

CS608

Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

Tutorial: Lab 4

- Develop DT tests (combinations) for Insurance.premium()
 1. Analysis (Causes, Effects, Combinations, Decision Table)
 2. Test Coverage Items
 3. Test Cases
 4. Review
 5. Implement
 6. Run
 7. Results

CS608

Combination Testing with Decision Tables in More Detail

(Essentials of Software Testing, Chapter 4.4-4.7)

Fault Model

- The fault model is where **combinations of input values** are not processed correctly
- This leads to incorrect outputs
- Faults tend to be associated with an incorrect algorithm, or missing code, which do not identify or process the different combinations properly
- By testing every combination of values from the input partitions, decision table testing attempts to find these faults

Description

- The previous two techniques (equivalence partition and boundary value analysis) reduce the number of tests by not considering combinations
- Decision table testing provides additional coverage by identifying as few tests as possible to cover all the possible combinations that have an impact on the output
- The test objective is to achieve 100% coverage of the rules (i.e. combinations) in the decision table

Summary

- Analysis
 - Causes and Effects
 - Combinations
 - Decision Tables
- Test Coverage Items
- Test Cases

Analysis/Causes and Effects

- Examples of program behaviour:
 - Simple data processing just based on the individual input values
 - Complex processing based on the combinations of input values
- Handling complex combinations incorrectly can be a source of faults
- Analysis of combinations: identify all the different combinations of inputs/causes, and their associated outputs/effects
- Causes and effects are described as logical statements (predicates)
- These state how a particular variable should cause a particular effect
- Look at 4 examples

Causes: Example A

- Consider the method `boolean isNegative(int x)` as described previously
- There is a single cause:
 - `x < 0` (which can be true or false)
- And a single effect:
 - the return value (which can be true or false)
- For mutually exclusive expressions, such as `x < 0` and `x >= 0`:
 - only one form of the expression is needed, as it can take on the value true or false
- For boolean variables:
 - The two expressions `variable == true` and `variable == false` are not needed
 - The single expression `variable` can take on the value true or false

Causes: Example B

- Consider the method `boolean isZero(int x)` which returns `true` if `x` is zero, and otherwise returns `false`
- There is a single cause:
 - `x==0`
- And a single effect:
 - the return value

Causes: Example C

- Consider the method `int largest(int x, int y)`, which returns the largest of the two input values
- The causes are:
 - `x > y`
 - `x < y`
- And the effects are:
 - `return value == x`
 - `return value == y`

Notes on C

- Where there are three possible situations (here, $x < y$, $x == y$, and $x > y$) you need at least two expressions to cover them:
 1. If $x > y$, then $x < y$ is false
 2. If $x < y$, then $x > y$ is false
 3. If $x == y$, then $x > y$ and $x < y$ are both false.
- Where the effect is for an output (here the return value) to take on one of the input values, it is important to list all the possibilities, as they are not mutually exclusive
- In this case, when x is equal to y , the output is equal to both x and y

Causes: Example D

- Consider the method `boolean inRange(int x, int low, int high)`, which returns `true` when $\text{low} \leq x \leq \text{high}$. Note that the return value is undefined if $\text{high} < \text{low}$, so cannot test for this
- It is not optimal, but it would be possible to create 3 causes as follows:
 1. $x < \text{low}$
 2. $\text{low} \leq x \leq \text{high}$
 3. $x > \text{high}$
- However, mutually exclusive rules (where only one can be true) make for large decision tables, which are difficult to handle
- A preferred set of causes is as follows:
 1. $x < \text{low}$
 2. $x \leq \text{high}$

Notes on D

- Interpretation of combinations

$x < \text{low}$	$x \leq \text{high}$	Interpretation
T	T	x out of range, $x < \text{low}$
T	F	not possible, x cannot be both smaller than low and greater than high
F	T	x in range, $x \geq \text{low}$ and $x \leq \text{high}$
F	F	x out of range, $x > \text{high}$

- This both reduces the number of causes
- And allows for better use of combinations of causes (as they are no longer completely mutually exclusive)

Example D - Effects

- Only one effect for the boolean return value (true or false):
 1. return value
- Notes:
 - It is redundant to use the effect `return value==true`. The expression `return value` is a boolean expression, and the T in the decision table indicates when it is true or false
 - For a boolean return value, it is also redundant to use two effects:
 1. return value
 2. `!(return value)`
 - Only a single rule is required:
 1. return value

Valid Combinations of Causes

- Before developing a decision table, the infeasible combinations of input causes should be removed
- The decision table then only contains feasible combinations, thus reducing work which would be wasted by removing these later
- **Example E**
 - Boolean condInRange(int x, int low, int high, Boolean flag)
 - Only returns true if flags if **true** and **low**≤**x**≤**high**

Example E: All Combinations

- The causes are:
 1. flag
 2. $x < \text{low}$
 3. $x \leq \text{high}$
- Full set of combinations

Causes	All Combinations							
flag	T	T	T	T	F	F	F	F
$x < \text{low}$	T	T	F	F	T	T	F	F
$x \leq \text{high}$	T	F	T	F	T	F	T	F

Example E: Feasible Combinations

Causes	All Combinations							
flag	T	T	T	T	F	F	F	F
$x < \text{low}$	T	T	F	F	T	T	F	F
$x \leq \text{high}$	T	F	T	F	T	F	T	F

- Highlighted combinations are infeasible, as x cannot be both less than low and greater than high (for normal/non-error values of low and high)
- All Feasible Combinations:

Causes	All Combinations					
flag	T	T	T	F	F	F
$x < \text{low}$	T	F	F	T	F	F
$x \leq \text{high}$	T	T	F	T	T	F

Analysis: Decision Tables

- A Decision Table is used to map the causes and effects through rules
- Each rule states that under a particular combination of input causes, a particular set of output effects should result
- Only one rule may be active at a time: the decision table is invalid if any input matches more than one rule
- To generate the decision table, each cause is listed in a separate row, and each combination of causes listed in a separate column creates different effects
- Each column is referred to as a Rule in the table
 - each rule is a different test coverage item, and a different test case
- T is used as shorthand for true, and F for false

Analysis: Decision Tables

- A Decision Table is used to map the causes to the effects
- Each rule states that under a particular combination of causes, a particular set of output effects should result
- Only one rule may be active at a time: exactly one rule matches more than one rule
- To generate the decision table, each combination of causes listed in a separate column
- Each column is referred to as a Rule
 - each rule is a different test coverage item
- T is used as shorthand for true, and F for false

In mathematics, these tables are referred to as Truth Tables, as they only contain the values true and false

The term Decision Tables is widely used in software testing as they define the decisions to be made by the software

“Cause-Effect Graphs” can also be used – but very hard to use properly, especially for large graphs

Decision Table for isNegative()

		Rules	
		1	2
Causes	$x < 0$	T	F
Effects	return value	T	F

- **Rule 1** states that if $x < 0$, then the return value is true
- **Rule 2** states that if $!(x < 0)$, then the return value is false

Decision Table for isZero()

		Rules	
		1	2
Causes	$x == 0$	T	F
Effects	return value	T	F

- **Rule 1** states that if $x == 0$, then the return value is true
- **Rule 2** states that if $!(x == 0)$, then the return value is false

Decision Table for largest()

		Rules		
		1	2	3
Causes	$x > y$	T	F	F
	$x < y$	F	T	F
Effects	return value == x	T	F	T
	return value == y	F	T	T

- **Rule 1** states that if $(x > y)$ and $!(x < y)$, then the return value is x
- **Rule 2** states that if $!(x > y)$ and $(x < y)$, then the return value is y
- **Rule 3** states that if $!(x > y)$ and $!(x < y)$, implying that $(x == y)$, then the return value is equal to both x and y

Decision Table for inRange()

		Rules		
		1	2	3
Causes	$x < \text{low}$	T	F	F
	$x \leq \text{high}$	T	T	F
Effects	return value	F	T	F

- **Rule 1** states that if $(x < \text{low})$ and $(x \leq \text{high})$, then the return value is false
- **Rule 2** states that if $!(x < \text{low})$ and $(x \leq \text{high})$, then the return value is true
- **Rule 3** states that if $!(x < \text{low})$ and $!(x \leq \text{high})$, then the return value is false

New Example: condInRange(flag,x)

- Return true if and only if flag is true and x is in the range low..high

Decision Table for condInRange()

		Rules					
		1	2	3	4	5	6
Causes	flag	T	T	T	F	F	F
	$x < \text{low}$	T	F	F	T	F	F
	$x \leq \text{high}$	T	T	F	T	T	F
Effects	return value	F	T	F	F	F	F

- **Rule 1:** if (flag) and ($x < \text{low}$) and ($x \leq \text{high}$), then return value is false
- **Rule 2:** if (flag) and $\neg(x < \text{low})$ and ($x \leq \text{high}$), then the return value is true
- **Rule 3:** if (flag) and $\neg(x < \text{low})$ and $\neg(x \leq \text{high})$, then the return value is false
- **Rule 4:** if $\neg(\text{flag})$ and ($x < \text{low}$) and ($x \leq \text{high}$), then the return value is false
- **Rule 5:** if $\neg(\text{flag})$ and $\neg(x < \text{low})$ and ($x \leq \text{high}$), then the return value is false
- **Rule 6:** if $\neg(\text{flag})$ and $\neg(x < \text{low})$ and $\neg(x \leq \text{high})$, then the return value is false

		Rules					
		1	2	3	4	5	6
Causes	flag	T	T	T	F	F	F
	x<low	T	F	F	T	F	F
	x≤high	T	T	F	T	T	F
Effects	return value	F	T	F	F	F	F

Poor Decision Table for condInRange()

		Rules															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Causes	flag	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
	x<low	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
	low≤x≤high	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
	x>high	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F
Effects	return value==true				F		T	F					F		F	F	
	return value==false				T		F	T					T		T	T	

- Even if you remove the invalid combinations, the table is not being used for its purpose, which is to generate the combinations
- And still contains unnecessary duplication
- The previous slide contains exactly the same information in a much more concise form

Handling Large Tables

- With more than 16 rules the tables become too large to handle easily
- Two techniques for reducing the size of large tables are:
 - Sub-Tables
 - Pairwise Testing
- We won't cover the other standard technique “don't-care conditions”
 - It is considered an advanced technique and is difficult to implement correctly
 - It also makes some assumptions that the software is implemented correctly

Sub-Tables

		Rules					
		1	2	3	4	5	6
Causes	flag	T	T	T	F	F	F
	$x < \text{low}$	T	F	F	T	F	F
	$x \leq \text{high}$	T	T	F	T	T	F
Effects	return value	F	T	F	F	F	F

condInRange() with flag true

		Rules		
		1	2	3
Causes	$x < \text{low}$	T	F	F
	$x \leq \text{high}$	T	T	F
Effects	return value	F	T	F

condInRange() with flag false

		Rules		
		1	2	3
Causes	$x < \text{low}$	T	F	F
	$x \leq \text{high}$	T	T	F
Effects	return value	F	F	F

Sub-Tables and Error Rules

- We have already used this technique to remove error rules
- If there are interesting combinations of causes that produce errors, then these can be presented in a separate table (with error rules only)

Pairwise Testing

- Large decision tables can be reduced by limiting the number of rules
- Instead of including **every** combination of causes, we can instead include just the combinations for every pair of causes
- The technique involves identifying every possible pair of combinations, and then combining these pairs into as few rules as possible

Example of Pairwise Testing for condInRange()

- All the possible pairs of causes are included at least once

		Rules				
		1	2	3	4	5
Causes	flag	T	T	T	F	F
	$x < \text{low}$	T	F	F	T	F
	$x \leq \text{high}$	T	T	F	T	F
Effects	return value	F	T	F	F	F

- Pairs of causes where both ($x < \text{low}$) and ($x \leq \text{high}$) are not possible and are not included in the pairwise decision table

Example of Pairwise Testing for condInRange()

		Rules				
		1	2	3	4	5
Causes	flag	T	T	T	F	F
	x < low	T	F	F	T	F
	x ≤ high	T	T	F	T	F
Effects						
return value		F	T	F	F	F

- flag and x < low – Rule 1
- flag and !(x < low) – Rule 2
- !flag and x < low – Rule 4
- !flag and !(x < low) – Rule 5

T T F F
T F T F

Example of Pairwise Testing for condInRange()

		Rules				
		1	2	3	4	5
Causes	flag	T	T	T	F	F
	x < low	T	F	F	T	F
	x ≤ high	T	T	F	T	F
Effects	return value	F	T	F	F	F

- Note: TF not possible

T	T	F	F
T	F	T	F

Example of Pairwise Testing for condInRange()

		Rules				
		1	2	3	4	5
Causes	flag	T	T	T	F	F
	x < low	T	F	F	T	F
	x ≤ high	T	T	F	T	F
Effects	return value	F	T	F	F	F

T T F F
T F T F

Example of Pairwise Testing for condInRange()

		Rules				
		1	2	3	4	5
Causes	flag	T	T	T	F	F
	x<low	T	F	F	T	F
	x≤high	T	T	F	T	F
Effects	return value	F	T	F	F	F

- flag and x<low – Rule 1
- flag and !(x<low) – Rule 2
- !flag and x<low – Rule 4
- !flag and !(x<low) – Rule 5
- flag and x≤high – Rule 1
- flag and !(x≤high) – Rule 3
- !flag and x≤high – Rule 4
- !flag and !(x≤high) – Rule 5
- x<low and x≤high – Rule 1
- !(x<low) and x≤high – Rule 2
- !(x<low) and !(x≤high) – Rule 3

Size Reduction (using Pairwise Testing)

- The larger the original table, the larger the reduction
- Example: 16 rules to 6 rules

Rules															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F



Rules					
1	2	3	4	5	6
T	T	F	F	T	F
T	F	T	F	T	F
T	T	F	T	F	F
T	F	T	T	F	F

Effectiveness of Pairwise Testing?

- Extensive debate!
- The method does not find N-way faults (associated with a combination of N different causes, with $N > 2$)
- Some research suggests that the additional benefits of higher-order (3-way, 4-way, N-way) combinations may not be cost effective
- The obvious benefit is in reducing the number of test cases
- Pairwise Testing also reduces the time for test design (fewer tests to design), implementation (fewer tests to code), and execution (fewer tests to run).

Test Coverage Items (TCI)

- Each rule in the decision table is a test coverage item
- If a test item includes multiple decision tables (for example, in a class) then it is useful to signify each rule/test coverage item with a unique identifier (i.e. a different name)
- Software tools can help to automatically produce the required sets of pairs
- Usually a decision table will not include error situations, due to the number of rules required to describe all of these
- Errors would be included in a separate table for clarity, and only when different combinations of causes result in different errors

Test Cases

- Input test data for the test cases is selected, based on rules not yet covered
- Each test case will cover exactly one test coverage item (or rule)
- The expected results values are derived from the specification (using the decision table)
- Hint: easiest to identify test data by going through the rules in order, and selecting a value for each parameter that matches the required causes
- It is easier to remove duplicates, and review the test design for correctness, if you use as few different values as possible for each parameter
- So it is recommended to re-use the equivalence partition values

Pitfalls

- Ensure that the causes are complete and do not overlap
 - Otherwise the table may be incomplete or inconsistent
- Use simple logical expressions for causes (i.e. with no boolean operators, such as the && or || operators)
 - Otherwise it is difficult to ensure all the combinations are created by the table.
- The rules must be unique, you must ensure that no two rules can be true at the same time (i.e. every rule must have a different combination of causes)
 - Otherwise the table is invalid and cannot be used to derive test cases.
- Ensure that there are no possible combinations of input values that cause no rules to be matched

Evaluation

- Decision table testing approaches black-box testing from a slightly different viewpoint compared to equivalence partition and boundary value analysis testing
 - Instead of considering each input individually
 - Consider combinations of input values
- This provides increased coverage of decisions in the code relating to:
 - categorising the inputs
 - or deciding what action to take based on a combination of inputs
- Complex decisions are a frequent source of mistakes in the code
- These decisions generally reflect the correct identification of a complex situation, and if not correctly implemented can result in the wrong processing taking place

Limitations

- The software has passed all the EP, BVA & DT tests
- So is it fault free?

Is It Fault Free?

- Only exhaustive testing can answer this question
- Faults may remain
- Explore limitations of decision table testing by injecting two different categories of fault into the code

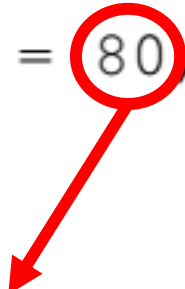
Correct Code

```
22     public static Status giveDiscount(long bonusPoints, boolean
        goldCustomer)
23     {
24         Status rv = FULLPRICE;
25         long threshold=120;
26
27         if (bonusPoints<=0)
28             rv = ERROR;
29
30         else {
31             if (goldCustomer)
32                 threshold = 80;
33             if (bonusPoints>threshold)
34                 rv=DISCOUNT;
35         }
36
37         return rv;
38     }
```

NOTE: If
bonusPoints is
greater than the
threshold, return
DISCOUNT

Recap: Fault 3

```
--  
31      if (goldCustomer)  
32          threshold = 80;  
  
31      if (goldCustomer)  
32          threshold = 120; // fault 3
```



Run DT Tests Against Fault 3



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 80, false, FULLPRICE)
PASSED: test_giveDiscount("T2.3", 81, false, FULLPRICE)
PASSED: test_giveDiscount("T2.4", 120, false, FULLPRICE)
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 9223372036854775807, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -9223372036854775808, false, ERROR)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
FAILED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
java.lang.AssertionError: expected [DISCOUNT] but found [FULLPRICE]
=====
Command line suite
Total tests run: 14, Passes: 13, Failures: 1, Skips: 0
=====
```

Fault 4

The inserted fault (lines 35 and 36) adds extra functionality that is not in the specification and is unlikely to be found by any type of black-box testing

```
22     Status rv = FULLPRICE;
23     long threshold=120;
24
25     if (bonusPoints<=0)
26         rv = ERROR;
27
28     else {
29         if (goldCustomer)
30             threshold = 80;
31         if (bonusPoints>threshold)
32             rv=DISCOUNT;
33     }
34
35     if (bonusPoints==43) // fault 4
36         rv = DISCOUNT;
37
38     return rv;
39 }
40
41 }
```

Run DT Tests Against Fault 4



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 80, false, FULLPRICE)
PASSED: test_giveDiscount("T2.3", 81, false, FULLPRICE)
PASSED: test_giveDiscount("T2.4", 120, false, FULLPRICE)
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 9223372036854775807, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -9223372036854775808, false, ERROR)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
```

```
=====
Command line suite
Total tests run: 14, Passes: 14, Failures: 0, Skips: 0
=====
```


Run DT Tests Against Fault 4

As expected, the fault is not found

The inserted fault bears no relationship to the specification, and therefore is unlikely to be found by any black-box testing technique

```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
34
35     if (bonusPoints==43) // fault 4
36         rv = DISCOUNT;
37
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 922337203685477, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -922337203685477, false, DISCOUNT)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
=====
Command line suite
Total tests run: 14, Passes: 14, Failures: 0, Skips: 0
=====
```

Demonstrating The Fault

- The results of executing the code with Fault 4, using specially selected input values, are:

```
$ check 43 true  
DISCOUNT
```



- Here, the wrong result is returned for the inputs (43,true)
 - The correct result is FULLPRICE
 - Not DISCOUNT

Strengths & Weaknesses

- Strengths
 - Exercises combinations of input data values
 - Expected results are created as part of the analysis process
- Weaknesses
 - The decision tables can sometimes be very large (different approaches exist to reduce the tables)
 - A more detailed specification may lead to increased causes and effects, and consequently large and complex tables.

Key Points

- Decision table testing ensures that the software works correctly with combinations of input values
- Test coverage items and test cases are based on rules in the decision table
- The data for the test cases is selected from the equivalence partition test values

Notes for Experienced Testers

- Experienced testers may produce the causes and effects in their mind and create decision tables directly with support from the specification
- They may also implement tests directly from the decision table
- However, in cases where high quality is required (embedded systems, life-critical systems, etc.) even an experienced tester may need to document these steps for quality review, or in the case of a legal challenge to the quality of the software

Lab 4 – Insert Faults

- Insert fault into `Insurance.premium()` that is found by EP, BVA & DT
- Insert fault into `Insurance.premium()` that is not found by EP, BVA & DT