SOFTWARE TESTING

CO3015 / CO5252

CH4. BLACK-BOX TESTING

Content

Black-box testing

https://www.guru99.com/black-box-testing.html

- Black Box Testing:
  - *the **functionalities** of software applications are **tested***
  - ***without** having knowledge of internal code structure, implementation details and internal paths.*
- Black Box Testing:
  - *mainly **focuses** on **input and output** of software*
  - *entirely based on software **requirements and specifications**.*
- It is also known as Behavioral Testing.

A Back-box testing process

https://www.guru99.com/black-box-testing.html

- Examine the requirements and specifications of the system.
- Choose valid inputs (positive test scenario) to check whether the system processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the the system is able to detect them.
- Determines expected outputs for all those inputs.
- Constructs test cases with the selected inputs.
- Execute the test cases.
- Compares the actual outputs with the expected outputs.
- => Fix the defects if any, and re-test.

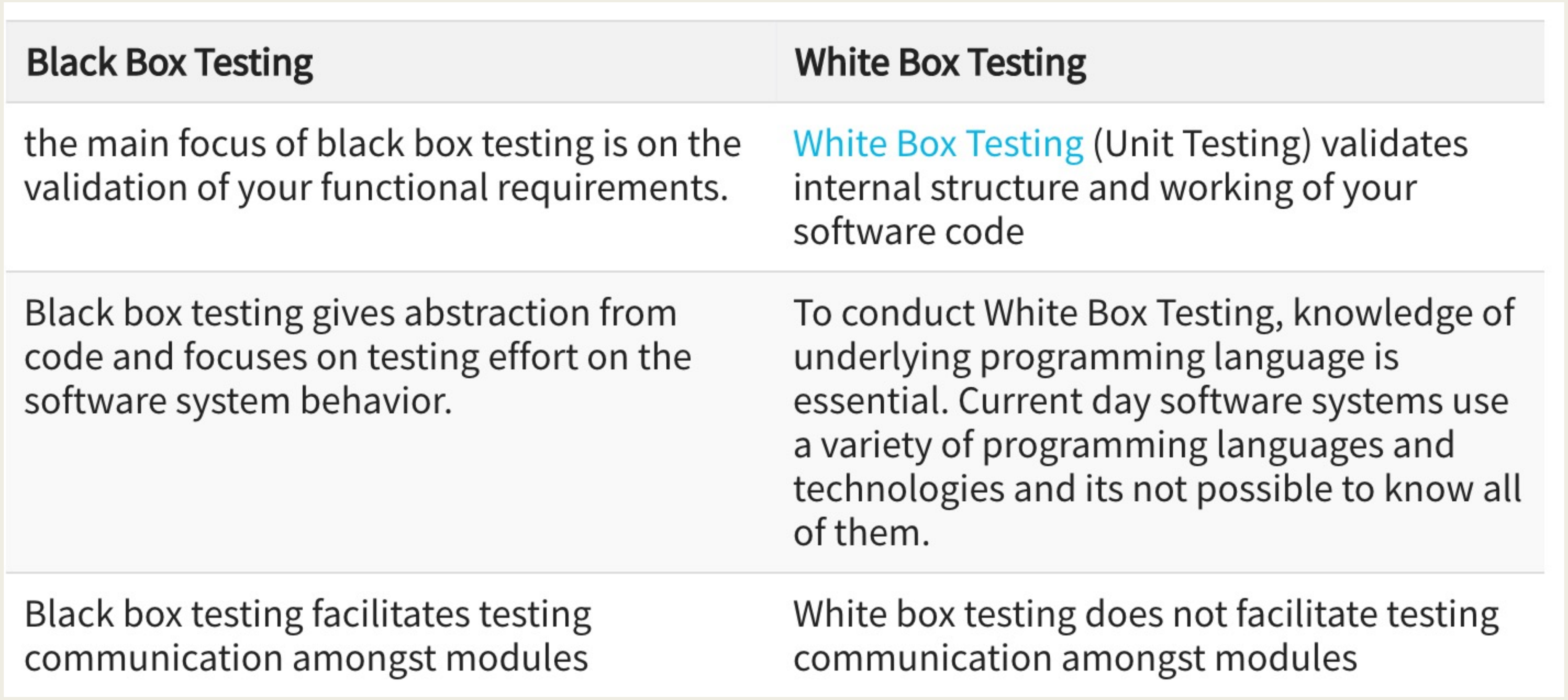
Types of Black Box Testing

https://www.guru99.com/black-box-testing.html

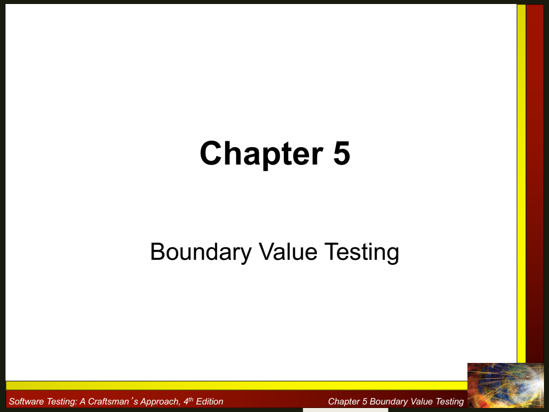
- Functional testing - This black box testing type is related to the functional requirements of a system; it is done by software testers.
- Non-functional testing - This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- Regression testing - Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.
- ...

Black-box vs. White-box testing

https://www.guru99.com/black-box-testing.html



Black Box Testing	White Box Testing
the main focus of black box testing is on the validation of your functional requirements.	White Box Testing (Unit Testing) validates internal structure and working of your software code
Black box testing gives abstraction from code and focuses on testing effort on the software system behavior.	To conduct White Box Testing, knowledge of underlying programming language is essential. Current day software systems use a variety of programming languages and technologies and its not possible to know all of them.
Black box testing facilitates testing communication amongst modules	White box testing does not facilitate testing communication amongst modules



Chapter 5

Boundary Value Testing

Software Testing: A Craftsman's Approach, 4th Edition

Chapter 5 Boundary Value Testing

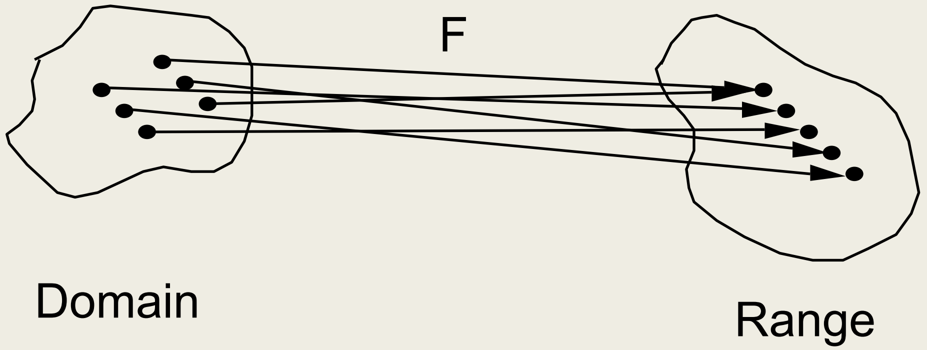
BOUNDARY VALUE ANALYSIS TECHNIQUE

Functional Testing

- The rationale for referring to specification-based testing as “functional testing” is likely due to the abstraction that any program can be viewed as a mapping from its Input Domain to its Output Range:

$$\text{Output} = F(\text{Input})$$

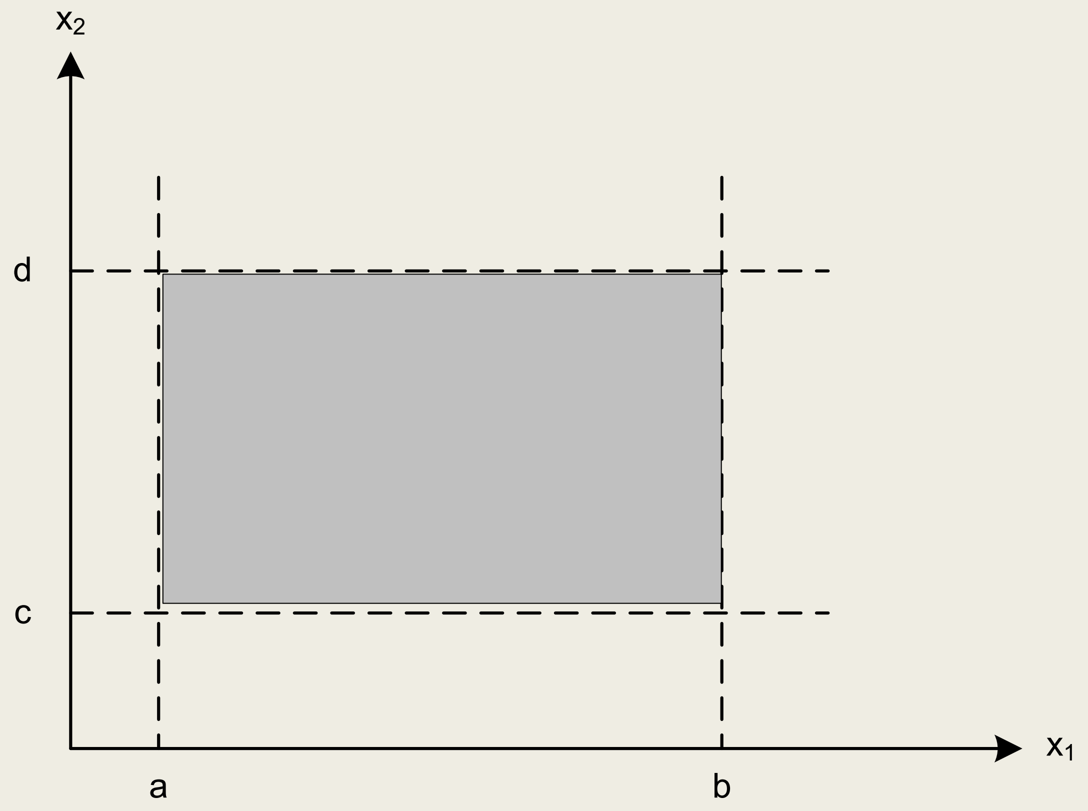
- Functional testing uses information about functional mappings to identify test cases.



Boundary Value Testing

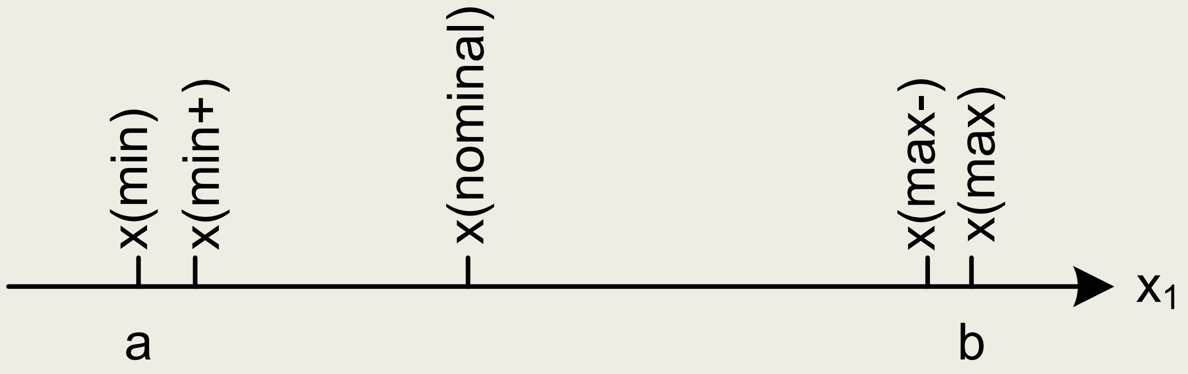
- Two considerations apply to boundary value testing
  - *are invalid values an issue?*
  - *can we make the “single fault assumption” of reliability theory?*
- Consequences...
  - *invalid values require the robust choice*
  - *multiplicity of faults requires worst case testing*
- Taken together, these yield four variations
  - *Normal boundary value testing*
  - *Robust boundary value testing*
  - *Worst case boundary value testing*
  - *Robust worst case boundary value testing*

Input Domain of F(x1, x2) where $a \le x1 \le b$ and $c \le x2 \le d$

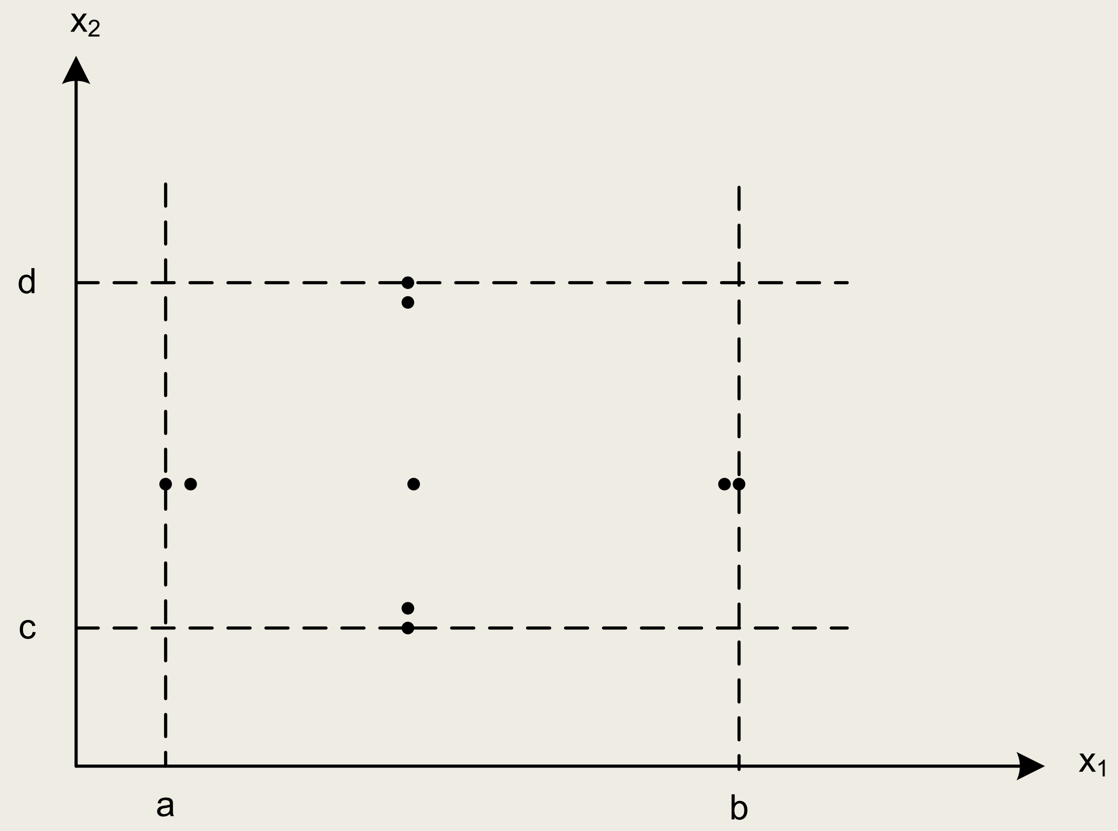


Input Boundary Value Testing

- Test values for variable x, where
  - *a ≤ x1 ≤ b, and*
  - *x(min), x(min+), ... x(max) are the names from the T tool.*



Normal Boundary Value Test Cases

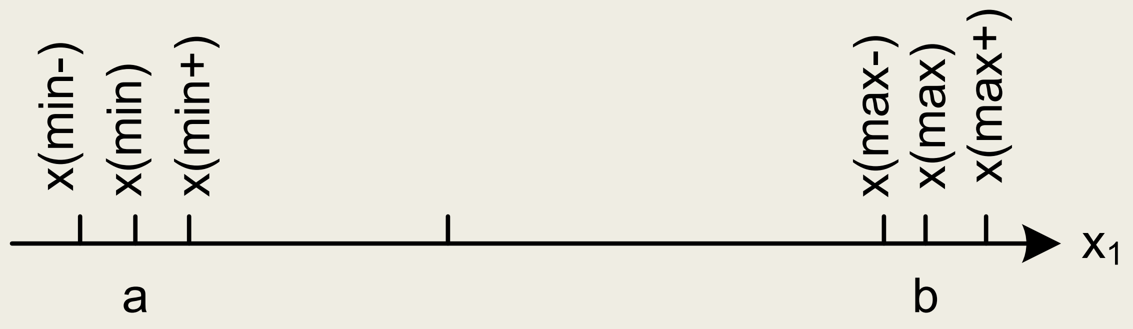


As in reliability theory, two variables rarely both assume their extreme values.

Method

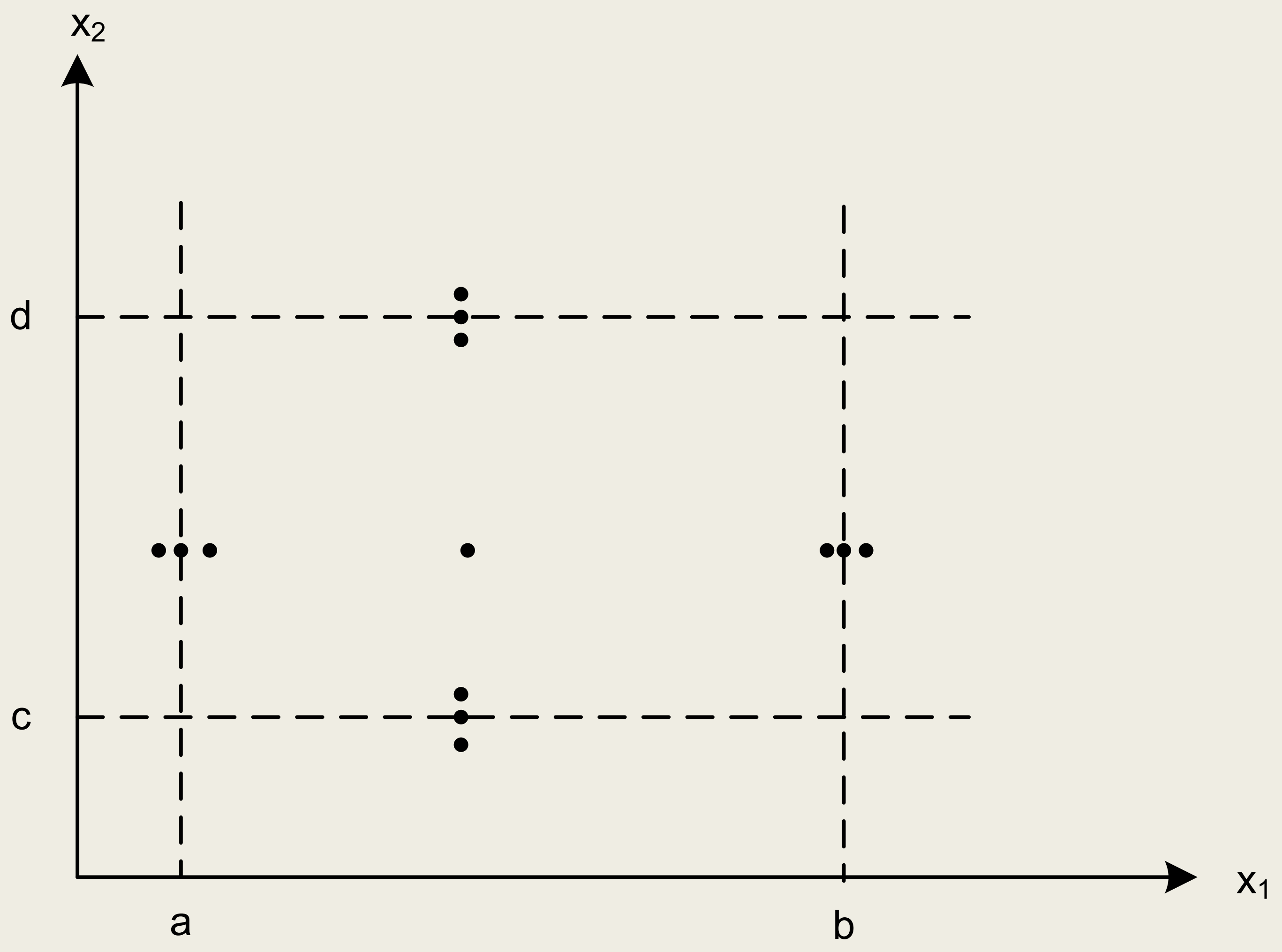
- Hold all variables at their nominal values.
- Let one variable assume its boundary values.
- Repeat this for each variable.
- This will (hopefully) reveal all faults that can be attributed to a single variable.

Robustness Testing

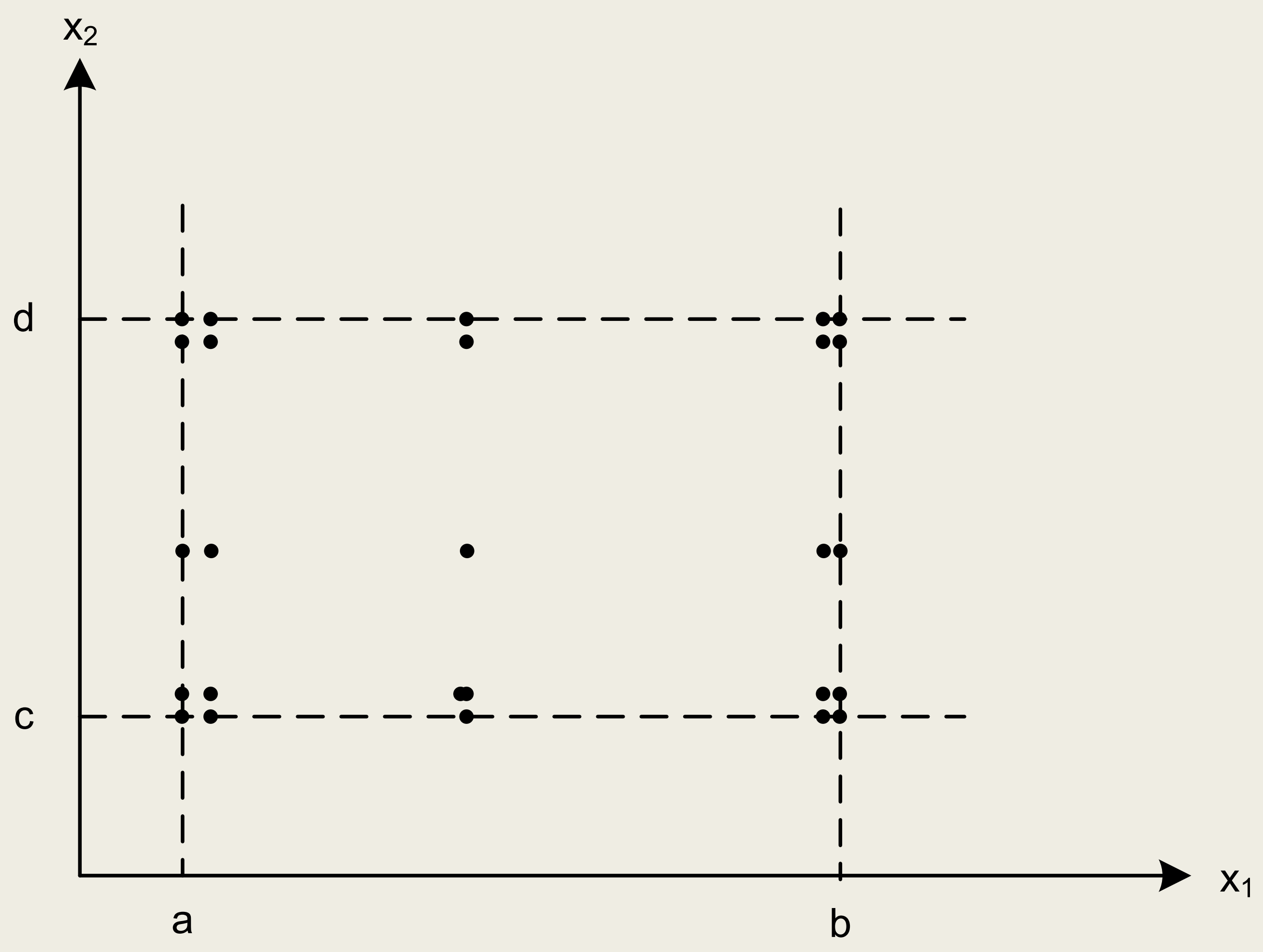


- Stress boundaries
- Possible advantages
  - *find hidden functionality*
  - *leads to exploratory testing*
- But...
  - *what are the expected outputs?*
  - *what if the programming language is strongly typed? (e.g., Ada)*

Robust Boundary Value Test Cases

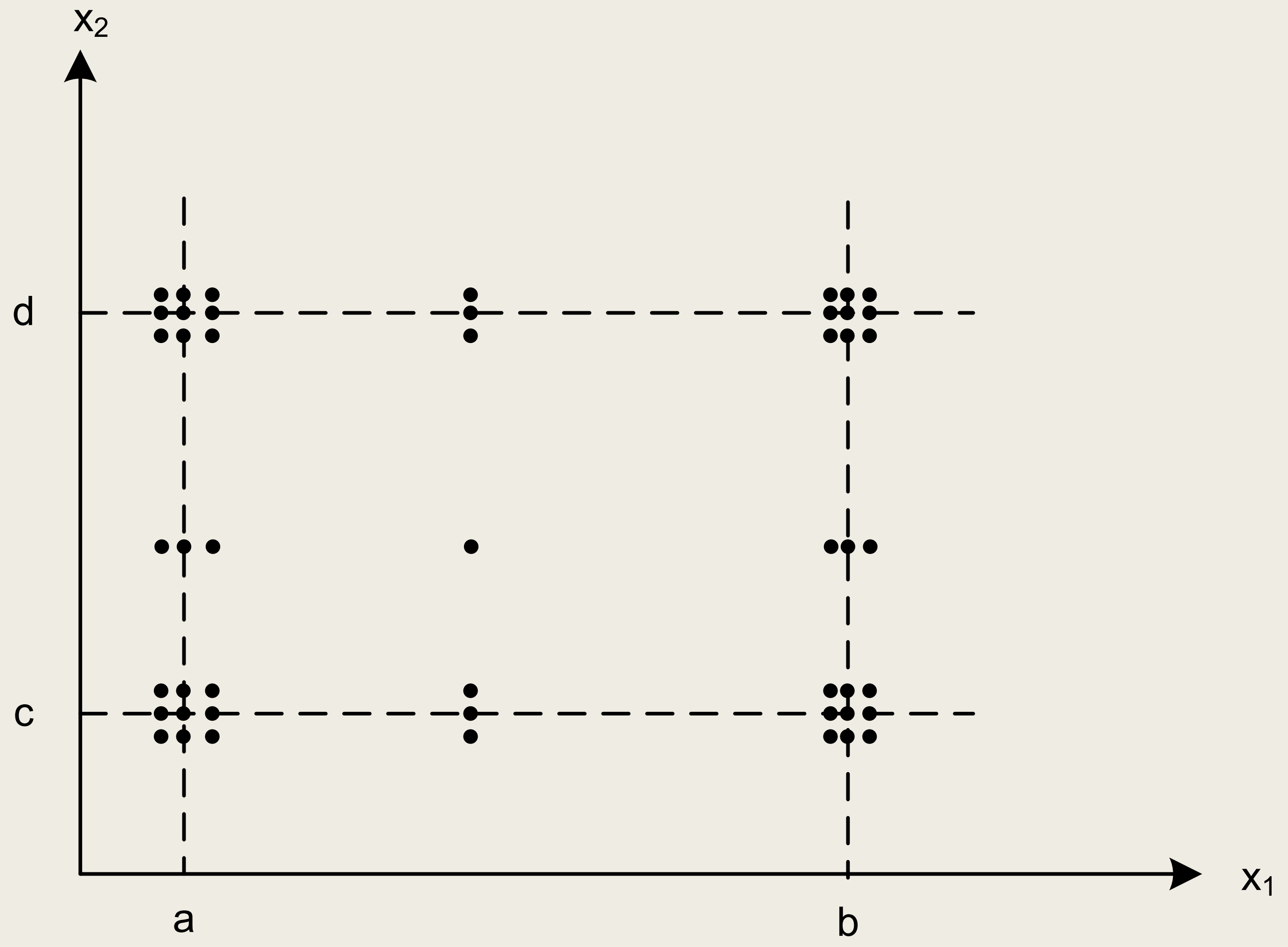


Normal Worst Case Boundary Value Test Cases



Responding to the single-fault assumption.

Robust Worst Case Boundary Value Test Cases



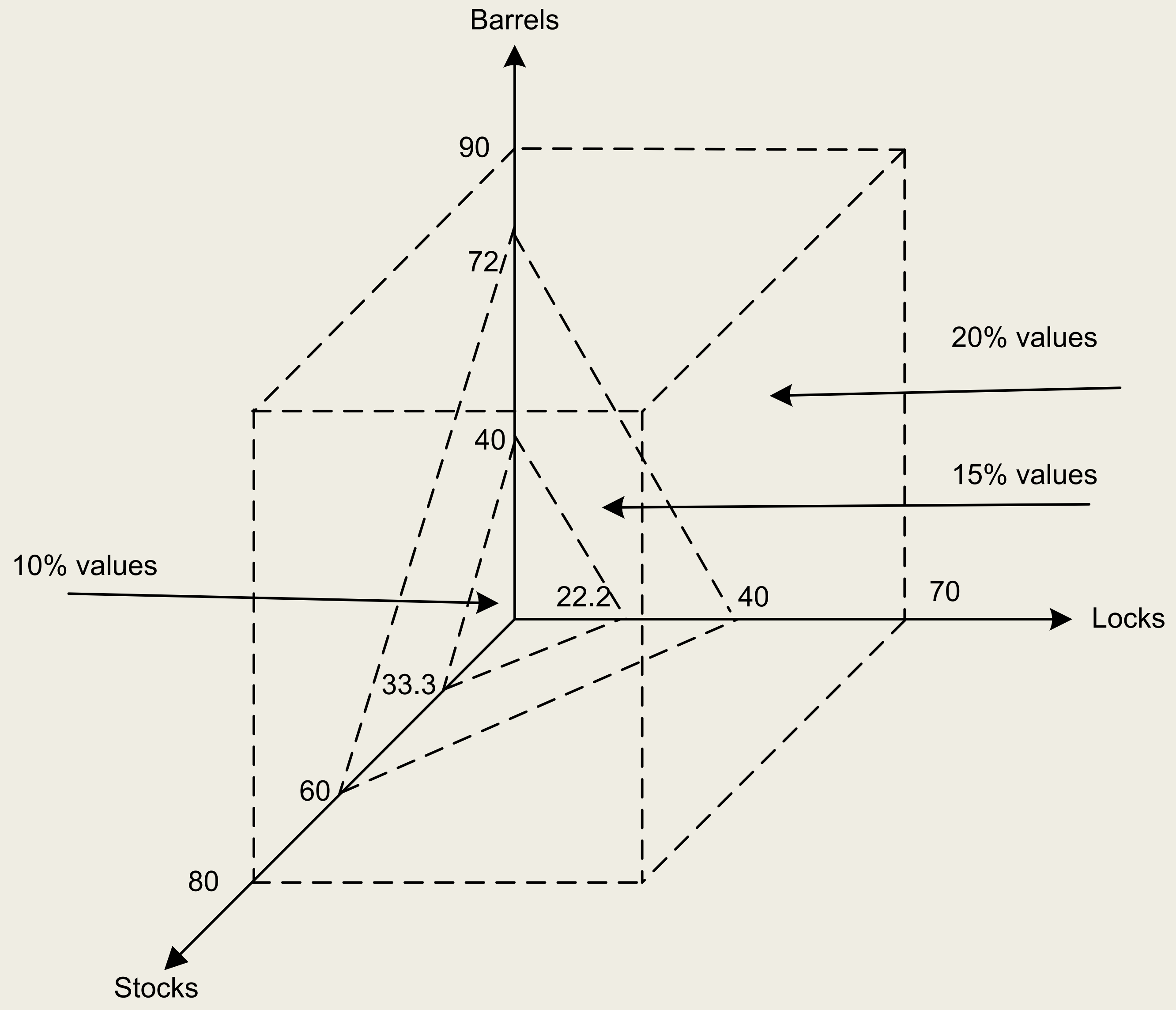
Special Value Testing

- Appropriate for
  - *complex mathematical calculations*
  - *worst case situations (similar to robustness)*
  - *problematic situations from past experience*
- Characterized by...
  - *"second guess" likely implementations*
  - *experience helps*
  - *frequently done by customer/user*
  - *defies measurement*
  - *highly intuitive*
  - *seldom repeatable*
  - *(and is often most effective)*

Output Range Coverage

- 1. Work "backwards" from expected outputs
  - *(assume that inputs cause outputs).*
- 2. Mirror image of equivalence partitioning
  - *(good cross check)*
- 3. Helps identify ambiguous causes of outputs.

Input Domain for Commission Problem



NextDate Function

- NEXTDATE is a function of three variables: month, day, and year, for years from 1812 to 2012. It returns the date of the next day.
  - *NEXTDATE(Dec, 31, 1991) returns Jan 1 1992*
  - *NEXTDATE(Feb, 21, 1991) returns Feb 22 1991*
  - *NEXTDATE(Feb, 28, 1991) returns Mar 1 1991*
  - *NEXTDATE(Feb, 28, 1992) returns Feb 29 1992*

- Leap Year: Years divisible by 4 except for century years not divisible by 400. Leap Years include 1992, 1996, 2000. 1900 was not be a leap year.

Boundary Value Test Cases for NextDate

- Observations
  - *not much reason for robustness testing*
  - *good reasons for worst case testing*
- Large number of test cases
  - *15 normal boundary value test cases*
  - *125 worst case boundary value test cases*
    - (see text for test case values)

Boundary Value Test Cases for NextDate

Case	***Month***	***Day***	***Year***	***Expected Output***
1	1	1	1812	1, 2, 1812
2	1	1	1813	1, 2, 1813
3	1	1	1912	1, 2, 1912
4	1	1	2011	1, 2, 2011
5	1	1	2012	1, 2, 2012
6	1	2	1812	1, 3, 1812
7	1	2	1813	1, 3, 1813
8	1	2	1912	1, 3, 1912
9	1	2	2011	1, 3, 2011
10	1	2	2012	1, 3, 2012
11	1	15	1812	1, 16, 1812
12	1	15	1813	1, 16, 1813
13	1	15	1912	1, 16, 1912
14	1	15	2011	1, 16, 2011
15	1	15	2012	1, 16, 2012

Special Value Test Cases

Case	***Month***	***Day***	***Year***	***Reason***
SV-1	2	28	2012	Feb. 28 in a leap year
SV-2	2	28	2013	Feb. 28 in a common year
SV-3	2	29	2012	Leap day in a leap year
SV-4	2	29	2000	Leap day in 2000
SV-5	2	28	1900	Feb. 28 in 1900
SV-6	12	31	2011	End of year
SV-7	10	31	2012	End of 31-day month
SV-8	11	30	2012	End of 30-day month
SV-9	12	31	2012	Last day of defined interval

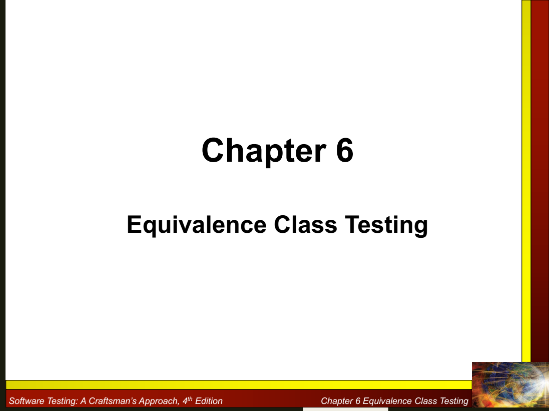
Output Range Test Cases

- In the case of the NextDate function, the range and domain are identical except for one day. Nothing interesting will be learned from output range test cases for this example.

- Part of the reason for this is that the NextDate function is a one-to-one mapping from its domain onto its range. When functions are not one-to-one, output range test cases are more useful.

Pros and Cons of Boundary Value Testing

- Advantages
  - *(Commercial tool support available)*
  - *Easy to do/automate*
  - *Appropriate for calculation-intensive applications with variables that represent physical quantities (e.g., have units, such as meters, degrees, kilograms)*
- Disadvantages
  - *Inevitable potential for both gaps and redundancies*
  - *The gaps and redundancies can never be identified (specification-based)*
  - *(Does not scale up well (?))*
  - *Tools only generate inputs, user must generate expected outputs.*



Chapter 6

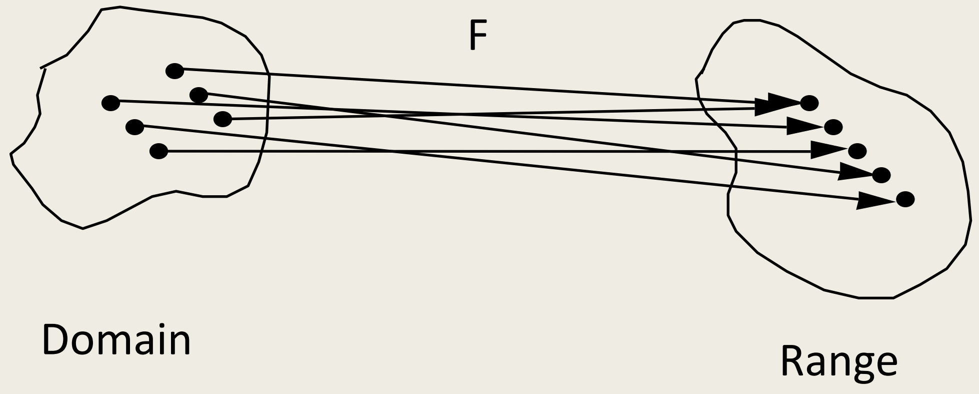
Equivalence Class Testing

Software Testing: A Craftsman's Approach, 4th Edition

Chapter 6 Equivalence Class Testing

EQUIVALENCE CLASS PARTITIONING TECHNIQUE

Equivalence Class Testing



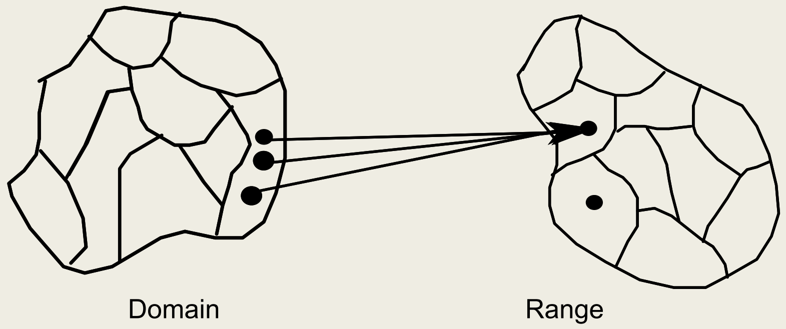
Equivalence class testing uses information about the functional mapping itself to identify test cases

Equivalence Relations

- Given a relation R defined on some set S, R is an equivalence relation if (and only if), for all, x, y, and z elements of S:
  - *R is reflexive, i.e., xRx*
  - *R is symmetric, i.e., if xRy, then yRx*
  - *R is transitive, i.e., if xRy and yRz, then xRz*
- An equivalence relation, R, induces a partition on the set S, where a partition is a set of subsets of S such that:
  - *The intersection of any two subsets is empty, and*
  - *The union of all the subsets is the original set S*
- Note that the intersection property assures no redundancy, and the union property assures no gaps.
  - *vs. Boundary analysis techniques?*

Equivalence Partitioning

- Define a relation R on the input domain D as:
  - *∀x, y ϵ D, xRy iff F(x) = F(y), where F is the program function.*
- R is the "**treated the same**" relation
- R is an equivalence relation ?



Equivalence Partitioning (continued)

- Works best when F is a many-to-one function
- Test cases are formed by selecting one value from each equivalence class.
- Identifying the classes may be hard

-

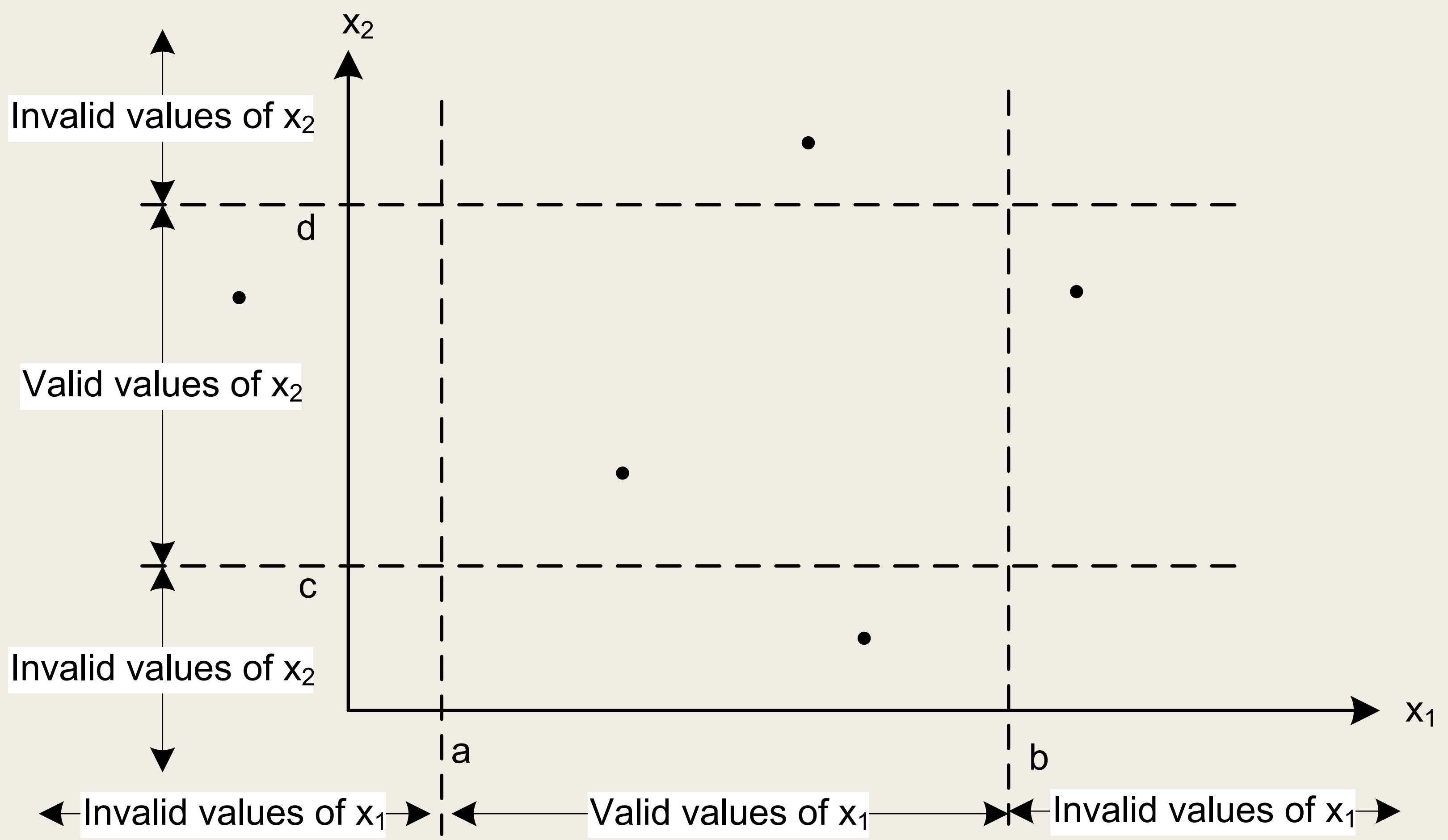
Forms of Equivalence Class Testing

- “Traditional” - focus on invalid inputs
- Normal: classes of valid values of inputs
- Robust: classes of valid and invalid values of inputs
- Weak: (single fault assumption) one from each class
- Strong: (multiple fault assumption) one from each class in Cartesian Product

Continuing Example

  - *(only 2-dimensions for drawing purposes)*
- F(x1, x2) has these classes...
  - *valid values of x1: $a \le x1 \le b$*
  - *invalid values of x1: $x1 < a$, $b < x1$*
  - *valid values of x2: $c \le x2 \le d$*
  - *invalid values of x2: $x2 < c$, $d < x2$*
- Process
  - *test F for valid values of all variables,*
  - *then test one invalid variable at a time*
  - *(note this makes the single fault assumption)*

Example

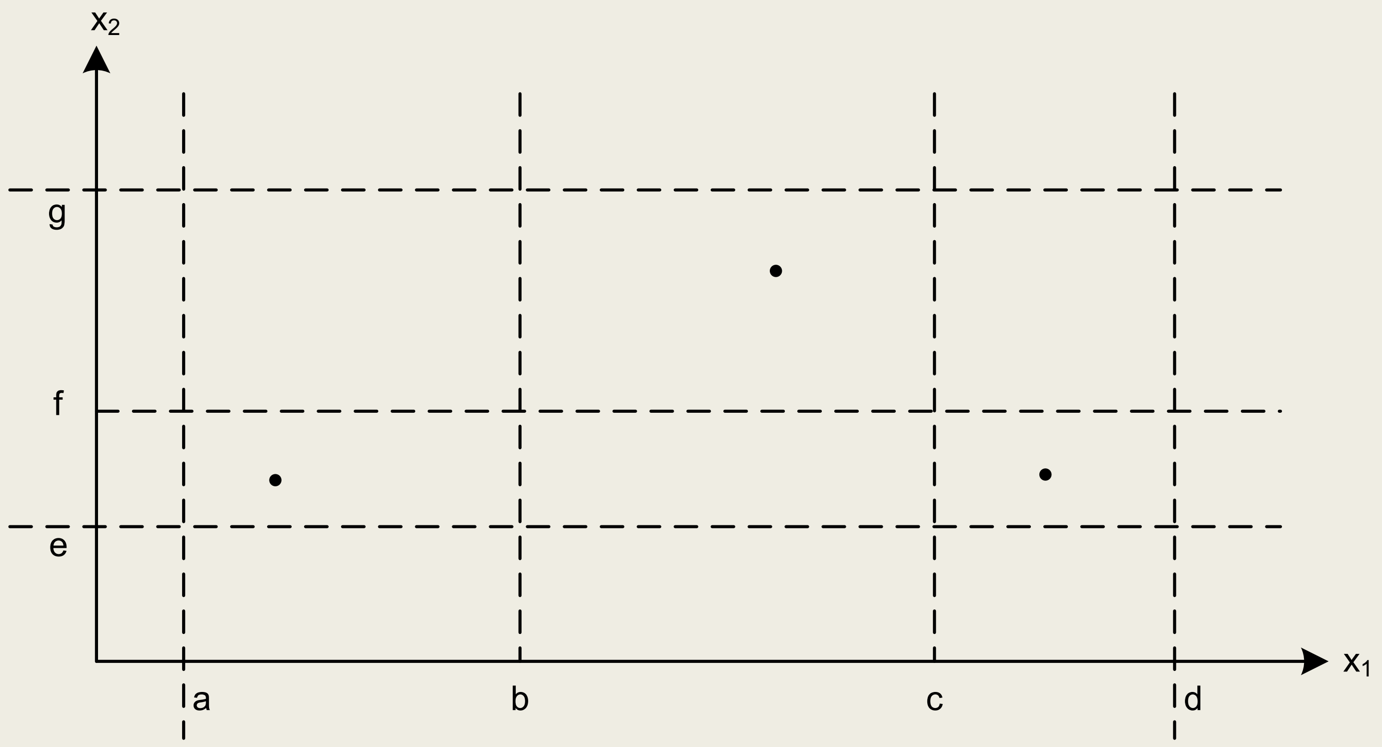


x_2
Invalid values of x_2
d
Valid values of x_2
c
Invalid values of x_2
x_1
a
b
Invalid values of x_1
Valid values of x_1
Invalid values of x_1

Weak Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- **Test cases have all valid values.**
- Detects faults due to calculations with valid values of a single variable.
- OK for regression testing.
- Need an expanded set of valid classes
  - *valid classes: {a <= x1 < b}, {b <= x1 < c}, {c <= x1 <= d}, {e <= x2 < f}, {f <= x2 <= g}*
  - *invalid classes: {x1 < a}, {x1 > d}, {x2 < e}, {x2 > g}*

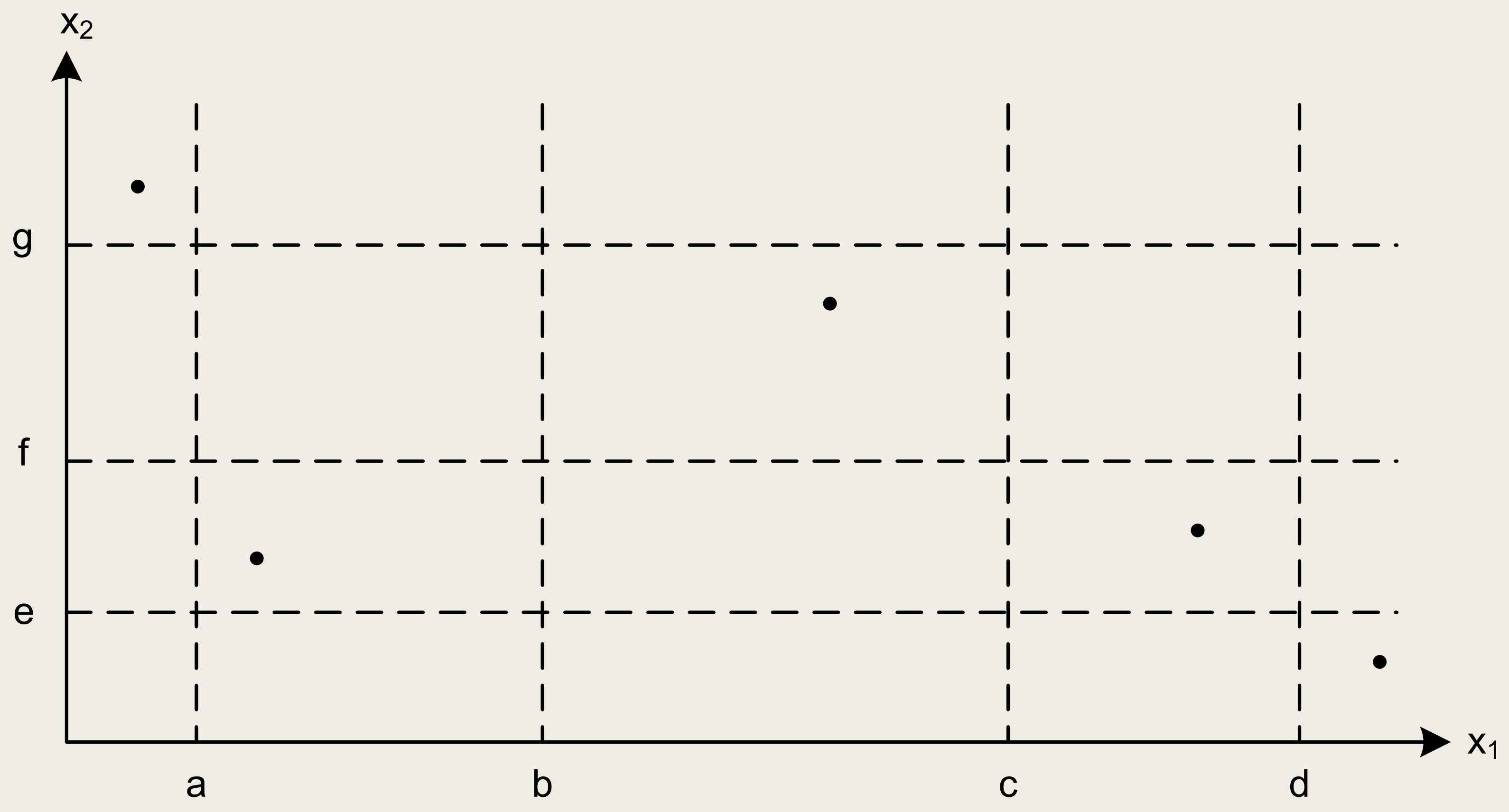
Weak Normal Equivalence Class Test Cases



Weak Robust Equivalence Class Testing

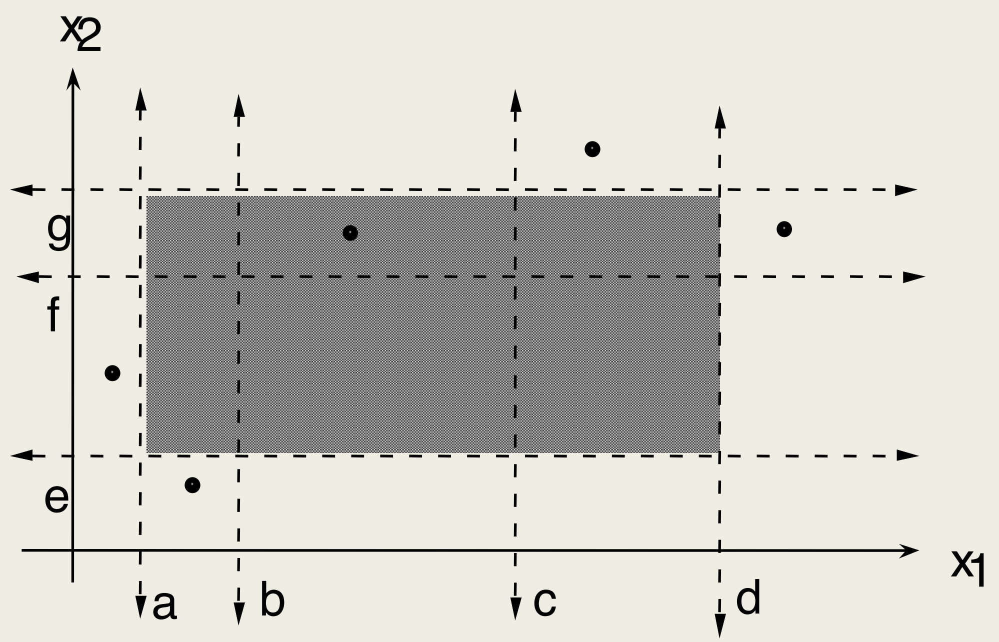
- Identify equivalence classes of valid and invalid values.
- **Test cases have all valid values except one invalid value.**
- Detects faults due to calculations with valid values of a single variable.
- Detects faults due to invalid values of a single variable.
- OK for regression testing.

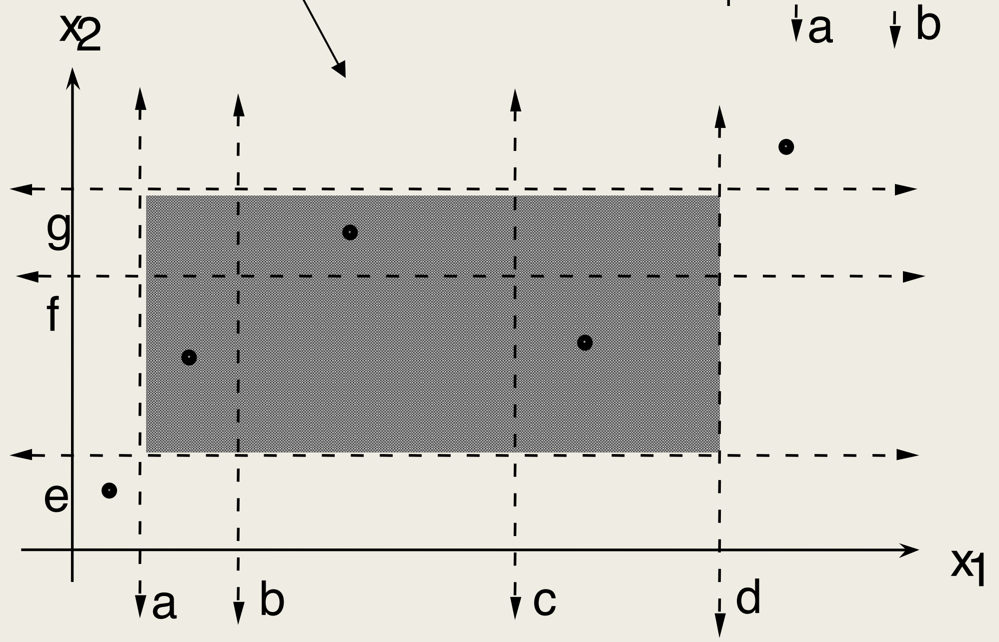
Weak Robust Equivalence Class Test Cases



x1 has all valid and invalid (one for each class)
x2 has all valid and invalid (one for each class)

Is this preferable to this? Why?



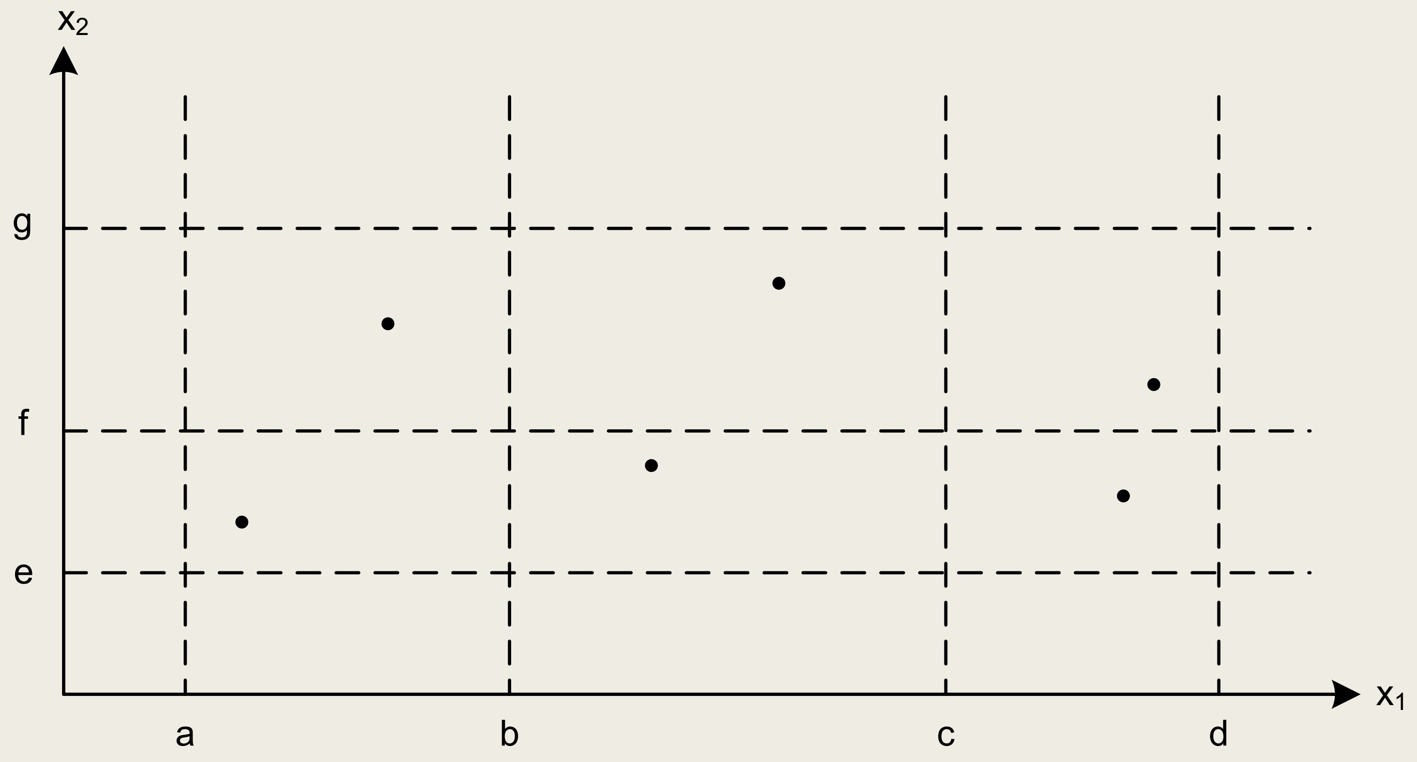


Errors for (x1>d, x2>g) or (x1<a, x2<e) could be due to the interaction of two variables ☹

Strong Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- **Test cases from Cartesian Product of valid values.**
- Detects faults due to interactions with valid values of any number of variables.
- OK for regression testing, better for progression testing.

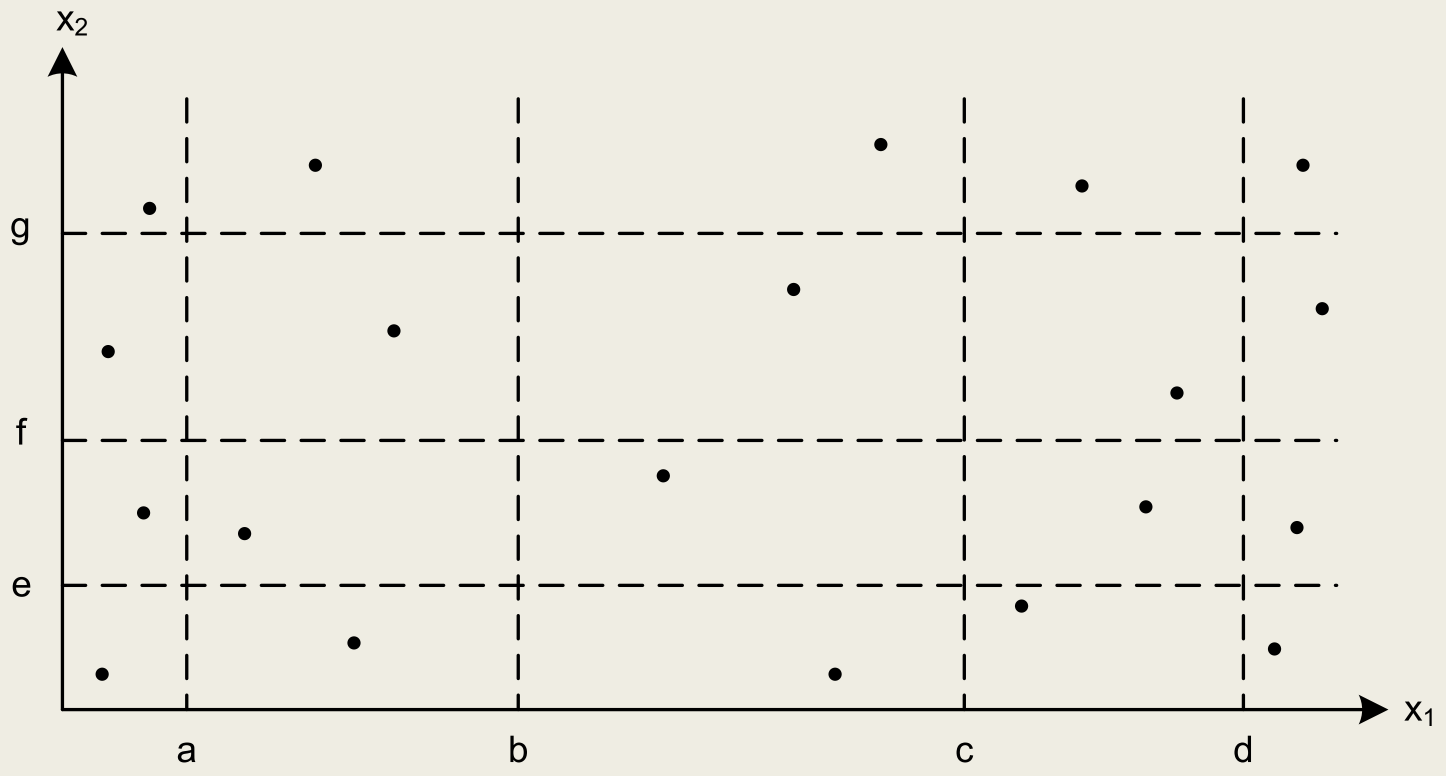
Strong Normal Equivalence Class Test Cases



Strong Robust Equivalence Class Testing

- Identify equivalence classes of valid and invalid values.
- **Test cases from Cartesian Product of all classes.**
- Detects faults due to interactions with any values of any number of variables.
- OK for regression testing, better for progression testing.
  - *(Most rigorous form of Equivalence Class testing, BUT,*
  - *Jorgensen’s First Law of Software Engineering applies.)*
- Jorgensen’s First Law of Software Engineering:
  - *The product of two big numbers is a really big number.*
  - *(More elegant: scaling up can be problematic)*

Strong Robust Equivalence Class Test Cases



Selecting an Equivalence Relation

- There is no such thing as THE equivalence relation.
- If x and y are days, some possibilities for Nextdate are:
  - *x R y iff x and y are mapped onto the same year*
  - *x R y iff x and y are mapped onto the same month*
  - *x R y iff x and y are mapped onto the same date*
  - *x R y iff x(day) and y(day) are "treated the same"*
  - *x R y iff x(month) and y(month) are "treated the same"*
  - *x R y iff x(year) and y(year) are "treated the same"*
- Best practice is to select an equivalence relation that reflects the behavior being tested.

NextDate Equivalence Classes

- Month:
  - *M1 = { month : month has 30 days}*
  - *M2 = { month : month has 31 days}*
  - *M3 = { month : month is February}*

- Day
  - *D1 = {day : 1 <= day <= 28}*
  - *D2 = {day : day = 29 }*
  - *D3 = {day : day = 30 }*
  - *D4 = {day : day = 31 }*

- Year (are these disjoint?)
  - *Y1 = {year : year = 2000}*
  - *Y2 = {year : 1812 <= year <= 2012 AND (year ≠ 0 Mod 100) and (year = 0 Mod 4)*
  - *Y3 = {year : (1812 <= year <= 2012 AND (year ≠ 0 Mod 4)*

Not Quite Right

- A better set of equivalence classes for year is
  - *Y1 = {century years divisible by 400} i.e., century leap years*
  - *Y2 = {century years not divisible by 400} i.e., century common years*
  - *Y3 = {non-century years divisible by 4} i.e., ordinary leap years*
  - *Y4 = {non-century years not divisible by 4} i.e., ordinary common years*
- All years must be in range: 1812 <= year <= 2012
- Note that these equivalence classes are disjoint.

Weak Normal Equivalence Class Test Cases

- Select test cases so that one element from each input domain equivalence class is used as a test input value.
  - *The number of test cases is the same as the highest number of equivalence classes for a variable.*
  - *Notice that all forms of equivalence class testing presume that the variables in the input domain are independent; logical dependencies are not recognized.*

Test Case	Input Domain Equiv. Classes	Input Values	Expected Outputs
WN-1	M1, D1, Y1	April 1 2000	April 2 2000
WN-2	M2, D2, Y2	Jan. 29 1900	Jan. 30 1900
WN-3	M3, D3, Y3	Feb. 30 1812	impossible
WN-4	M1, D4, Y4	April 31 1901	impossible

Strong Normal Equivalence Class Test Cases

- With 4 day classes, 3 month classes, and 4 year classes, the Cartesian Product will have 48 equivalence class test cases. (Jorgensen's First Law of Software Engineering strikes again!)

- Note some judgment is required. Would it be better to have 5 day classes, 4 month classes and only 2 year classes? (40 test cases)

- Questions such as this can be resolved by considering Risk.

Revised NextDate Domain Equivalence Classes

- Month:
  - *M1 = { month : month has 30 days}*
  - *M2 = { month : month has 31 days except December}*
  - *M3 = { month : month is February}*
  - *M4 = {month : month is December}*
- Day
  - *D1 = {day : 1 <= day <= 27}*
  - *D2 = {day : day = 28 }*
  - *D3 = {day : day = 29 }*
  - *D4 = {day : day = 30 }*
  - *D5 = {day : day = 31 }*
- Year (are these disjoint?)
  - *Y1 = {year : year is a leap year}*
  - *Y2 = {year : year is a common year}*

The Cartesian Product of these contains 40 elements.

When to Use Equivalence Class Testing

- Variables represent logical (rather than physical) quantities.
  - *“treated as the same”*
- Variables “support” useful equivalence classes.

=> Equivalence class testing is appropriate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.

Assumption Matrix

	Valid Values	Valid and Invalid Values
Single fault	Boundary Value Weak Normal Equiv. Class	Robust Boundary Value Weak Robust Equiv. Class
Multiple fault	Worst Case Boundary Value Strong Normal Equiv. Class	Robust Worst Case Boundary Value Strong Robust Equiv. Class

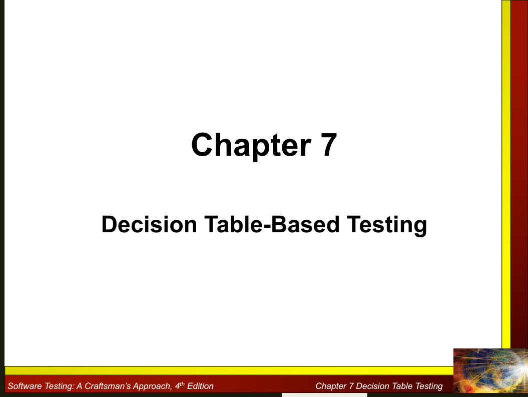
DOMAIN ANALYSIS TECHNIQUE

Domain analysis technique

https://www.guru99.com/domain-testing.html

Domain analysis technique - Summary

- Domain analysis facilitates the testing of multiple variables simultaneously.
- It builds on and generalizes equivalence class and boundary value testing to n simultaneous dimensions.
- In using the 1x1 domain analysis technique
  - *for each relational condition (≥, >, ≤, or <) we choose one on point and one off point*
  - *for each strict equality condition (=) we choose one on point and two off points:*
    - one slightly less than the conditional value and one slightly greater than the value.


Chapter 7

Decision Table-Based Testing

Software Testing: A Craftsman's Approach, 4th Edition

Chapter 7 Decision Table Testing

DECISION TABLE TECHNIQUE

Decision Table Based Testing

- Equivalent to forming a decision table in which:
  - *inputs are conditions*
  - *outputs are actions*
- **Test every (possible) rule in the decision table.**
- Recommended for logically complex situations.

Content of a Decision Table

- Conditions
  - *binary in a Limited Entry Decision Table (LEDT)*
  - *finite set in an Extended Entry Decision Table (EEDT)*
  - *condition stub*
  - *condition entries*
- Actions
  - *also binary, either do or skip*
  - *the "impossible" action*
- Rules
  - *a rule consists of condition entries and action entries*
  - *a complete, non-redundant LEDT with n conditions has 2n rules*
  - *logically impossible combinations of conditions are "impossible rules", denoted by an entry in the impossible action*

Example

Stub	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
c1	T	T	T	T	F	F	F	F
c2	T	T	F	F	T	T	F	F
c3	T	F	T	F	T	F	T	F
a1	X	X			X			
a2	X					X		
a3		X						
a4			X	X	X		X	X

Condition c3 has no effect on the actions of Rules 3 and 4. Similarly for Rules 7 and 8. They can be algebraically combined.

Example (continued)

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4*	*Rule 5*	*Rule 6*	*Rules 7, 8*
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	T
a1	X	X		X		—
a2	X				X	
a3		X				
a4			X	X		X

The condition entries in rules 3 and 4, and rules 7 and 8 have the same actions. The "—" means ...

- "Don't Care" (as in circuit analysis),
- Irrelevant, or
- not applicable, n/a

Example (continued)

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4, 7, 8*	*Rule 5*	*Rule 6*
c1	T	T	—	F	F
c2	T	T	F	T	T
c3	T	F	—	T	F
a1	X	X		X	
a2	X				X
a3		X			
a4			X	X	

One more algebraic simplification

Example (continued)

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4, 7, 8*	*Rule 5*	*Rule 6*
c1	T	T	—	F	F
c2	T	T	F	T	T
c3	T	F	—	T	F
a1	X	X		X	
a2	X				X
a3		X			
a4			X	X	
count	1	1	4	1	1

Rule counting

- a rule with no don't care entries counts as 1
- each don't care entry in a rule doubles the rule count
- for a table with n limited entry conditions, the sum of the rule counts should be 2n.

Problematic Decision Tables

- For LEDTs, simple rule counting helps identify decision tables that are ...
  - *incomplete (rule count < 2n) ,*
  - *redundant (rule count > 2n) , or*
  - *inconsistent*
    - (rule count > 2n) AND
    - at least two rules have identical condition entries but different action entries.
- Redundancy and inconsistency are more likely with algebraically simplified tables that have been “maintained”.

A Redundant DT

conditions	*1 – 4*	*5*	*6*	*7*	*8*	*9*
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	—
a3	X	—	X	X	X	X

- Rule 9 is redundant with Rules 1 – 4 (technically, with what was rule 4)
- But the action entries are identical (No harm, no foul?)

An Inconsistent DT

conditions	*1 – 4*	*5*	*6*	*7*	*8*	*9*
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	—
a2	—	X	X	X	—	X
a3	X	—	X	X	X	—

- Rule 9 is inconsistent with Rules 1 – 4 (technically, with what was rule 4)
  - *condition portion is identical, BUT*
  - *action portion is different*
- What happens when Rule 4 is executed? Rule 9?

Last Day of Month Decision Table

conditions	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
c1. month in M1?	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
c2. month in M2?	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
c3. month in M3?	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
c4. leap year?	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
a1. last day = 30								x								
a2. last day = 31											x	x				
a3. last day = 28														x		
a4. last day = 29													x			
a5. impossible	x	x	x	x	x	x			x	x					x	x

- Rule pairs 1 and 2, 3 and 4, 5 and 6, 9 and 10 don’t need c4, so they could be combined, BUT
- Impossible because c1, c2, and c3 are mutually exclusive.

Extended Entry Decision Tables

- When conditions are mutually exclusive, exactly one must be true.
- Extended entry decision tables typically (but not necessarily) have mutually exclusive conditions.
- The “extended” part is because a condition stub is an incomplete statement that is completed by the condition entry.
- (See the revised Last Day of Month EEDT)

Revised Last Day of Month DT

c1. month in	M1	M1	—	—	—	—
c2. month in	—	—	M2	M2	—	—
c3. month in	—	—	—	—	M3	M3
c4. leap year?	T	F	T	F	T	F
a1. last day = 30	x	x				
a2. last day = 31			x	x		
a3. last day = 28						x
a4. last day = 29					x	

- When conditions are mutually exclusive, exactly one must be true.
- This can be further simplified.

The “Emphatic False”

c1. month in	M1	—	—	—
c2. month in	—	M2	—	—
c3. month in	—	—	M3	M3
c4. leap year?	—	—	T	F
a1. last day = 30	x			
a2. last day = 31		x		
a3. last day = 28				x
a4. last day = 29			x	

- Maybe “—” should be replaced by “must be False”
- One writer suggested “F!” (before “!” meant “NOT”)
- It is a Don’t Care in c4.
- Technically, this is a Mixed Entry Decision Table, because it has both extended and limited entry conditions.

Triangle Program Decision Table

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Why are some rules impossible?

Expanding c1...

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	x	x	x								
a2: Scalene											x
a3: Isosceles						x			x	x	
a4: Equilateral				x							
a5: Impossible					x	x		x			

- Is this a complete decision table?
  How many test cases does this imply?

Rule Counting

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
Rule count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a triangle	x	x	x								
a2: Scalene											x
a3: Isosceles						x			x	x	
a4: Equilateral				x							
a5: Impossible					x	x		x			

Corresponding Test Cases

Case ID	*a*	*b*	*c*	*Expected Output*
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

NextDate Decision Table (first half)

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
c1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
c2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
c3: year in	—	—	—	—	—	—	—	—	—	—
a1: impossible					X					
a2: increment day	X	X	X			X	X	X	X	
a3: reset day				X						X
a4: increment month				X						X
a5: reset month										
a6: increment year										

NextDate Decision Table (first half reduced)

	1–3	*4*	*5*	*6–9*	*10*
c1: month in	M1	M1	M1	M2	M2
c2: day in	D1, D2, D3	D4	D5	D1, D2, D3, D4	D5
c3: year in	—	—	—	—	—
a1: impossible			X		
a2: increment day	X			X	
a3: reset day		X			X
a4: increment month		X			X
a5: reset month					
a6: increment year					

NextDate Decision Table (second half)

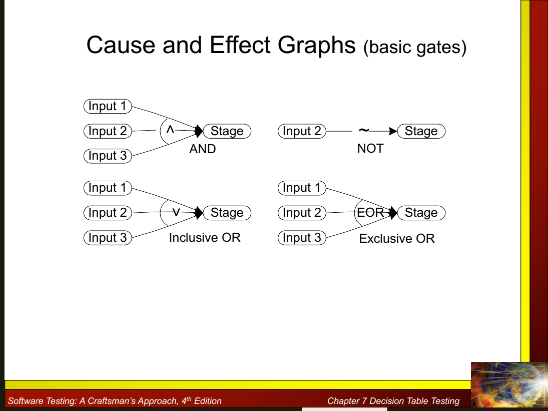
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
c1: month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2: day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: year in	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—	—
a1: impossible										X	X	X
a2: increment day	X	X	X	X		X	X					
a3: reset day					X			X	X			
a4: increment month								X	X			
a5: reset month					X							
a6: increment year					X							

NextDate Decision Table (second half reduced)

	11–14	*15*	*16*	*17*	*18*	*19*	20	*21, 22*
c1: month in	M3	M3	M4	M4	M4	M4	M4	M4
c2: day in	D1, D2, D3, D4	D5	D1	D2	D2	D3	D3	D4, D5
c3: year in	—	—	—	Y1	Y2	Y1	Y2	—
a1: impossible							X	X
a2: increment day	X			X				
a3: reset day		X	X		X	X		
a4: increment month					X	X		
a5: reset month			X					
a6: increment year			X					

NextDate Test Cases

Test Case	*Rule(s)*	*Month*	*Day*	*Year*	*Expected Output*
1	1– 3	4	15	2001	4/16/2001
2	4	4	30	2001	5/1/2001
3	5	4	31	2001	Invalid Input Date
4	6–9	1	15	2001	1/16/2001
5	10	1	31	2001	2/1/2001
6	11–14	12	15	2001	12/16/2001
7	15	12	31	2001	1/1/2002
8	16	2	15	2001	2/16/2001
9	17	2	28	2004	2/29/2004
10	18	2	28	2001	3/1/2001
11	19	2	29	2004	3/1/2004
12	20	2	29	2001	Invalid Input Date
13	21, 22	2	30	2001	Invalid Input Date


Cause and Effect Graphs (basic gates)

Software Testing: A Craftsman's Approach, 4th Edition

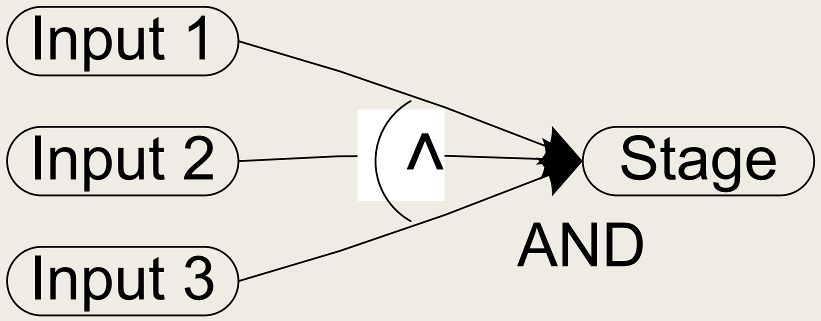
Chapter 7 Decision Table Testing

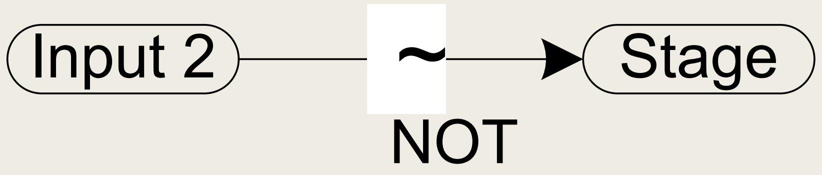
CAUSE-EFFECT GRAPH TECHNIQUE

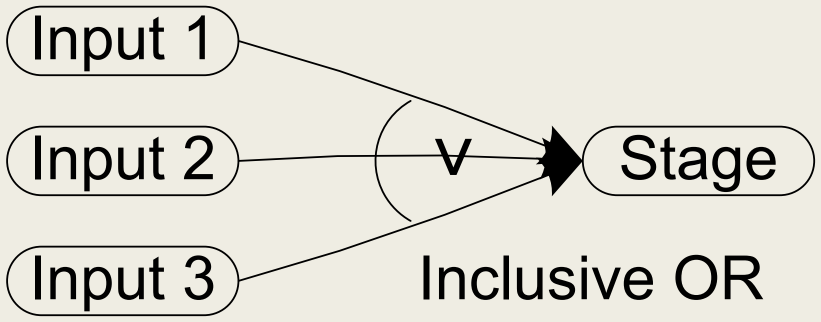
Cause and Effect Graph based Testing

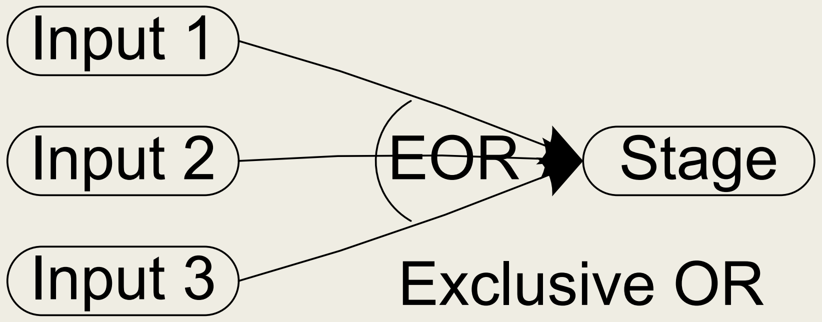
- Cause and Effect Graphing
  - *Done with a graphical technique that expressed AND-OR-NOT logic.*
  - *Causes and Effects were graphed like circuit components*
  - *Inputs to a circuit "caused" outputs (effects)*
- Equivalent to forming a decision table in which:
  - *inputs are conditions*
  - *outputs are actions*
- Test every (possible) rule in the decision table.
- Recommended for logically complex situations.

Cause and Effect Graphs (basic gates)

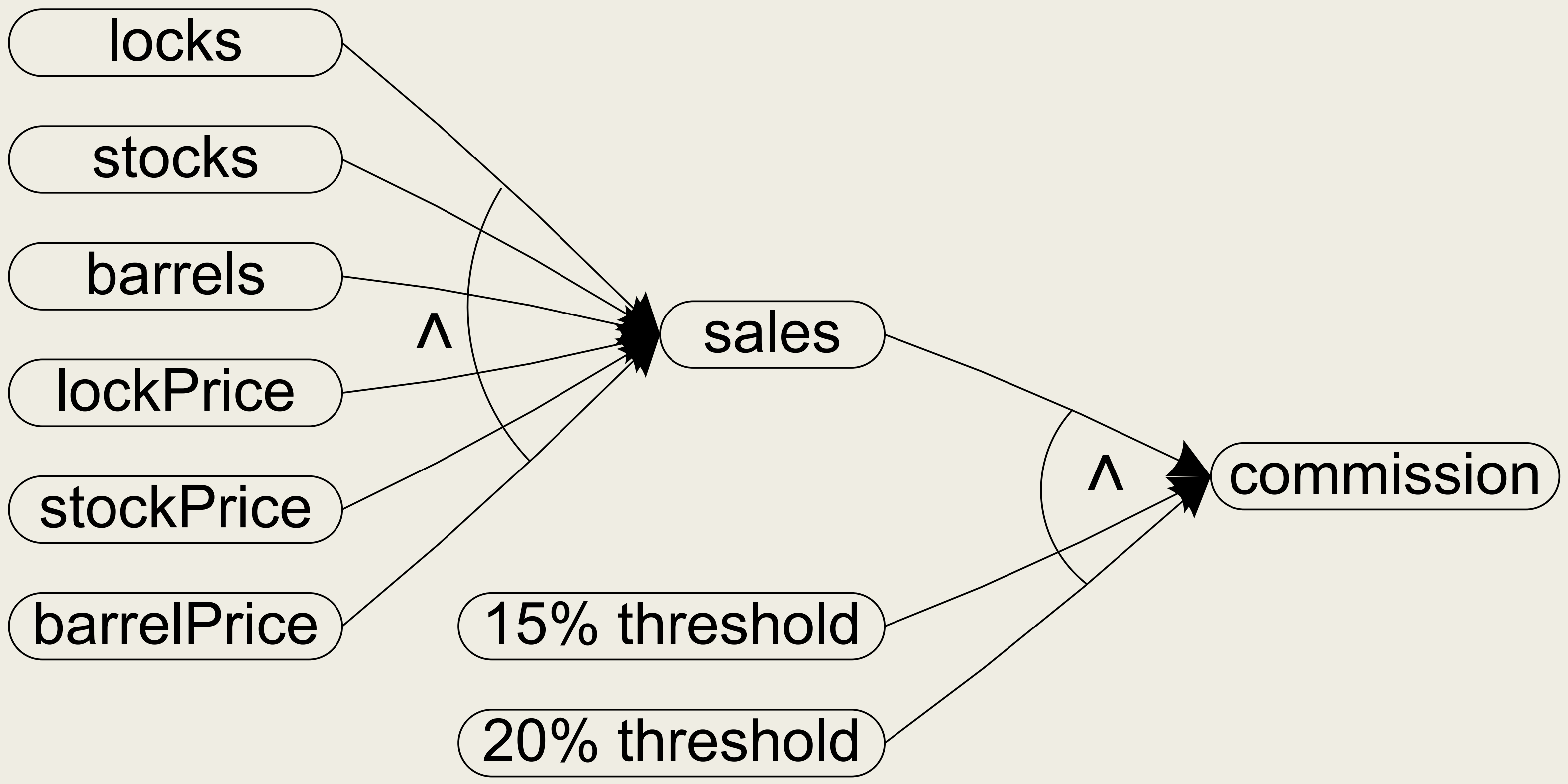








Cause and Effect Graph for the Commission Problem

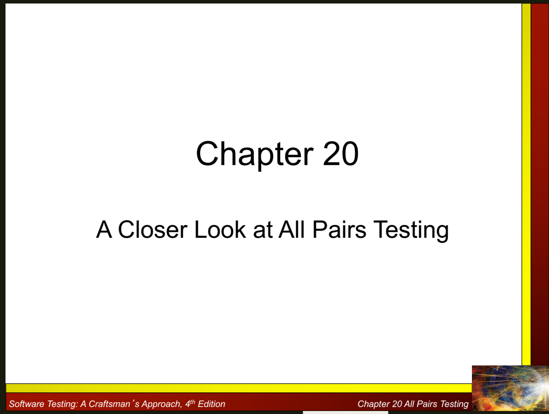


Cause and Effect Graph based Testing

- Cause and Effect Graphing
  - *Done with a graphical technique that expressed AND-OR-NOT logic.*
  - *Causes and Effects were graphed like circuit components*
  - *Inputs to a circuit "caused" outputs (effects)*
- **Equivalent to forming a decision table in which:**
  - ***inputs are conditions***
  - ***outputs are actions***
- Test every (possible) rule in the decision table.

Cause and Effect Graph testing example

https://www.softwaretestinghelp.com/cause-and-effect-graph-test-case-writing-technique/



Chapter 20

A Closer Look at All Pairs Testing

Software Testing: A Craftsman's Approach, 4th Edition

Chapter 20 All Pairs Testing

PAIRWISE TECHNIQUE (READING)

The All Pairs Testing Method

- Based on Orthogonal Arrays
- Drastically reduces number of test cases
- A combination of worst case and equivalence class testing
- Strong recommendation from Dorothy Wallace (NIST)
  - *"98% of the reported software defects in recalled medical devices could have been detected by testing all pairs of parameter settings."*
- Supported by commercial products

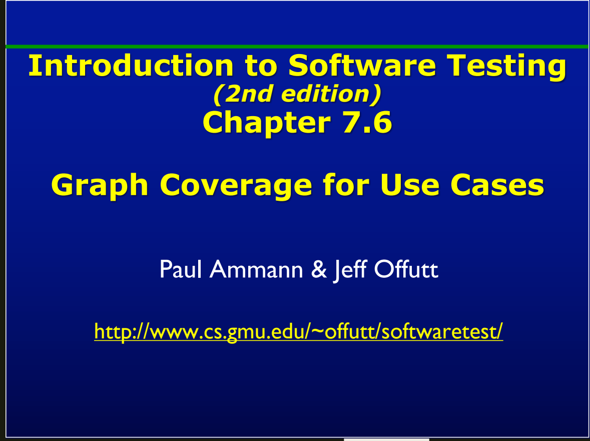
D.R. Wallace, D.R. Kuhn, *"Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," Intl. Journal of Reliability, Quality, and Safety Engineering, vol. 8, no. 4. (2001)*

All-Pairs Test Tools

- James Bach's program: allpairs.exe
  - *Free at https://www.satisfice.com*
  - *input is a Notepad file*

Bernie Berger, "Efficient Testing Using the Pairwise Approach", STAREast 2003 International Conference on Software Testing

- Twelve variables, with varying numbers of values, have
  7 x 6 x 6 x 5 x 3 x 3 x 2 x 2 x 2 x 2 x 2 x 2 = 725,760 combinations of values (test cases)
- "All Pairs does it in 50."



Introduction to Software Testing
(2nd edition)
Chapter 7.6

Graph Coverage for Use Cases

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

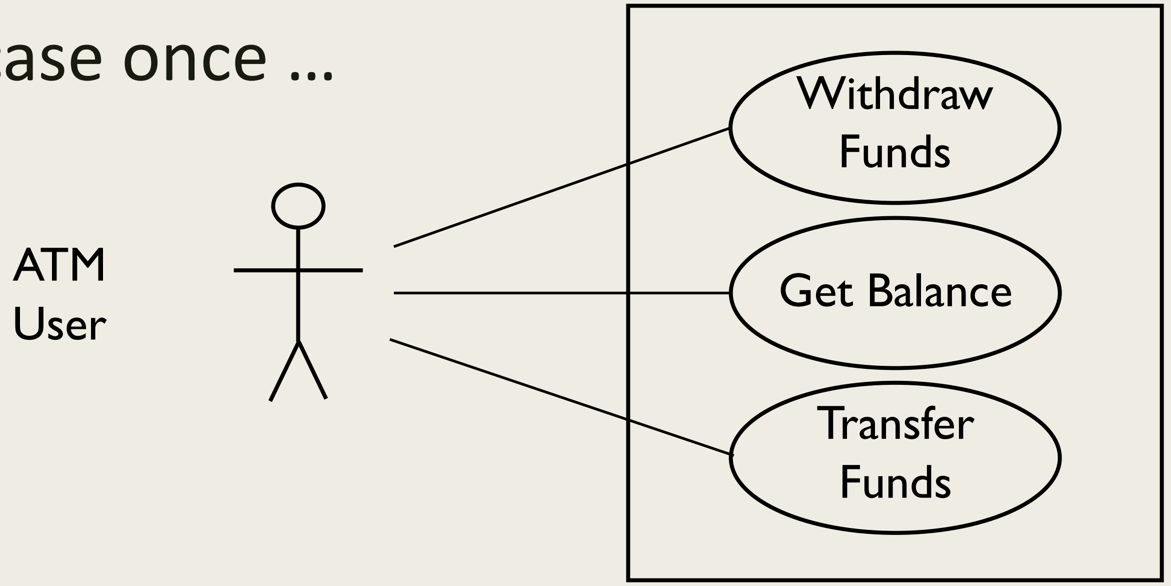
USE-CASE TESTING

UML Use Cases

- UML use cases are often used to express software requirements
- They help express computer application workflow
- We won’t teach use cases, but show examples

Simple Use Case Example

- Actors : Humans or software components that use the software being modeled
- Use cases : Shown as circles or ovals
- Node Coverage : Try each use case once …



Use case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly elaborated (or documented)
- Elaboration is first written textually
  - *Details of operation*
  - *Alternatives model choices and conditions during execution*

Elaboration of ATM Use Case

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - *Customer inserts an ATM Card into the ATM Card Reader.*
  - *If the system can recognize the card, it reads the card number.*
  - *System prompts the customer for a PIN.*
  - *Customer enters PIN.*
  - *System checks the card's expiration date and whether the card has been stolen or lost.*
  - *If the card is valid, the system checks if the entered PIN matches the card PIN.*
  - *If the PINs match, the system finds out what accounts the card can access.*
  - *System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)*

Elaboration of ATM Use Case—(2/3)

- Description (continued) :
  - *Customer selects Withdraw Funds, selects the account number, and enters the amount.*
  - *System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.*
  - *If all four checks are successful, the system dispenses the cash.*
  - *System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.*
  - *System ejects card.*
  - *System displays the idle welcome message.*

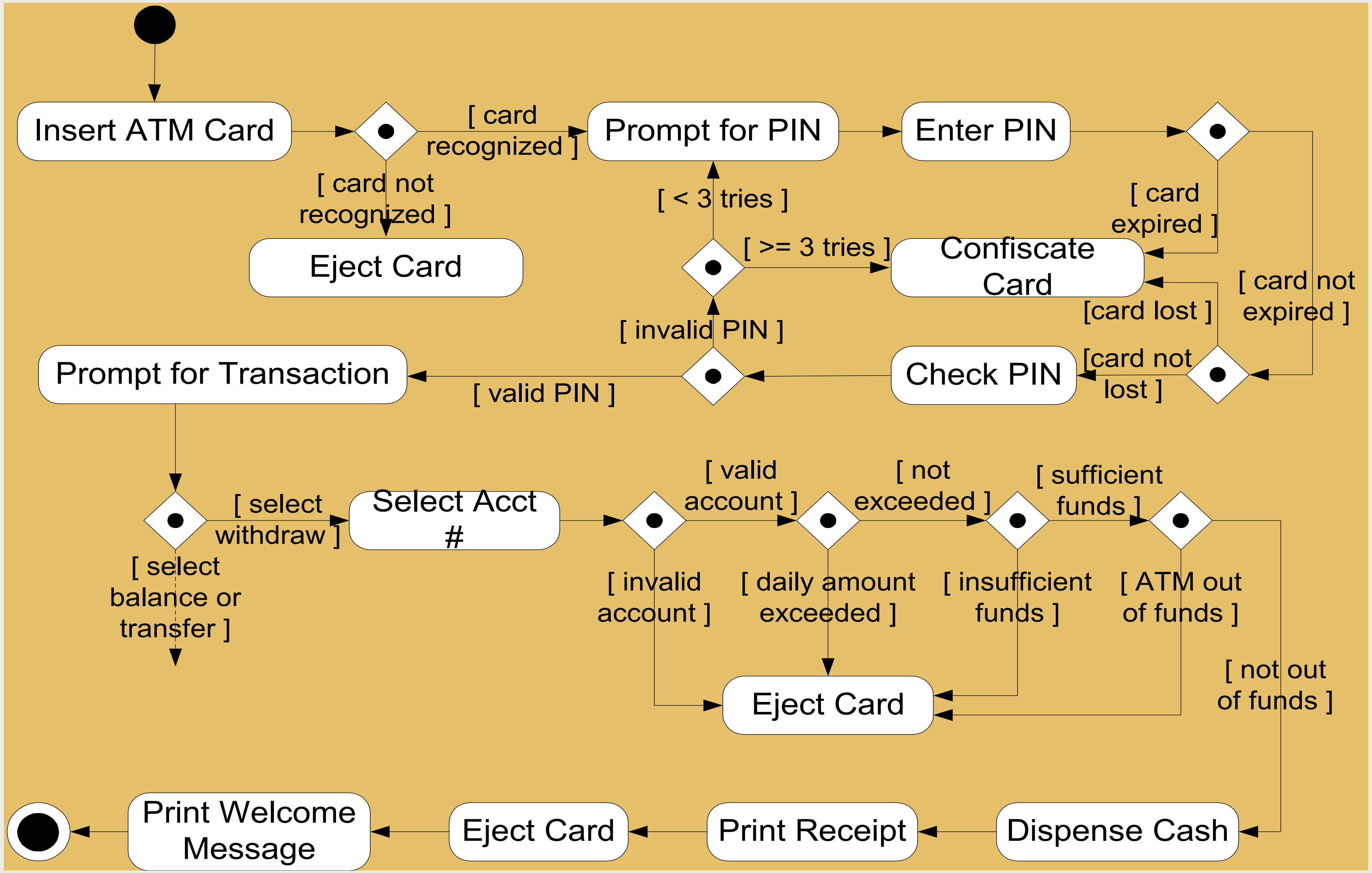
Elaboration of ATM Use Case—(3/3)

- Alternatives :
  - *If the system cannot recognize the card, it is ejected and the welcome message is displayed.*
  - *If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.*
  - *If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.*
  - *If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.*
  - *If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.*
  - *If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.*
  - *If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.*
  - *If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.*
  - *If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.*
- Postcondition :
  - *Funds have been withdrawn from the customer’s account.*

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - *Action states*
  - *Sequential branches*
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - *Few loops*
  - *Simple predicates*
  - *No obvious DU pairs*

ATM Withdraw Activity Graph



Covering Activity Graphs

- Node Coverage
  - *Inputs to the software are derived from labels on nodes and predicates*
  - *Used to form test case values*
- Edge Coverage
- Data flow techniques do not apply
- Scenario Testing
  - *Scenario : A complete path through a use case activity graph*
  - *Should make semantic sense to the users*
  - *Number of paths often finite*
  - *If not, scenarios defined based on domain knowledge*
  - *Use "specified path coverage," where the set S of paths is the set of scenarios*
  - *Note that specified path coverage does not necessarily subsume edge coverage, but scenarios should be defined so that it does*

Summary of Use Case Testing

- Use cases are defined at the requirements level
- Can be very high level
- UML Activity Diagrams encode use cases in graphs
  - *Graphs usually have a fairly simple structure*
- Requirements-based testing can use graph coverage
  - *Straightforward to do by hand*
  - *Specified path coverage makes sense for these graphs*

BB testing - Summary

- What is Black-box testing?
- Typical process
- Types
- Black-box vs. White-box testing

BB testing - Summary

- Boundary value analysis technique
  - *Input boundary testing: x(min), x(min+), x(max-), x(max)*
    - Normal Boundary Value Test Cases: two variables rarely both assume their extreme values
    - Robustness Testing: x(min-), x(min), x(min+), x(max-), x(max), x(max+)
    - Normal Worst Case Boundary Value Test Cases:
    - Robust Worst Case Boundary Value Test Cases
    - Special values
  - *Output Range Coverage: "Work "backwards" from expected outputs"*

BB testing - Summary

- Equivalence class partitioning technique
  - *Equivalence relations and partitioning*
  - *Forms of Equivalence Class Testing:*
    - “Traditional”: focus on invalid inputs
    - Normal: classes of valid values of inputs
    - Robust: classes of valid and invalid values of inputs
    - Weak: (single fault assumption) one from each class
    - Strong: (multiple fault assumption) one from each class in Cartesian Product

BB testing - Summary

- Domain analysis technique
  - *Testing of multiple variables simultaneously*
  - *Boundary analysis + Equivalence class partitioning*

BB testing - Summary

- Decision Table-Based Testing
  - *Condition => Action*
  - *Reducing the rules*

BB testing - Summary

- All pairs (pairwise testing) technique
  - *Based on Orthogonal array*
  - *The test set cover all values of each variables*
  - *For every pair of variables, cover all combinations of their values*
  - *Use when:*
    - Variables must be independent
    - Variables must be “logical”, not “physical”.
    - Variables have clear equivalence classes.
    - Failures only due to interaction of pairs of variables.
  - *Tool: allpairs.exe at www.satisfice.com*

BB testing - Summary

- FSM testing
  - *Determine the predicates (for transactions)*
  - *Find the truth assignments that let the all clauses determine the value of the predicate.*
  - *Map to expected results*
- Use-case testing
  - *To activity diagram*
  - *Test scenario: A complete path through a use case activity graph*
  - *Can apply the white-box testing techniques to ensure the path/node/... Coverage*

Summary - All

- Black-box testing
  - *Spec -> Test cases -> Test -> Report*
- Boundary value analysis technique
- Equivalence class partitioning technique
- Domain analysis technique
- Decision table technique
- Cause-effect graph technique
- Pairwise technique
- State transition technique
- Use-case testing