Black-Box and White-Box Testing

Comparison

Black-Box Testing	White-Box Testing
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated, which may be to fix a fault or add new features.	Tests are generally invalidated by any changes to the code, and cannot be reused.
Tests can be developed before the code is written as they only require the specification.	Tests can only be developed after the code is written as they require both the specification and the executable code.
Tests do not ensure that all of the code has been executed. They may miss *errors of commission* (see Section 8.1.3).	Tests do not ensure that the code fully implements the specification. They miss *errors of omission*.
Few tools provide automated coverage measurement of any black-box tests.	Many tools provide automated coverage measurement of white-box tests.

Basic BBT

- **EP**: Use equivalence partitions to verify the basic operation of the software by ensuring that one representative of each partition is executed
- **BVA**: f the specification contains boundary values, use boundary value analysis to verify correct operation at the edges
- **CIT**: If the specification states different processing for different combinations of inputs, or there is a complex input model, with many multi-valued parameters, a combinatorial interaction test suite can exercise it effectively

Basic WBT

- **Measure** the statement and branch coverage for black-box tests
  - If required coverage not achieved, add white-box testing
  - To enhance quality of testing
- **SC**: Use statement coverage to ensure that 100% of the statements have been executed
- **BC**: Use branch coverage to ensure that 100% of the branches have been taken

Extra WBT

- **AP**: If code contains complex end-to-end paths, use all paths testing
- **DU-Pairs**: if code contains complex data usage patterns, then use du pair testing
  - Described later in this lecture
- **CC, DC, DCC, MCC, MCDC**: If code contains complex decisions, then use these forms of testing to ensure coverage of these
  - Described later in this lecture

Equivalence Partitions

Equivalence Partitions

- An Equivalence Partition is **a range of values for a parameter** for which the specification states **equivalent processing**
- Example:
  - boolean isNegative(int x)
  - Returns true if x is negative, otherwise false
- Two equivalence partitions for the parameter x can be identified:
  - Integer.MIN VALUE..-1
  - 0..Integer.MAX VALUE

Step 1. Analysis

- Analyse the specification to identify the equivalence partitions
- Two stages:
  - first identify the **natural ranges** for each parameter
  - then identify the **specification-based ranges** (or equivalence partitions)
- Based on the principle of equivalent processing

Natural Ranges

- Natural ranges for common types
  - **byte** [Byte.MIN VALUE..Byte.MAX VALUE]
  - **short** [Short.MIN VALUE..Short.MAX VALUE]
  - **int** [Integer.MIN_VALUE..Integer.MAX_VALUE]
  - **long** [Long.MIN VALUE..Long.MAX VALUE]
  - **char** [Character.MIN VALUE..Character.MAX VALUE]
- For types with no natural ordering, each value is a separate range containing one value
  - **boolean** [true][false]
  - **enum Colour {Red, Blue, Green}** [Red][Blue][Green]
  - Treat numeric outputs with limited, specific values in the same way as enum's

EP Example

- Two equivalence partitions for the parameter x can be identified:
  - Integer.MIN VALUE..-1
  - 0..
- These specification-based ranges are called equivalence partitions
- According to the specification, any value in the partition is processed equivalently to any other value

Analysis/Parameters

- Methods (and functions) have **explicit and implicit** parameters
- Explicit parameters are passed in the method call
- Implicit parameters are not:
  - in C may be **global variables**
  - in Java may **be attributes**
- Both types of parameter must be considered in testing
- A complete specification should include all inputs and outputs.

Test Coverage Items (TCI)

- **Each partition** for each input and output **is a test coverage item**
- It is good practice to give each test coverage item (for each test item) a unique identifier
- It is often useful to use the prefix "EP-" for EP test coverage items
- So, for example, for two methods, you can have:
  - method1(): EP1, EP2, EP3, etc.
  - method2(): EP1, EP2, EP3, etc.
- Selected values used in the Test Cases

giveDiscount()

Status giveDiscount(long bonusPoints, boolean goldCustomer)

Inputs

bonusPoints: the number of bonusPoints the customer has accumulated

goldCustomer: true for a Gold Customer

Outputs

return value:

FULLPRICE if bonusPoints≤120 and not a goldCustomer
FULLPRICE if bonusPoints≤80 and a goldCustomer
DISCOUNT if bonusPoints>120
DISCOUNT if bonusPoints>80 and a goldCustomer
ERROR if any inputs are invalid (bonusPoints<1)

Status is defined as follows:

```
enum Status { FULLPRICE, DISCOUNT, ERROR };
```


Equivalence Values

Parameter	Equivalence Partition	Equivalence Value
bonusPoints	Long.MIN_VALUE..0	-100
	1..80	40
	81..120	100
	121..Long.MAX_VALUE	200
goldCustomer	true	true
	false	false
Return Value	FULLPRICE	FULLPRICE
	DISCOUNT	DISCOUNT
	ERROR	ERROR

•

Test Coverage Items for giveDiscount()

TCI	Parameter	Equivalence Partition	Test Case
EP1*	bonusPoints	Long.MIN_VALUE..0	To be completed later
EP2		1..80	
EP3		81..120	
EP4		121..Long.MAX_VALUE	
EP5	goldCustomer	true	
EP6		false	
EP7	Return Value	FULLPRICE	
EP8		DISCOUNT	
EP9		ERROR	

Selecting Next Uncovered TCIs

TCI	Parameter	Equivalence Partition	Test Case
EP1*	bonusPoints	Long.MIN_VALUE..0	To be completed later
EP2		1..80	
EP3		81..120	
EP4		121..Long.MAX_VALUE	
EP5	goldCustomer	true	
EP6		false	
EP7	Return Value	FULLPRICE	
EP8		DISCOUNT	
EP9		ERROR	

EP Test Cases for giveDiscount()

ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T1.1	EP2,5,7	40	true	FULLPRICE
T1.2	EP3,6,[7]	100	false	FULLPRICE
T1.3	EP4,[6],8	200	false	DISCOUNT
T1.4*	EP1*,9	-100	false	ERROR

- Each **input error** test coverage item is tested separately
- **Minimising** the number of test cases can require multiple iterations
- Target – the maximum number of partitions of any input or output:
  - In this example it is 4 (for the bonusPoints parameter)
  - Not always achievable
- Combinations of input values are not considered
  - In this example, no test for bonusPoints equal to 40 with goldCustomer both true and false

Reviewing the Test Coverage Items

- Once complete, check that
  - every TCI is covered by at least one test
  - no duplicate (redundant) tests

ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T1.1	EP2,5,7	40	true	FULLPRICE
T1.2	EP3,6,[7]	100	false	FULLPRICE
T1.3	EP4,[6],8	200	false	DISCOUNT
T1.4*	EP1*,9	-100	false	ERROR

TCI	Parameter	Equivalence Partition	Test Case
EP1*	bonusPoints	Long.MIN_VALUE..0	T1.4
EP2		1..80	T1.1
EP3		81..120	T1.2
EP4		121..Long.MAX_VALUE	T1.3
EP5	goldCustomer	true	T1.1
EP6		false	T1.2
EP7	Return Value	FULLPRICE	T1.1
EP8		DISCOUNT	T1.3
EP9		ERROR	T1.4

Fault Model

- The equivalence partition fault model is where **entire ranges of values are not processed correctly**
- These faults can be associated with incorrect decisions in the code, or missing sections of functionality
- By testing with **at least one value** from every equivalence partition, where every value should be **processed in the same** way, equivalence partition testing attempts to find these faults

Fault at boundaries

- Equivalence partition tests are not designed to find faults at the values at each end of an equivalence partition
- If we inject a fault which moves the boundary value for the processing that returns DISCOUNT, then we do not expect to see any failed tests

```
33          if (bonusPoints>threshold)
34                  rv=DISCOUNT;
```

```
33          if (bonusPoints>=threshold) // fault
34              rv=DISCOUNT;
```

Error Hiding

```
1   // return true if both x and y are valid
2   //       (within the range 0..100)
3   // otherwise return false to indicate an error
4   public static boolean valid(int x, int y) {
5      if (x<0 || x>100) return false;
6      if (y<0 || y>1000) return false;
7      return true;
8   }
```

- Fault on line 6 – should use 100, not 1000
- Example Error TCIs: x=500, y=500
- Test with multiple error TCIs
  - assertFalse(valid(500,500)) – PASSES
- Does NOT find the error on line 6 – line 5 returns false first
- **The Error TCI "y=500" is NOT tested**

9. Evaluation

- Testing with equivalence partitions provides a **minimum level** of black-box testing
- At least one value has been tested from every input and output partition...
- ...using a minimum number of tests cases
- Ensure correctness of basic data processing
- Do not exercise: different decisions made in the code
- Decisions frequent source of mistakes – generally reflect **boundaries** of input partitions, or **combinations** of inputs requiring particular processing
- These issues will be addressed in later techniques

Boundary Value Analysis

Testing with Boundary Value Analysis

- Boundary values are the minimum and maximum values for each Equivalence Partition
- Having identified the partitions, identifying the boundary values is straightforward
- The goal is to verify that the software works correctly at these boundaries

Typical Fault in Boundary Value Handling

- Line 3 contains a fault
  !(x<100) is wrong
- Either !(x<=100) or (x>100) would be correct

```
1   // Return true is x is greater than 100
2   public void greater(int x) {
3      if (!(x<100)) return true;
4      else return false;
5   }
```

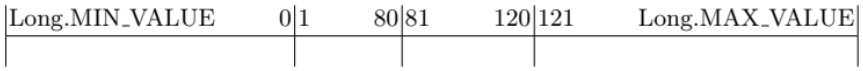

Example

- Continue testing OnlineSales.giveDiscount()
- Summary – the method returns:

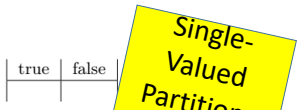
FULLPRICE if bonusPoints≤120 and not a goldCustomer
FULLPRICE if bonusPoints≤80 and a goldCustomer
DISCOUNT if bonusPoints>120
DISCOUNT if bonusPoints>80 and a goldCustomer
ERROR if any inputs are invalid (bonusPoints<1)

Specification-Based Ranges

- bonusPoints



- goldCustomer



Single-Valued Partitions

- Return Value

Single-Valued Partitions

2. Test Coverage Items

TCI	Parameter	Boundary Value	Test Case
BV1*	bonusPoints	Long.MIN_VALUE	To be completed later
BV2*		0	
BV3		1	
BV4		80	
BV5		81	
BV6		120	
BV7		121	
BV8		Long.MAX_VALUE	
BV9	goldCustomer	true	
BV10		false	
BV11	Return Value	FULLPRICE	
BV12		DISCOUNT	
BV13		ERROR	

3. Test Cases (complete)

ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T2.1	BV3,9,11	1	true	FULLPRICE
T2.2	BV4,10,[11]	80	false	FULLPRICE
T2.3	BV5,[10,11]	81	false	FULLPRICE
T2.4	BV6,[10,11]	120	false	FULLPRICE
T2.5	BV7,[10],12	121	false	DISCOUNT
T2.6	BV8,[10,12]	Long.MAX_VALUE	false	DISCOUNT
T2.7	BV1*,13	Long.MIN_VALUE	false	ERROR
T2.8	BV2*,[13]	0	false	ERROR

Completed Test Coverage Item Table

TCI	Parameter	Boundary Value	Test Case
BV1*	bonusPoints	Long.MIN_VALUE	T2.7
BV2*		0	T2.8
BV3		1	T2.1
BV4		80	T2.2
BV5		81	T2.3
BV6		120	T2.4
BV7		121	T2.5
BV8		Long.MAX_VALUE	T2.6
BV9	goldCustomer	true	T2.1
BV10		false	T2.2
BV11	Return Value	FULLPRICE	T2.1
BV12		DISCOUNT	T2.5
BV13		ERROR	T2.7

Evaluation

- Boundary value analysis enhances the testing provided by equivalence partitions
- Experience indicates that this is likely to **find significantly more errors** than equivalence partitions
- Boundary value analysis tests two **additional** values from every input and output partition, the minimum and maximum
- These tests provide some assurance that the **correct decisions** are made in the code
- Boundary value analysis provides exactly the same test coverage items as equivalence partitions for boolean and for enumerated parameters

BVA: Decisions and Combinations

- Boundary value analysis does **not** explore the decisions in detail
- In particular BVA does not explore decisions associated with different combinations of inputs
- Combinations of inputs will be addressed next

Combinatorial testing

Prof. Andrea Calvagna

Motivation:

- verification of reactive systems
  - many configuration options
  - incomplete/unavailable specification
- go for BB testing, but....
  - partition testing not very effective in revealing faults
  - boundary testing does not exercise all combinations of inputs

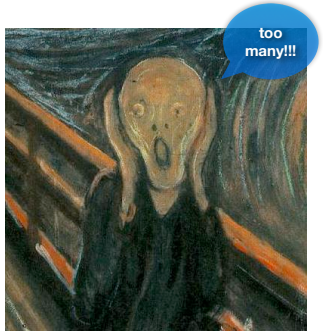
Combinatorial testing

- cover all combinations of inputs?
  - requires only knowledge of the input domain
  - still lots of tests

Example: 20 variables,10 values each:

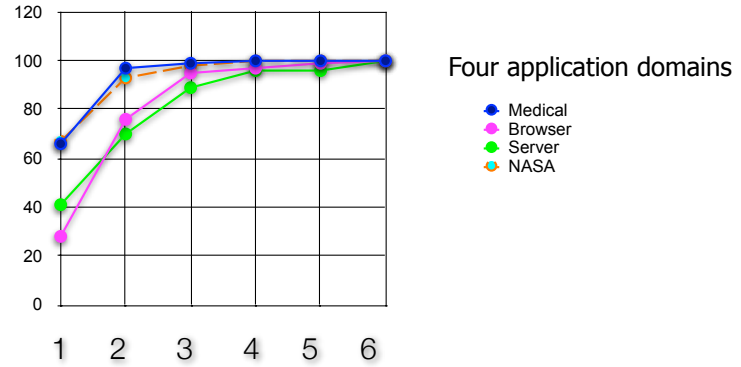
10^20 combinations

A test for each atom in the universe



Feature Interaction testing

- all failures triggered by 1 to 6-way interaction [Kuhn, Okun@NIST]



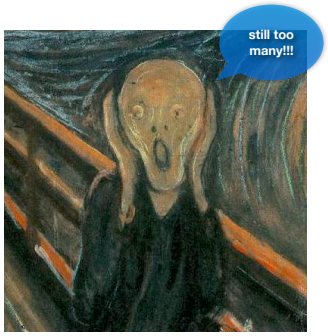
Feature Interaction testing

- Test combinations up to a given size (degree/strength)
- Most failures triggered by interaction of **just pairs** of input settings
  - test coverage items are all pairs of input parameter values
- empirical data show pairwise testing detect usually over 60% of faults
- go for higher interaction degrees, if required

k-wise Combinatorial Interaction test suites

For *n* variables with *v* values, *k*-way combinations $\binom{n}{k}v^k$

- Example:
- 4-way combinations of 30 parameters, 5 values each: **3,800** test goals



- Task: build a **covering array**
- minimal, if possible
- Task is NP-complete

COVERING ARRAY

- $CA_\lambda(N; t,K,V)$, $\lambda=1$
- $MCA_\lambda(N; t,K,V)$, $V=\{v_1,v_2,...,v_K\}$
  - $V=\{2,2,2,2,3,3,4\} \Rightarrow 2^4 3^2 4^1$
- $OA_\lambda(N; t,K,V)$
- latin squares

OLD CHINESE MAGIC SQUARE

Existing tools

- commercial tools: AETG, testCover, …
- research tools: IPO - PairTest, IBM whitch, MS Pict, TTuples…
- www.pairwise.org

Most focus on optimization for size and speed

- $16^4 10^{26}$ -> 256 tests,
- 2^{120} -> 10 tests

SPIN model checker

- MCA(N;2,$2^{13}4^5$) ,i.e., space of $8.3*10^6$ different system configurations.
- 13 pairwise constraints
- relate 9 of the18 factors
- no implied forbidden pairs.
- enforcing just 1 of the13 constraints eliminates over 2 million configurations.

SPIN verifier

- MCA(N;2,$2^{42}3^24^{11}$);
- space of $1.7*10^{20}$ different configurations.
- 49 constraints
- 47 over pairs of combinations and 2 over triples
- 33 ofthe 55 factors are involved in constraints.
- 9 implied forbidden pairs
- a single constraint eliminates more than $2*10^{19}$

GCC 4.1

- MCA(N ; 2, 2^{189} 3^{10})
- state space of $4.6*$ 10^{61} different configurations.
- 40 constraints, 3 three-way and 37 pairwise
- related to 35 of the 199 factors
- 2 implied forbidden pairs
- $1.2 * 10^{61}$ conf. x constraint

RESULTS

	mAETG_SAT		SA_SAT	
	without constraints	with constraints	without constraints	with constraints
$t = 2$				
SPIN Verifier	33 9.8 sec	41 32.2 sec	27 19.6 sec	35 31,595.2 sec
SPIN simulator	25 0.4 sec	24 1.5 sec	16 25.6 sec	19 694.3 sec
GCC	24 323.2 sec	23 371.6 sec	16 4,137.0 sec	20 18,186.2 sec
$t = 3$				
SPIN simulator	100 6.3 sec	108 11.9 sec	78 1,577.5 sec	95 13,337.4 sec

Table 3: CIT size and time for GCC and SPIN

Statement Coverage

Testing with Statement Coverage

- Statement coverage (SC) testing (or more formally testing with statement coverage criteria) ensures that **every line, or statement,** of the source code **is executed** during tests, while verifying that the **output is correct**

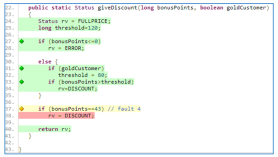
Definition:
a statement is a line in the source code that can be executed
(**Not all lines can be executed:** comments, blank lines, {}'s, etc.)

Test Coverage Items

- Each statement in the source code is a test coverage item
  - Every **executable** statement
- Normally, just consider 'extra' TCIs (not already covered by BB tests)
- Normally, a single line of source is regarded as being a statement
- Issues:
  - Multiple statements on one line
  - Multi-line statements

Identifying Unexecuted Statements

- **Using coverage results** from previous tests (black-box tests), unexecuted statements easily identified
  - **You can use "JaCoCo" – see www.jacoco.org**
- For complex code, a **Control-Flow Graph (CFG)** may be developed first to help with understanding the code flow at a more abstract level, but these are seldom required for statement coverage
- Statement coverage tests may be developed before black-box tests, though this is not usual practice
  - In this case, CFGs are traditionally used to assist in developing the tests



Coverage results

- Examine the source code coverage report
- determine which lines have not been executed

- Uncovered item at line 38
- identify the condition that will cause line 38 to execute: bonusPoints=43

```
19.    * FULLPRICE - charge the full price<br>
20.    * ERROR - invalid inputs
21.    */
22.   public static Status giveDiscount(long bonusPoints
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.      if (bonusPoints==43) // fault 4
38.         rv = DISCOUNT;
39.
40.      return rv;
41.   }
42.
43. }
```


Adding Extra Test Cases

- Unlike in black-box testing, there is no differentiation between error and non-error cases for white-box testing

ID	TCI	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T4.1	SC1	43	false	FULLPRICE

Strengths

- Provides a minimum level of coverage by executing all the statements in the code at least once
- There is a significant risk in releasing software before every statement has been executed at least once during testing
  - its behaviour has not been verified, and may well be faulty
- Statement coverage can generally be achieved using only a small number of extra tests

The **full test** implementation **includes** the previously developed equivalence **partition and boundary** value analysis tests!!

Weaknesses

- Can be difficult to determine the required input parameter values
- Hard to test code only executed in unusual circumstances
- **Does not provide coverage for the NULL else**
  - `if (number < 3) number++;`
  - Statement coverage does not force a test case for number ≥ 3
- Not demanding of compound decisions
  - `if ( (a>1) || (b==0) ) then x = x/a;`
  - No test cases for the different boolean conditions that may cause the decision to be true or false
  - No test cases for the possible combinations of the boolean conditions

Branch Coverage

Branch Coverage

- Essentially the same, except that we **consider branches instead of statements**
- Note that there are different ways to measure branches:
  - Every statement with a whole decision being true or false
  - Every binary condition in a decision being true or false
- JaCoCo measures the outcome of every Boolean condition in a decision as a separate branch
  - Based on the Java bytecode

Branches Explained

```
1 if (x && y)
2       z = 10;
3 else
4       z = 20;
```

- Decisions-Based

Two branches:

1. Line 1 to line 2 if x and y are true
2. Line 1 to line 4 otherwise

- Conditions-Based

Four branches:

1. Condition x to y if x is true
2. Condition x to line 4 if x is false
3. Condition y to line 2 if y is true
4. Condition y to line 4 if y is false

Fault Model

- Branches that has not been taken in previous tests (**untaken branches**) **may contain a fault**
- As for statement coverage, these tend to be associated with edge cases, or other unusual circumstances
- Branch coverage tests with input values carefully selected to **ensure** that **every branch is taken** during test execution
- These tests attempt to find **faults associated with individual branches** in the source code

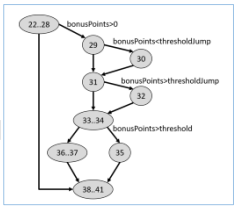
Using JaCoCo

- Can use previous coverage to start from
- JaCoCo counts the outcome of each **boolean condition** as a branch
- a different tool may count the outcomes of each **decision** as branches instead

```
27.     if (bonusPoints<=0)
28.         rv = ERROR;
29.
30.     else {
31.         if (goldCustomer && bonusPoints!=93) // fa
32.             threshold = 80;
33.         1 of 4 branches missed.  nts>threshold)
34.             rv=DISCOUNT;
35.     }
36.
37.     return rv;
38.   }
39.
```

- This would reduce the number of branches, which leads to slightly reduced test effectiveness

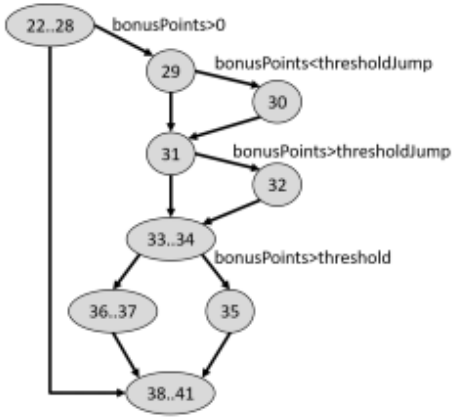
"From Scratch"



- Develop a CFG – all edges are branches
- Decisions (not the boolean expressions) invariably used
- Seldom used in practice for two reasons
  1. Time consuming to develop CFG for significant code
  2. If the code is changed, either to fix a fault or add new features, the control flow graph will have to be reviewed possibly re-done. And the associated test implementation redeveloped
- The rapid change of code in a modern, Agile development environment makes this approach less realistic than tools-based/ coverage measurement

Control Flow Graph

```
22    public static Status giveDiscount(long bo
          goldCustomer)
23    {
24       Status rv = ERROR;
25       long threshold=goldCustomer?80:120;
26       long thresholdJump=goldCustomer?20:30;
27
28       if (bonusPoints>0) {
29          if (bonusPoints<thresholdJump)
30             bonusPoints -= threshold;
31          if (bonusPoints>thresholdJump)
32             bonusPoints -= threshold;
33          bonusPoints += 4*(thresholdJump);
34          if (bonusPoints>threshold)
35             rv = DISCOUNT;
36          else
37             rv = FULLPRICE;
38        }
39
40       return rv;
41    }
```



Strengths and Weaknesses

- Strengths:
  - Branch coverage is a **stronger** form of testing than statement coverage: 100% branch coverage guarantees 100% statement coverage – but the test data is harder to generate
  - **Resolves the NULL else problem**
- Weaknesses:
  - Can be **difficult to determine the required input** parameter values
  - If the tool only counts decisions as branches, or if a control flow graph has been manually developed, then it is undemanding of compound decisions. In these cases it does not explore all the different reasons (i.e. the boolean conditions) for the decision evaluating as true or false

All Paths Coverage

Introduction to All Paths Coverage (AP)

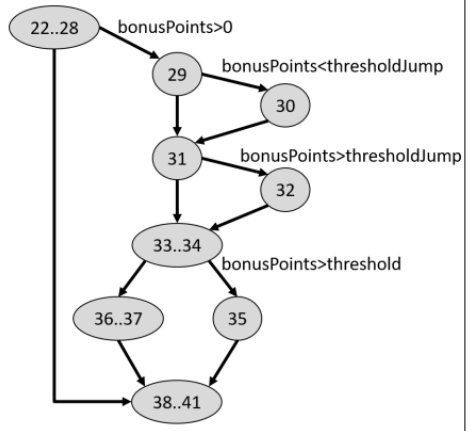
- **The strongest form** of white-box testing based on "**program structure**"
- If you achieve AP, then you have also achieved SC and BC
- All the paths, from entry to exit of a block of code (usually a method; the test item), are executed during testing
- Developing all paths tests is complex and time consuming
- In practice it is seldom used (only considered for critical software)

Control Flow Graphs (CFGs)

- In statement and branch testing:
  - Control Flow Graphs (CFGs) are seldom used in practice
  - Tests can be more efficiently developed to supplement black-box tests
  - Using automated tools to measure the coverage
- However, **in all paths testing, CFGs are essential**
- Today we will see how to develop and use them

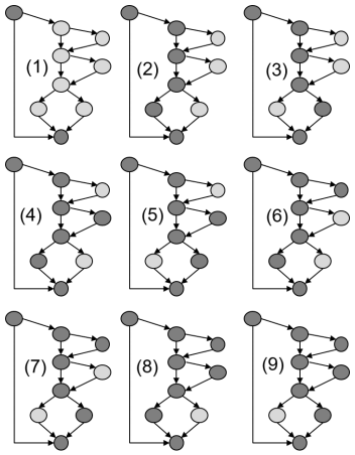
Definition

- Definition: an **end-to-end path** is a single flow of execution from the start to the end of a section of code
- This path may include multiple executions of any loops on the path
- **Complessità ciclomatica** (cc) = numero di archi - numero di nodi + 2
- Per testare tutti le condizioni (**coprire tutto il cfg)**, il numero di test da effettuare è cc
- Non per testare tutte le **combinazioni dei percorsi**



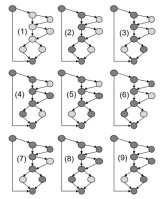
Identifying the Candidate Paths

- By tracing all the start-to-end paths through the CFG, the candidate paths can be identified
- These are candidate paths: not of them are logically possible to take
- A systematic approach must be taken to avoid missing paths
- Work through the CFG starting at the top, and taking the left hand branches first



Identifying the Possible Paths

- Path 1, nodes (22..28)–(38..41) covered by bonusPoints=-100, and goldCustomer=false (T1.4)
- Path 2, nodes (22..28)–(29)–(31)–(33..34)–(36..37)–(38..41) not possible
- Path 3, nodes (22..28)–(29)–(31)–(33..34)–(35)–(38..41) covered by 20, true or 30, false
- Path 4, nodes (22..28)–(29)–(31)–(32)–(33..34)–(36..37)–(38..41) covered by 1, true (T2.1)
- Path 5, nodes (22..28)–(29)–(31)–(32)–(33..34)–(35)–(38..41) covered by 40, true (T1.1)
- Path 6, nodes (22..28)–(29)–(30)–(31)–33..34)–(36..37)–(38..41) not possible
- Path 7, nodes (22..28)–(29)–(30)–(31)–(33..34)–(35)–(38..41) not possible
- Path 8, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(36..37)–(38..41) not possible
- Path 9, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(35)–(38..41) not possible



Analysis Results

- This analysis of the CFG has produced three results:
  1. The possible end-to-end paths
  2. Where available, existing test data that causes a path to be taken
  3. Where no previous test causes a path to be taken, criteria or values for the new test data

Path	Input Values	Existing Test
1	(-100,false)	T1.4
3	(20,true) or (30,false)	New test required
4	(1,true)	T2.1
5	(40,true)	T1.1

Fault Model

- The all-paths fault model is where a particular sequence of operations, reflected by a particular path through the code, does not produce the correct result
- These faults are often associated with complex or deeply nested code structures where the **wrong functionality** is executed for a specific situation
- Trovo errori funzionali complessi che si manifestano solo con l'esecuzione di **una sequenza più lunga di un singolo branch**

Strengths and Weaknesses

- **Strengths**
  - Covers all possible paths, which may have not been exercised using other methods
  - Guarantees statement coverage and branch coverage coverage
- **Weaknesses**
  - Difficult and time consuming
  - All-paths does not explicitly evaluate the boolean conditions in each decision
  - Does not explore faults related to incorrect data processing (e.g. bitwise manipulation or arithmetic errors)
  - Does not explore non-code faults (for example, faults in a lookup table)

WBT: Some More Techniques

- Dataflow coverage/Definition-Use pairs
- Condition Coverage (CC)
- Decision Coverage (DC)
- Decision/Condition Coverage (DCC)
- Multiple Condition Coverage (MCC)
- Modified Condition/Decision Coverage (MCDC)
- Test ranking

Dataflow coverage/Definition-Use pairs

- Each **path between** the writing of a value to a variable (**definition**) and its subsequent reading (**use**) is executed during testing
  - Definition: **x=10**
  - Use: **y=x+20** or **if (x>100)**
- Every possible du pair is a test coverage item
- Strengths
  - This is a strong form of testing.
  - It generates test data in the pattern that data is manipulated in the program
- Weaknesses
  - The number of test cases can be very large
  - Hard to handle object references (C/C++ pointers) and arrays

Condition Coverage (CC)

- Each **boolean condition** within a decision is tested for its true and false values
- TCI for “if (a || (b && c))”
  - a and !a
  - b and !b
  - c and !c
- Does not ensure decision is true and false
- Can be difficult to determine the required input parameter values

Decision Coverage (DC)

- In decision coverage (often abbreviated to DC), **every decision** is evaluated to true and false
- This is **equivalent to branch coverage** if a coverage measurement tool is used that identifies decisions as branches (or if a control flow graph has been used)

Decision/Condition Coverage (DCC)

- In decision condition coverage (often abbreviated to DCC) , tests are generated that **both** cause **every decision** to be taken at least once (decision coverage), **and** also **every** boolean **condition** to be true and false at least once (condition coverage)
- TCIs
  - Each value of each conditions (true and false)
  - Each value of each decision (true and false)
- Doesn't test every combination of conditions
- Can be difficult to determine required input parameter values

Multiple Condition Coverage (MCC)

- Tests are generated to cause **every possible combination** of boolean **conditions** for **every decision** to be tested
- Use a Decision Table
- Each decision with **n independent** boolean conditions has **2^n test** coverage items
- A large number of TCIs
- Can be difficult to determine required input parameter values

Modified Condition/Decision Coverage (MCDC)

- MCC generates a very large number of tests
- Reduced by **only considering those combinations** of Boolean conditions that cause a discernible **effect on the output**
- The test conditions:
  - decision condition coverage, plus
  - additional conditions to verify the independent effect of each boolean condition on the output
- Each decision has two test coverage items (true and false)
- Every boolean condition in each decision has two test coverage items
- In addition, test coverage items must be created that show the effect on the output of changing each of the boolean conditions independently

MCDC Example

```
1   int func(int a, int b) {
2      int x=100;
3      if ( (a>10) || (b==0) ) then x = x/a;
4      return x;
5   }
```

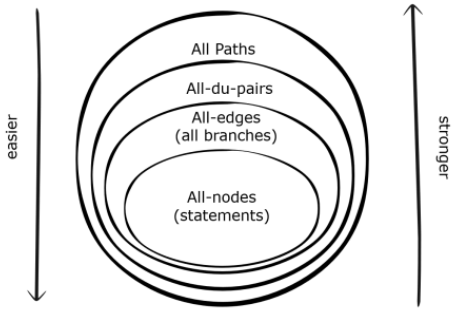
- Tests must ensure each condition true and false:
  - a>10
  - b==0
- Tests must ensure each decision true and false:
  - (a>10) || (b==0)
- Tests must ensure the effect on the output value of changing the value of every boolean condition is shown

MCDC Example/Test Data

```
1   int func(int a, int b) {
2      int x=100;
3      if ( (a>10) || (b==0) ) then x = x/a;
4      return x;
5   }
```

- Inputs (a=50,b=1) should result in the output value: 2
- (a=5,b=1) should result in the output value: 100
  - This shows the independent effect of changing the boolean condition (a>10)
- The input data (a=5,b=0) should result in the output value: 20
  - This shows the independent effect of changing the boolean condition (b==0)

Structural Test ranking



15 min. break