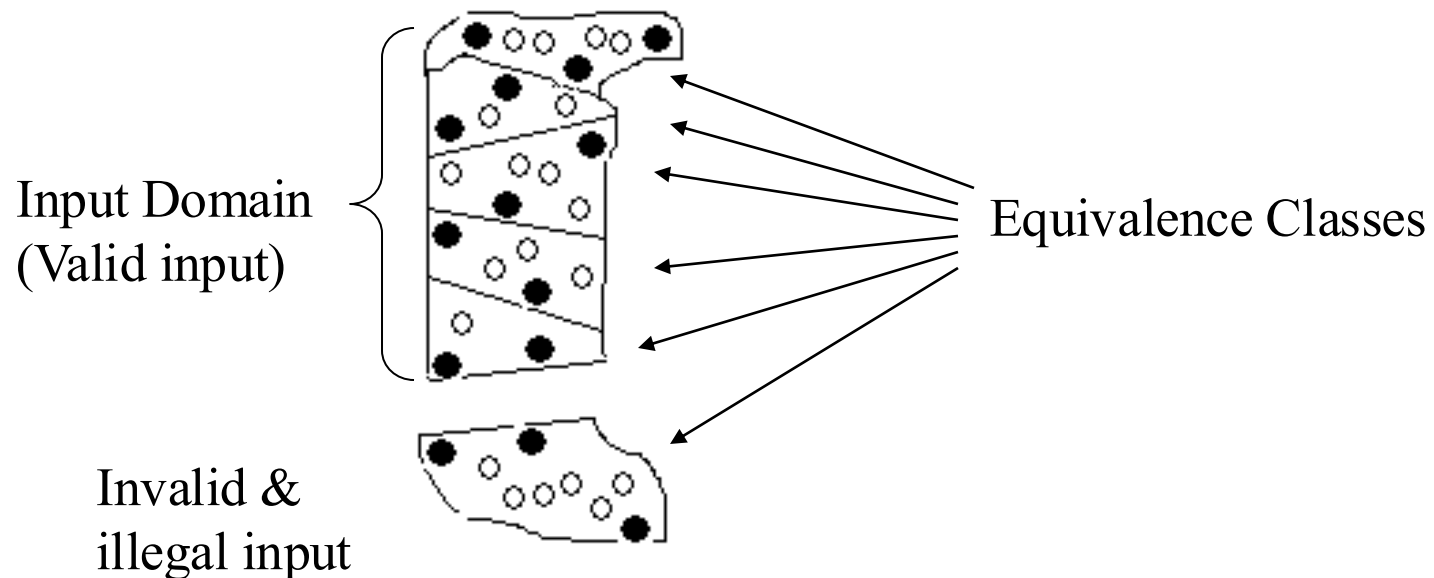
- To specify: a ***set of values***
- Corresponding ECs: ***one valid and one invalid classes***
- ***Example 3:*** “2D geometric shape x shall have 4 corners”
 - Valid EC: $x \in \{\text{square, rectangle, trapezoid, parallelogram, ...}\}$,
 - Invalid EC: $x \in \{\text{ellipsoid, circle, triangle, pentagon, hexagon, heptagon, ...}\}$,
- To specify: a ***boolean value***
- Corresponding ECs: ***one valid and one invalid classes***
- ***Example 4:*** “ x shall be true”
 - Valid EC: $x = \text{true}$
 - Invalid EC: $x = \text{false}$

Forming equivalence classes (ECs)

- one or more ECs for *illegal* values (i.e., values incompatible with the type of the input parameter and therefore out of the parameter's domain)
- **Example:** “integer values x ”
 - Illegal EC: real-number x
 - Illegal EC: character-string x
- **How many ECs?**
 - As many as the *potential groups of values* that are believed to be *handled by the system in different ways*.
 - Any EC shall be further divided into subclasses if there is reason to believe values in different subclasses are not processed identically by the system.

Boundary analysis

- EC boundaries are where bugs critically show up. That's why boundary conditions are subject to test.
- Each “*black dot*” represents a boundary condition of its relevant EC it is in.



Boundary Conditions

- ***open boundaries***: generated by inequality operators ($<$, $>$)
- ***closed boundaries***: generated by equality operators ($=$, \leq , \geq)
- ***on point***: value that lies on a boundary
 - for open boundaries: the boundary value; for instance $x > 0$
- ***off point***: value not on a boundary
- ***“one-by-one” domain testing strategy***: one on point and one off point for each domain boundary

Selection rules for on and off points:

- **open boundary: one on point and one off point**
 - **on point:** a value outside the domain \Rightarrow the condition is *false*
 - **off point:** a value inside the domain \Rightarrow the condition is *true*
- **closed boundary: one on point and two off points** (on both sides of the boundary, as close as possible)
 - **on point:** a value inside the domain \Rightarrow the condition is *true*
 - **off point:** a value outside the domain \Rightarrow the condition is *false*
- **nonscalar type: one on point and one off point**
 - enumerations, Booleans, strings, complex numbers, ...
 - on point: the condition is *true*
 - off point: the condition is *false*
 - the difference between on and off values should be minimized (for instance, for strings a single character difference)

Examples

- **range of values:** two boundary conditions
- **“integer x shall be between a and b ”** \Rightarrow
 $\{\text{integer } x \mid (x \geq a) \cup (x \leq b)\}$: $(x \geq a)$, $(x \leq b)$ are closed boundaries
 - **on points:** a, b
 - **off points:** $a-1, a+1, b-1, b+1$
- **strict inequality operator** \Rightarrow open subdomain
“integer x shall be greater than a ” $\Rightarrow \{\text{integer } x \mid x > a\}$
 - **on point:** a
 - **off point:** $a+1$

Examples

- **specific value:** one closed boundary condition
 - “value of integer x shall be a ” \Rightarrow {integer x | $x = 100$ }
 - *on point:* a
 - *off points:* $a-1, a+1$
- **set of values** \Rightarrow *nonscalar type*
 - “weekday x shall be a working day” \Rightarrow
 - $x \in \{\text{Monday, Tuesday, Wednesday, Thursday, Friday}\}$
 - *on point:* Friday, *off point:* Saturday
- **Boolean** \Rightarrow *nonscalar type*
 - *on point:* true, *off point:* false

The category-partition method

- *systematic black-box test design method*
- *based on domain partitioning* of input into *ECes*.
- ***Steps***
 - i. Specification of input *categories* or “*problem parameters*”**
 - ii. Division of categories into *choices* = equivalence classes**
 - iii. Test specification:***
 - iv. Generation of test cases for the test frames into executable form (using a tool), combination into *test suites*.**
 - v. Storing the testware into a test database.***
 - vi. Testing of the unit by the test cases, refinement of conflicting choices, maintenance of test database (using a tool).**

Array Sorting Example: Steps

i. **Specification of input *categories or*** ***“problem parameters”***

– ***Array sorting categories:***

- *size of array*
- *type of elements*
- *maximum element value*
- *minimum element value*
- *position of maximum element in the array*
- *position of minimum element in the array*

Step 2: Division of Categories

ii. Division of categories into *choices* = equivalence classes

– *Array sorting / choices for size of array:*

- $size = 0$
- $size = 1$
- $2 \leq size \leq 100$
- $size > 100$
- (“*size* is illegal”)

Step 3: Test Specification

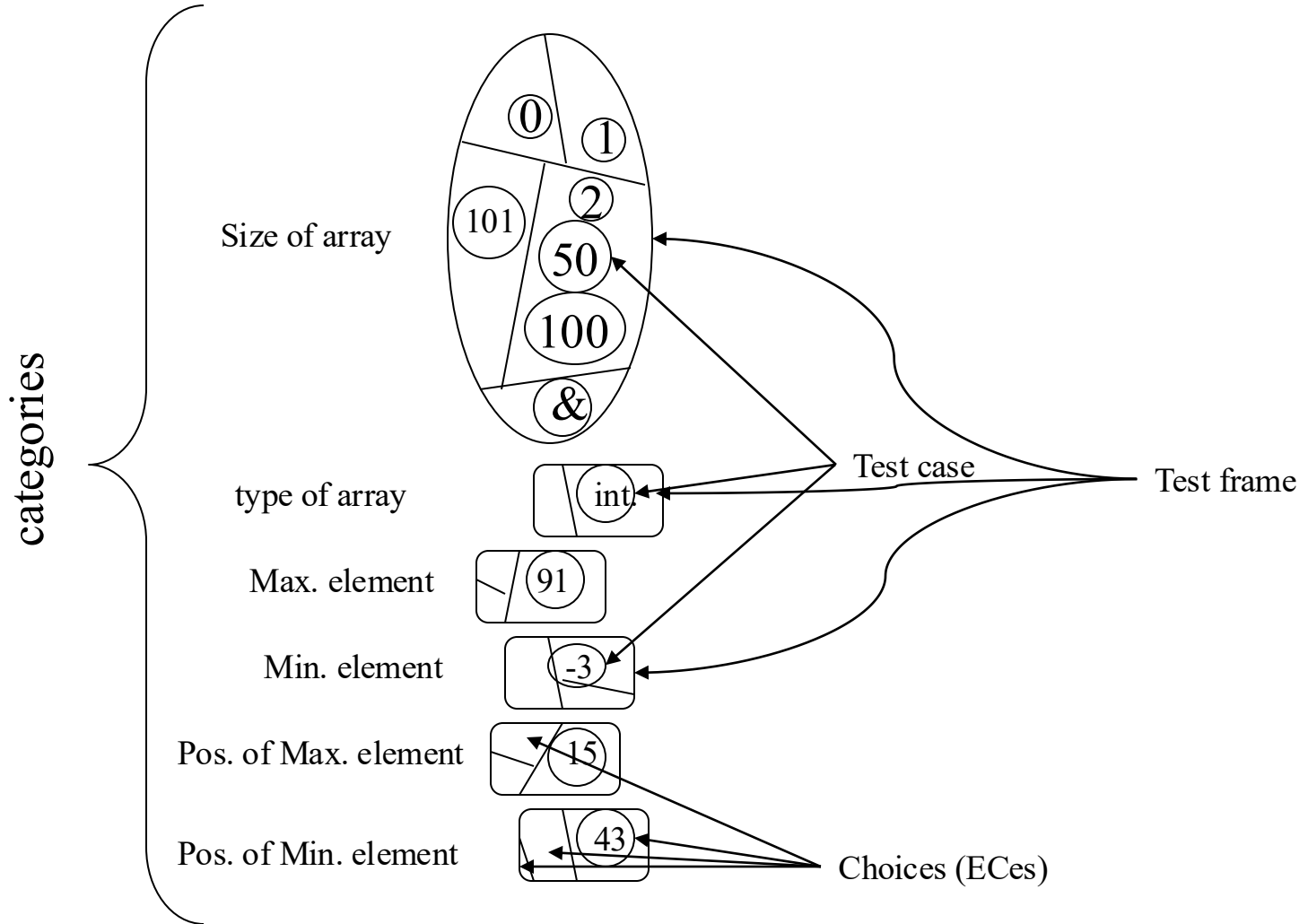
iii. Test specification:

- A *set of test frames*: sets of choices, with each category contributing either zero or one choice.
- A *set of test cases*: a single value from each of the choices in a test frame.
- *Array sorting example* / test case:
 - *size of array* = 50 (choice: $2 \leq \text{size} \leq 100$)
 - *type of elements* = integer
 - *maximum element value* = 91
 - *minimum element value* = -3
 - *position of maximum element in the array* = 15
 - *position of minimum element in the array* = 43

The category-partition method

- (4) ***Generation of test cases*** for the test frames into ***executable form*** (using a tool), combination into *test suites*.
- (5) ***Storing the testware into a test database***.
- (6) ***Testing of the unit*** by the test cases, refinement of conflicting choices, maintenance of test database (using a tool).

Example



System testing / GUI testing:

- **target:** *operations* available at the (graphical) user interface
- **parameters of operations** divided *into equivalence classes*
- testing by all different **combinations of equivalence classes** (with one input value from each class)
- **testing of operation sequences** (not independent)
- based on user's manual
- supported by tools (capture / replay)

Example: Find String in Document

- ***Find* (document, text, direction, match case)**
- *document*: the current text file, subject to search
- *text*: the character string to search for
- *direction* (*down*, *up*): direction of the search with respect to current position of the cursor
- *match case* (*yes*, *no*): whether or not the operation is case sensitive to letters

Equivalence classes

- **Input categories for various input**
- *text*:
 - {strings with lower-case letters but without upper-case letters}
 - {strings with upper-case letters but without lower-case letters}
 - {strings with both upper-case and lower-case letters}
 - {strings with no letters}
 - {empty (illegal) strings}
- *direction*: {*down*}, {*up*}
- *match case*: {*yes*}, {*no*}
- *document*: {text found}, {text not found}

Example

text					drcn		c. mtch		dcmnt	
lc	uc	luc	nlu	ϵ	d	u	n	y	f	n-f
☺					☺			☺	☺	
☺					☺			☺		☺
☺						☺		☺	☺	
☺						☺		☺		☺
...										
	☺									
	☺									
	...									
				☺	☺		☺			☺

How many tests?

- *# of (independent) combinations = Total number of tests*
- $E_1 * E_2 * E_3 * \dots * E_k$
 - with $E_i = \#$ equivalence classes for parameter i
- *For find example: $5 * 2 * 2 * 2 = 40$ tests*
- Some invalid, illegal combinations that might be unexecutable must be tested too!

Test Case Patterns

- *text: lower-case, direction: down, match case: yes, document: found (1)*
- *text: lower-case, direction: down, match case: yes, document: **not found (2)***
- *text: lower-case, direction: **up**, match case: yes, document: found (3)*
- *text: lower-case, direction: up, match case: yes, document: **not found (4)***
- ...
- *text: **empty**, direction: up, match case: no, document: not found (40)*

Selection of (40) test cases:

- each pattern generates a test case
- each equivalence class in a pattern is realized as an input value in the corresponding test case
- in different test cases, different values are selected for the same equivalence class (better coverage)
- boundary values are selected, when applicable
 - for text, both short and long character strings
 - for text, the whole character set

Test cases - 1

document		text	direction	Match case
This beautiful text	1	bea	down	yes
This beautiful text	2	beatles	down	yes
This 1 beautiful text	3	1bea	up	yes
This 1Beautiful text	4	1bea	up	yes
This &%1 BE autiful text	5	%1beau	down	no
This &%2beautiful text	6	%1beau	down	no
This BE utiful text	7	b	up	no
This BE utiful text	8	beauti	up	no
This BEAUTIFUL text	9	BEA	down	yes

Test cases - 2

document	text	direction	Match case
This BEAUTIFUL text 10	BEAT	down	yes
THIS beautiFUL text 11	THIS	up	yes
THIS beatiful text 12	T2S	up	yes
This Beautiful Text 13	HIS	down	no
this %#& beautiful text 14	S	down	no
this %#& beautiful text 15	HIS%#&	up	no
This %#&beautiful text 16	#& BE	up	no
This Beautiful Text 17	Text	down	yes
This Beautiful Text 18	Text	down	yes

Test cases - 3

document		text	direction	Match case
THIS is beautiful text	19	IS is	up	yes
This is beautiful text	20	IS is	up	yes
This text 1-99	21	ExT 1	down	no
This text 1 and text 2	22	eXt 1	down	no
This was beautiful text	23	His Was Beauti	down	no
(This) (Was) (123text)	24	aS()	up	no
123 one-two-three	25	123	down	yes
One-two-three 1-2-3	26	12-3	down	yes
This &007# mess	27	&	up	yes

Test cases - 4

document	text	direction	Match case
This Bloody Mess 28	#%	up	yes
(This) (was1) (was[2]) 29	2]	down	no
0987654321!'#%&/*/// 30	7654321#	down	no
1!2'3#4\$5%6&7/8(9)0=oops	#4\$5%6&7/8(9)	up	no
This %#&beautiful text 32	22	up	no
This is beautiful texT 33		down	no
1 or two 34		down	yes
1 or two 35		up	yes
0K1+(8Those 36		up	yes
1 & 2 37		down	no

Test cases - 5

document	text	direction	Match case
38		down	no
This %##&beautiful text	39	up	no
40		up	no

Example

- *print (file, copies, font, pagination)*
- **Input parameters:**
 - *name of the file* (must be provided)
 - *-cn*, where n is the number of copies ($1 \leq n \leq 100$);
 - default: $n = 1$
 - *-fkm*, where k indicates a font ($1 \leq k \leq 9$) and m indicates a mode (N for normal or B for bold);
 - defaults: $k = 1, m = N$
 - *-np*: no pagination (default: pagination shall be done)

Example... *Equivalence classes*

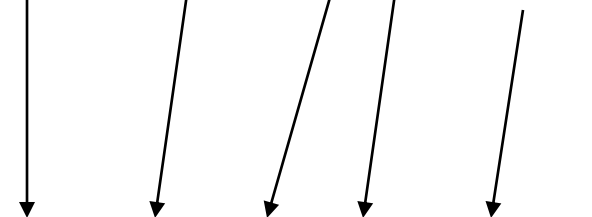
- *Originating from **file name**:*
 1. Name of existing file given (*Valid*).
 2. No file name given (*NotValid*).
 3. Name of non-existing file given (*NV*).
 4. “Name” does not follow the syntactic rules (*NV*).
- *Originating from **copies (-cn)**:*
 5. $1 \leq n \leq 100$ (*V*).
 6. Default: no *n* given (*V*).
 7. $n = 0$ or $n > 100$ (*NV*).

Example... *Equivalence classes*

- Originating from **fonts (-fkm)**:
 8. $1 \leq k \leq 9$ (*V*).
 9. Default: no *k* given (*V*).
 10. $m = N$ or $m = B$ (*V*).
 11. Default: no *m* given (*V*).
 12. $k = 0$ or $k > 9$ (*NV*).
 13. *m* other than *N* or *B* (*NV*).
- Originating from **pagination (-np)**:
 14. *-np* given (*V*).
 15. *-np* not given (*V*).
 16. Something else than *-np* given (*NV*). (This class covers also the other syntactically invalid ***-options***.)

Example... Number of exhaustive combinatory test cases

print file [-cn] [-f k m] [-np]



$4 * 3 * 3 * 3 * 3 = 324$ test cases

This might be too many, so a method reducing the number of test cases is needed.

Optimizing Principle

- ***print file*** [-**cn**] [-**fk****m**] [**np**]
- *Optimizing principle:*
 - one test case for each *NV* equivalence class
 - each equivalence class covered by *at least one* test case
 - i. -c5 -np
 - ii. xxyy -c3 (no file xxyy in directory)
 - iii. #%\$file5.3
 - iv. myfile -c0 (file *myfile* is in directory)
 - v. myfile -f100N
 - vi. myfile -f2H
 - vii. myfile -c5 -f1 -hjk

Equivalence Classes \ Test Cases

Test Cases

Equivalence Classes

EC/TC	i	ii	iii	iv	v	vi	vii
1				+	+	+	+
2	-						
3		-					
4			-				
5	+	+					+
6			+		+	+	
7				-			
8						+	+
9	+	+	+	+			
10					+		
11	+	+	+	+			+
12					-		
13						-	
14	+						
15		+	+	+	+	+	
16							-

Extending Principle

- combinations over the *number* of parameters
 - name of existing file always given
 - a test case where all parameters are missing (0 present)
 - a test case for each individual parameter (1 present)
 - each parameter included in the set of pairs (2 present)
 - each parameter included in the set of triplets (3 present)
 - all parameters given (4 present)

Example

- ***print* file [-cn] [-fkm] [-np]**

viii.myfile	(none present)
ix. myfile -c1	(n present)
x. myfile -f9	(k present)
xi. myfile -fB	(m present)
xii. myfile -np	(-np present)
xiii.myfile -f1N	(k, m present)
xiv.myfile -c100 -np	(n, -np present)
xv. myfile -c50 -f5 -np	(n, k, -np present)
xvi.myfile -c1 -fB -np	(n, m, -np present)
xvii.myfile -c99 -f2N -np	(all present)

Equivalence Classes \ Test Cases

Test Cases

Equivalence Classes

EC/TC	viii	ix	x	xi	xii	xiii	xiv	xv	xvi	xvii
1	+	+	+	+	+	+	+	+	+	+
2										
3										
4										
5		+					+	+	+	+
6	+		+	+	+	+				
7										
8			+			+		+		+
9	+	+		+	+		+		+	
10				+		+			+	+
11	+	+	+		+		+	+		
12										
13										
14					+		+	+	+	+
15	+	+	+	+		+				
16										

References

[1] Myers, *The Art of Software Testing*, 1978

1. Start with pure black-box testing

- You design test cases based only on:
- Requirements
- Specifications
- Input/output behavior
- At this stage, you **do not look at the code**. The goal is to simulate real user or system-level validation.

2. Run the test suite and collect execution traces

- While executing the black-box tests, you instrument the code (or run it with a coverage tool). This does **not influence test design**, only measurement.
- Typical tools collect:
 - Statement coverage
 - Branch coverage
 - Path coverage (if feasible)

3. Perform a *post* white-box analysis (not test design)

- Now you examine the code **after the tests have been run**, asking
 - “Which parts of the code were actually exercised by my black-box tests?”
- This is where white-box knowledge is used *analytically*, not constructively.

4. Compute coverage metrics as quality indicators

- **Statement Coverage**

- $SC = \frac{\textit{Executed Statements}}{\textit{Total Statements}}$

- **Branch Coverage**

- $BC = \frac{\textit{Executed Branches}}{\textit{Total Branches}}$

- These give you **quantitative indicators** of how thoroughly your black-box tests exercised the implementation.

5. Interpret the results (this is the key insight)

- **High coverage (e.g., >80–90%)**
 - Your black-box tests *incidentally* exercised most of the code
 - Suggests good alignment between spec and implementation
- **Low coverage**
 - Some code paths are not triggered by any spec-based test
 - Indicates:
 - Missing test scenarios
 - Possibly undocumented behavior
 - Dead or unreachable code

6. Feedback loop (optional but powerful)

- Now you have two options:
 - **Strict interpretation (purist black-box evaluation)**
 - Use coverage only as a *quality metric*, not to change tests.
 - **Pragmatic approach (industry standard)**
 - Add new tests to cover:
 - Uncovered branches
 - Edge conditions revealed by code structure
- This turns your process into a **coverage-guided test refinement loop.**

Why this works conceptually

- Black-box testing ensures **correctness vs. requirements**
- Post-white-box analysis ensures **thoroughness vs. implementation**
- Together, they reduce two different risks:
- Missing requirements → black-box handles this
- Missing code paths → coverage analysis exposes this