Efficient Test Suite Management

Prof. Durga Prasad Mohapatra
Professor
Dept.of CSE, NIT Rourkela

Why do test suites grow?

- Test cases in an existing test suite can often be used to test a modified program.
- However, if the test suite is inadequate for retesting, new test cases may be developed and added to the test suite.
- There may be unnecessary test cases in the test suite including both obsolete and redundant test cases.

Why do test suites grow? (contd..)

- A change in a program causes a test case to become obsolete by removing the reason for the test case's inclusion in the test suite.
- A test case is redundant if other test cases in the test suite provide the same coverage of the program.
- Thus due to obsolete and redundant test cases, the size of a test suite continues to grow unnecessarily.

Why test suite minimization is important?

- release date of the product is near
- limited staff to execute all the test cases
- limited test equipment or unavailability of testing tools.

Advantages of minimizing a test suite

- redundant test cases will be eliminated
- reduces the cost of the project by reducing a test suite to a minimal subset.
- decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software, thereby reducing the cost of regression testing.
- so it is a great practical advantage to reduce the size of test cases.

Defining test suite minimization problems

Harrold et al. have defined the problem of minimizing test suite as given below.

- **Given:** a test suite TS; a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired testing coverage of the program; and subsets of TS: $T_1, T_2, \ldots, T_n$, one associated with each of the r_i's such that any one of the test cases t_j belonging to T_i can be used to test r_i.

Defining test suite minimization problems (contd..)

- **Problem:** find a representative set of test cases from TS that satisfies all the r_i's
  - a representative set of test cases that satisfies the r_i's must contain at least one test case from each T_i,
  - such a set is called a **hitting set** of the group of sets, $T_1, T_2, ..., T_n$.

Defining test suite minimization problems (contd..)

- The r_i's can represent either all the test case requirements of a program or those requirements related to program modifications.
- maximum reduction is achieved by finding the smallest representative of test cases,
- however this subset of the test suite is the minimum cardinality hitting set of the Ti's.
- the problem of finding the minimum cardinality hitting set is NP-complete.
- thus minimization techniques resort to heuristics.

Test Suite Prioritization

- The reduction process can be understood if the cases in a test suite are prioritized in some order.
- The purpose of prioritization is to reduce the set of test cases based on some rational, non-arbitrary criteria, while aiming to select the most appropriate tests.
- For example, the following priority categories can be determined for the test cases.

Test Suite Prioritization cont…

- ***Priority I*** *The test cases must be executed*, otherwise there may be worse consequences after the release of the product.
- For example, if the test cases for this category are not executed, then critical bugs may appear.

Test Suite Prioritization cont…

- ***Priority 2*** *The test cases may be executed, if time permits.*
- ***Priority 3*** *The test case is not important prior to the current release.*
- It may be tested shortly after the release of the current version of the software.

Test Suite Prioritization cont…

- ***Priority 4*** *The test case is never important, as it's impact is nearly negligible.*
- In the prioritization scheme, the main guideline is to ensure that low-priority test cases do not cause any severe impact on the software.
- There may be several goals of prioritization.

Test Suite Prioritization cont…

- These goals can become the basis for prioritizing the test cases. Some of them are discussed here:

1. Testers or customers may want to get some critical features tested and presented in the first version of the software.
   - Thus, the important features become the criteria for prioritizing the test cases.
   - But the consequences of not testing some low-priority features must be checked.
   - So, risk factor should be analyzed for every feature in consideration.

Test Suite Prioritization cont…

2. Prioritization can be on the basis of the functionality advertised in the market.

It becomes important to test those functionalities on a priority basis, which the company has promised to its customers.

3. The rate of fault detection of a test suite can reveal the likelihood of faults earlier.

Test Suite Prioritization cont…

4. To increase the coverage of coverable code in the system under test at a faster rate, allowing a code coverage criterion to be met earlier in the test process.

5. To increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.

Test Suite Prioritization cont…

6. To increase the likelihood of revealing faults related to specific code changes, earlier in the regression testing process.

Types of Test Case Prioritization

- General Test Case Prioritization
- Version-Specific Test Case Prioritization

General Test Case Prioritization

- In this prioritization, we prioritize the test cases that will be useful over a succession of subsequent modified versions of P, without any knowledge of the modified version.
- Thus, a general test case prioritization can be performed following the release of a program version during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases.

Version-Specific Test Case Prioritization

- Here, we prioritize the test cases such that they will be useful on a specific version P' of P.
- Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P', with the knowledge of the changes that have been made.

Prioritization Techniques

- Prioritization techniques schedule execution of test cases in an order that attempts to increase their effectiveness at meeting some performance goal.
- Various prioritization techniques may be applied to a test suite with the aim of meeting that goal.
- Prioritization can be done at two levels:
  - Prioritization for regression test suite
  - Prioritization for system test suite

Types of Prioritization Techniques

- Coverage-based Test Case Prioritization
- Risk-Based Prioritization
- Prioritization Based on Operational Profiles
- Prioritization using Relevant Slices
- Prioritization Based on Requirements

Coverage-based Test Case Prioritization

a) Total statement Coverage Prioritization
b) Additional Statement Coverage Prioritization
c) Total Branch Coverage Prioritization
d) Additional Branch Coverage Prioritization
e) Total Fault-Exposing-Potential(FEP) Prioritization

Total statement Coverage Prioritization

- This prioritization orders the test cases based on the total number of statements covered.
- It counts the number of statements covered by the test cases and orders them in a descending order.
- If multiple test cases cover the same number of statements, then a random order may be used.
- For example, if T_1 covers 5 statements, T_2 covers 3, and T_3 covers 12 statements; then according to this prioritization, the order will be T_3, T_1, T_2.

Additional statement coverage prioritization

- Total statement coverage prioritization schedules the test cases based on the total statements covered.
- However, it will be useful if it can execute those statements as well that have not been covered yet.
- It iteratively selects a test case T_1, that yields the greatest statement coverage, and then selects a test case which covers a statement not covered by T_1.
- Repeat this process until all statements covered by at least one test case have been covered.

Additional statement coverage prioritization cont…

- For example, if we consider Table 1, according to total statement coverage criteria, the order is 2, 1, 3.
- But additional statement coverage selects test case 2 first and next, it selects test case 3, as it covers statement 4 which has not been covered by test case 2. Thus, the order according to addition coverage criteria 2, 3, 1.

Table 1. Statement coverage

Statement	Statement Coverage		
	Test Case 1	Test Case 2	Test Case 3
1	X	X	X
2	X	X	X
3		X	X
4			X
5			
6		X	
7	x	X	
8	X	X	
9	X	x	

Total branch coverage prioritization

- In this prioritization, the criterion to order is to consider condition branches in a program instead of statements.
- Thus, it is the coverage of each possible outcome of a condition in a predicate.
- The test case which will cover maximum branch outcomes will be ordered first.
- For example, in Table 2, the order will be 1, 2, 3.

Total branch coverage prioritization cont…

Table 2. Branch coverage

Branch statements	Branch Coverage		
	Test case 1	Test case 2	Test case 3
Entry to while	X	X	X
2-true	X	X	x
2-false	X		
3-true		X	
3-false	X		

Additional Branch Coverage Prioritization

- The idea is the same as in additional statement coverage of first selecting the test case with the maximum coverage of branch outcomes and
  - then, selecting the test case which covers the branch outcome not covered by the previous one.

Total Fault-Exposing-Potential (FEP) Prioritization

- Statement and branch coverage prioritization ignore a fact about test cases and faults:
  - Some bugs/faults are more easily uncovered than other faults.
  - Some test cases have the proficiency to uncover particular bugs as compared to other test cases.

Total Fault-Exposing-Potential (FEP) Prioritization

- Thus, the ability of a test case to expose a fault is called the fault exposing potential.
- It depends not only on whether test cases cover a faulty statement, but also on the probability that a fault in that statement will also cause a failure for that test case.
- To obtain an approximation of the FEP of a test case, an approach was adopted using mutation analysis.

This approach is discussed below.

- Given program P and test suite T,
- First create a set of mutants N = {n1, n2,…..nm} for P, noting which statement s_j in P contains each mutant.
- Next, for each test case t_i ɛ T, execute each mutant version n_k of P on t_i , noting whether t_i kills that mutant.
- Calculate FEP(s,t) = Mutants of s_j killed / Total number of mutants of s_j.
- To perform total FEP prioritization, given these FEP(s,t) values, calculate an *award value* for each test case ti ɛ T, by summing the FEP(sj, ti) values for all statements s_j in P. Given these award values, we prioritize test cases by sorting them in order of descending award value.

The next table shows FEP estimates of a program. Total FEP prioritization outputs the test order as (2, 3, 1).

Statement	FEP(s,t) values		
	Test case 1	Test case 2	Test case 3
1	0.5	0.5	0.3
2	0.4	0.5	0.4
3		0.01	0.4
4		1.3	
5			
6	0.3		
7	0.6		0.1
8		0.8	0.2
9			0.6
Award values	1.8	3.11	2.0

Summary

- Discussed why do test suites grow.
- Explained why test suite minimization is important.
- Explained the basics of test suite prioritization.
- Presented the different types of prioritization techniques.
- Discussed in detail the different types of coverage-based test case prioritization techniques.

References

1. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 12), Second Edition, Oxford University Press, 2016.

Thank you

Efficient Test Suite Management cont …

Prof. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela

Risk-Based Prioritization

- It is a well defined process that prioritizes modules for testing.
- Uses risk analysis to highlight potential problem areas, whose failure have adverse consequences.
- Testers use this risk analysis to select the most crucial tests.

Risk-Based Prioritization cont …

- This technique is used to prioritize the test cases based on *some potential problems* which may occur during the project. It uses:
  - **Probability of occurrence/fault likelihood:** It indicates the probability of occurrence of a problem.
  - **Severity of impact/failure impact:** If the problem has occurred, how much impact does it have on the software.

Risk-Based Prioritization cont …

- Risk analysis uses these two components by first listing the potential problems and then, assigning a probability and severity value for each identified problem, as shown in next Table.
- By ranking the results in this table in the form of risk exposure, testers can identify the potential against which the software needs to be tested and executed first.

Risk-Based Prioritization cont …

A risk analysis table consists of the following columns:

- **Problem ID:** A unique identifier to facilitate referring to a risk factor.
- **Potential problem:** Brief description of the problem.
- **Uncertainty factor:** It is the probability of occurrence of the problem. Probability values are on a scale of 1(low) to 10(high).
- **Severity of impact:** Severity values on a scale of 1(low) to 10(high).
- **Risk exposure:** Product of probability of occurrence and severity of impact.

Table 1 .A sample risk analysis table

Problem ID	Potential Problem	Uncertainty Factor	Risk Impact	Risk Exposure
P1	Specification ambiguity	2	3	6
P2	Interface Problem	5	6	30
P3	File corruption	6	4	24
P4	Databases not synchronized	8	7	56
P5	Unavailability of modules for integration	9	10	90

Inference from the table

The problems / modules given in the previous table can be prioritized in the order of P5, P4, P2, P3, P1, based on the risk exposure values.

Prioritization Based on Operational Profiles

- In this approach, the test planning is done based on the operational profiles of the important functions which are of use to the customer.
- An operational profile is a set of tasks performed by the system and their probabilities of occurrence.
- After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.

Prioritization using Slices

- During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original program, except where a change is excepted.
- But re-running the test suite for every change in the software makes regression testing a time-consuming process.

Prioritization using Slices cont …

- If we can find the portion of the software which has been affected with the change in software, then we can prioritize the test cases based on this information.
- This is called the slicing technique.

Execution Slice

- The set of statements executed under a test case is called the **execution slice** of the program.
- Please refer to the following program.
- Table 2 shows the test cases for the given program.

```
Begin
S1: read (basic, empid);
S2: gross = 0;
S3: if (basic > 5000 || empid > 0)
        {
S4:      da = (basic*30)/100;
S5:      gross = basic + da;
        }
S6:  else
        {
S7:      da = (basic*15)/100;
S8:      gross = basic + da;
        }
S9: print (gross, empid);
End
```

Fig 1. Example program for execution

Table 2 Test cases

Test Case	Basic	Empid	Gross	Empid
T1	8000	100	10400	100
T2	2000	20	2300	20
T3	10000	0	13000	0

Prioritization using Slices cont …

- T1 and T2 produce correct results.
- On the other hand, T3 produces an incorrect result.
- Syntactically, it is correct, but an employee with the empid '0' will not get any salary, even if his basic salary is read as input.
- So it has to be modified.

Prioritization using Slices cont …

- Suppose S3 is modified as [if(basic>5000 && empid>0)].
- So, for T1, T2, and T3, the program would be rerun to validate whether the change in S3 has introduced new bugs or not.
- But, if there is a change in S7,[da = (basic*25)/100; instead of da = (basic*15)/100;], then only T2 will be rerun.

Prioritization using Slices cont …

- So in the execution slice, we will have less number of statements.
- The execution slice is highlighted in the given code segment.

Example

```
Begin
   S1: read (basic, empid);
   S2: gross=0;
   S3: if(basic > 5000 || empid > 0)
            {
   S4:      da = (basic*30)/100;
   S5:      gross = basic + da;
            }
   S6: else
            {
   S7:      da = (basic*15)/100;
   S8:      gross = basic + da;
            }
   S9: print(gross, empid);
End
```

Dynamic Slice

The set of statements under a test case having an effect on the program output is called the **dynamic slice** of the program with respect to the out-put variables.

Example

```
Begin
   S1: read (a,b);
   S2: sum=0;
   S2.1: I=0;
   S3: if (a==0)
            {
   S4:          print(b);
   S5:          sum+=b;
            }
   S6: else if(b==0)
            {
   S7:          print(a);
   S8:          sum+=a;
            }
   S9: else
            {
   S10:         sum=a+b+sum;
   S10.1        I=25;
   S10.2        print(I);
            }
   S11:endif
   S12:print(sum);
End
```

Test Cases

Table 3: Test cases for the given program.

Test Case	a	b	sum
T1	0	4	4
T2	67	0	67
T3	23	23	46

Dynamic Slice cont …

- T1, T2 and T3 will run correctly but if some modification is done in S10.1 [say I = 50], then this change will not affect the output variable.
- So, there is no need to rerun any of the test cases.
- On the other hand, if S10 is changed [say, sum =a* b+sum], then this change will affect the output variable 'sum'.
- So there is a need to rerun T3.
- The dynamic slice is highlighted in the code segment (S1, S2,S10, S12).

```
Begin
   S1: read (a,b);
   S2: sum=0;
   S2.1: I=0;
   S3: if (a==0)

          {
   S4:       print(b);
   S5:       sum+=b;
          }
   S6: else if(b==0)
          {
   S7:       print(a);
   S8:       sum+=a;
          }
   S9: else
          {
   S10:      sum=a+b+sum;
   S10.1     I=25;
   S10.2     print(I);
          }
   S11:endif
   S12:print(sum);
End
```

Relevant Slice

- The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case, is known as the **relevant slice** of the program.
- For example, consider the example given in previous figure.

Relevant Slice cont …

- Statements S3 and S6 have the potential to affect the output, if modified.
- On the basis of relevant slices, we can prioritize the test cases.
- This technique is helpful for prioritizing the regression test suite which saves time and effort for regression testing.

Prioritization Based on Requirements

- This technique is used for prioritization of the system test cases.
- The system test cases also become too large in number, as this testing is performed on many grounds. Since system test cases are largely dependent on the requirements, the requirements can be analysed to prioritize the test cases.
- This technique does not consider all the requirements on the same level. Some requirements are more important and critical as compared to others, and these test cases having more weight are executed earlier.

PORT

- Hema srikanth et al. have proposed a requirements based test case prioritization technique.
- It is known as PORT (prioritization of requirements for test).
- They have taken the following four factors for analyzing and measuring the criticality of requirements.

Factors for analyzing & measuring criticality of requirements.

- **Customer-assigned priority of requirements**: Based on priority, the customer assigns a weight (on scale of 1 to 10) to each requirement.
- **Requirement volatility:** This is a rating based on the frequency of change of a requirement.
- **Developer-perceived implementation complexity:** The developer gives more weight to a requirement which he thinks is more difficult to implement.
- **Fault proneness of requirements:** This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for a new software.

Prioritization Factor Value

- Based on these four factor values, a prioritization factor value (PFV) is computed as given below.

$$PFV_i = \sum(FV_{ij} \times FW_j)$$

where FV_{ij} = Factor value which is the value of factor j corresponding to requirement i, and FW_j = Factor weight which is the weight given to factor j.

- PFV is then used to produce a prioritized list of system test cases.

Measuring Effectiveness of a prioritized test suite

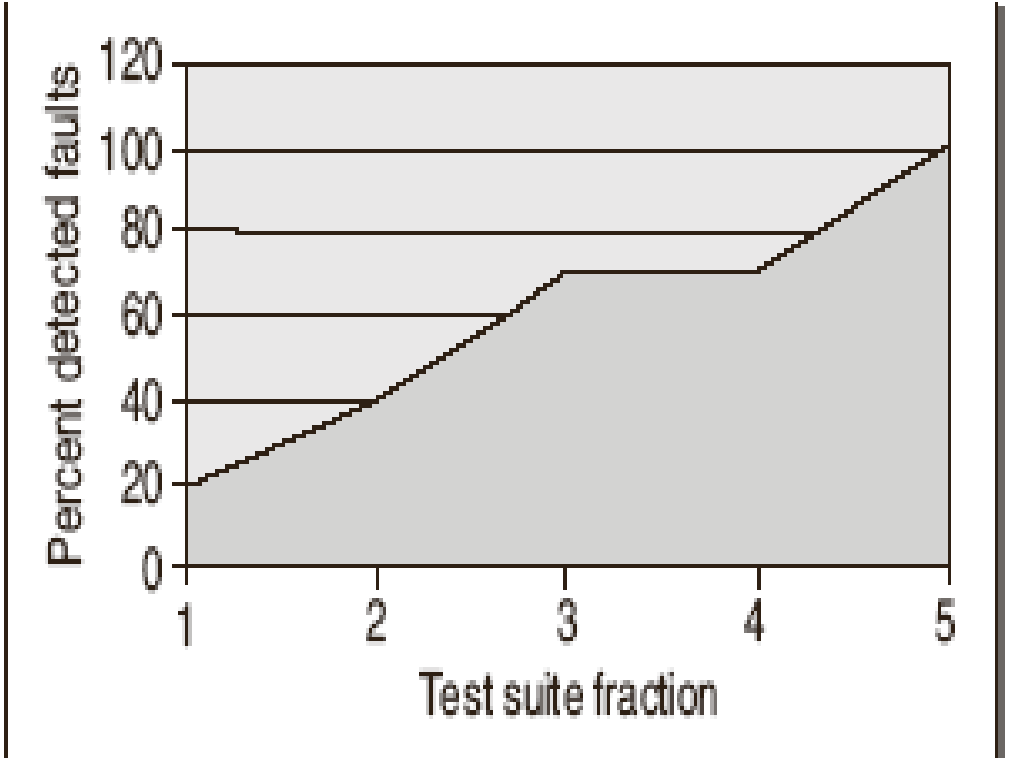
- Elbaum et al. developed APFD (average percentage of faults detection) metric that measures the weighted average of percentage of faults detected during the execution of a test suite.
- Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate.

APFD Metric

APFD is a metric to detect how quickly a test suite identifies the faults. It is defined as follows:

$$APFD = 1 - ((TF1+TF2+\ldots\ldots+TFm)/nm) + 1/2n$$

where TFi is the position of the first test in test suite T that exposes fault i, m is the total number of faults exposed in the system or module under T and n is the total number of test cases in T.


Percent detected faults
120
100
80
60
40
20
0
1
2
3
4
5
Test suite fraction

Example

Consider a program with 10 faults & test suite of 10 test cases, as shown in below table.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1					x			X		
F2		X	X	X		X				
F3	X	X	X	x			x	X		
F4						X				x
F5	X		X		X	X		x		X
F6					X	x	x		X	
F7									X	
F8		x		X		x			X	X
F9	X									X
F10			X	x					X	X

- Let us consider the order of test suite as (T1, T2, T3, T4, T5, T6, T7, T8, T9, T10).
- Then calculate the APFD for this program as follows:

$$APFD = 1 - (5+2+1+6+1+5+9+2+1+3)/10*10) + \frac{1}{2\times10}$$

$$=0.65 + 0.05$$

$$=0.7$$

- All the bugs detected are not of the same level of severity.
- One bug may be more critical as compared to others.
- Moreover, the cost of executing the test cases also differs.
- One test case may take more time as compared to others.
- Thus, APFD does not consider the severity level of the bugs and the cost of executing the test cases in a test suite.

Cost-cognizant APFD

- Elbaum *et al.* modified their APFD metric and considered these two factors to form a new metric which is known as *cost-cognizant* APFD and denoted as APFDc.
- In APFDc, the total cost incurred in all the test cases is represented on x-axis and the total fault severity detected is taken on y-axis.
- Thus, it measures the unit of fault severity detected per unit test case cost.

Summary

- Discussed in detail the following types of test case prioritization techniques.
  - Risk-Based Prioritization
  - Prioritization Based on Operational Profiles
  - Prioritization using Relevant Slices
  - Prioritization Based on Requirements
- Discussed a metric, called APFD, to measure the effectiveness of the prioritization technique.
- Also, discussed a variance of APFD, called *cost-cognizant* APFD, which is denoted as APFDc.

References

1. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 12), Second Edition, Oxford University Press, 2016.

Thank you

Efficient Test Suite Management cont …

Prof. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela

Overview of last class

- Discussed a technique for prioritizing system test cases based on requirements, proposed by **Srikanth et. al.**
- This technique is known as PORT (prioritization of requirements for test).
- They have considered the following four factors for analyzing and measuring the criticality of requirements.

Overview of last class cont …

- **Customer-assigned priority of requirements**: Based on priority, the customer assigns a weight (on scale of 1 to 10) to each requirement.
- **Requirement volatility:** This is a rating based on the frequency of change of a requirement.
- **Developer-perceived implementation complexity:** The developer gives more weight to a requirement which he thinks is more difficult to implement.
- **Fault proneness of requirements:** This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone , then this requirement in the current version is given more weight. This factor cannot be considered for a new software.

Overview of last class cont …

- Based on these four factor values, a prioritization factor value (PFV) is computed as given below.

$$PFV_i = \sum(FV_{ij} \times FW_j)$$

where FVij = Factor value is the value of factor j corresponding to requirement I, and FWj = Factor weight is the weight given to factor j.

- PFV is then used to produce a prioritized list of system test cases.

An Approach for Prioritization of Regression Test Cases

Kavitha & Kumar proposed an approach for prioritization of regression test cases by considering the following factors:

1. Customer-assigned priority of requirements
2. Changes in requirements
3. Developer-perceived code implementation complexity
4. Fault impact of requirements
5. Completeness
6. Traceability
7. Execution time

Based on these factors, a weightage is assigned to each test case and the test cases are prioritized according to the weightages.

Another Approach for Test Case Prioritization Based on Requirements

Kumar & Chauhan proposed a hierarchical test case prioritization approach, where prioritization is performed at three levels:

- Prioritize the requirements on the basis of 12 factors by assigning a weightage to each requirement.
- Map the prioritized requirements to their corresponding modules to get the prioritized modules.
- Rank (prioritize) the test cases of the prioritized modules for execution.

Prioritization of requirements

- The process of prioritization of requirements is performed on the basis of 12 factors.
- These factors are in accordance with every phase of SDLC.
- All these factors have been assigned a priority value between 0 and 10.
- These priority values are assigned by various stakeholders of the project.
- Table in next slide shows these factors.

Factors considered for requirement prioritization

S.NO	Factors	Phase of SDLC	Priority Value Assigned by
1	Requirement volatility	Requirement analysis	Customer
2	Customer assigned priority	Requirement analysis	Customer
3	Implementation complexity	Design	Developer
4	Fault proneness of requirements	Design	Developer
5	Developer assigned priority	Requirement analysis	Developer
6	Show stopper requirements	Design	Developer
7	Frequency of execution of requirements	Requirement analysis	Developer
8	Expected faults	Coding	Developer
9	Cost	Requirement analysis	Analyst
10	Time	Requirement analysis	Analyst
11	Penalty	Requirement analysis	Customer
12	Traceability	Testing	Tester

Requirement Prioritization Factor Value

- For each requirement, based on these 12 factors, a Requirement Prioritization Factor Value (RPFV) is calculated using below Eq.

$$RPFV = \sum_{j=1}^{n} (pfvalue_{ij} X pfweight_j)$$

- i represents the number of requirements, & j represents the number of factors
- RPFV represents the prioritization factor value for a requirement which is the summation of the product of priority value of a factor and the project factor weight.

Requirement Prioritization Factor Value cont …

- pfvalue is the value of factor for the i^{th} requirement
- pfweight is the factor weight for the j^{th} factor for a particular project.
- By using the previous Eq. the weight prioritization factor RPFV for every requirement can be calculated.
- The table in next slide shows the prioritization of 4 sample requirements on the basis of RPFV for each requirement.

Requirements prioritization

Factor	R1	R2	R3	R4	Weight factor
Customer assigned priority	8	10	9	9	0.02
Developer assigned priority	8	9	9	8	0.08
Requirements volatility	3	0	0	2	0.1
Fault proneness	0	0	0	0	0.15
Expected faults	2	3	4	2	0.10
Implementation complexity	3	4	5	3	0.10
Execution frequency	5	10	9	6	0.05
Traceability	0	0	0	0	0.05
Show stopper requirements	0	9	6	0	0.2
Penalty	1	4	3	3	0.05
Time	3	6	5	4	0.05
Cost	4	7	6	6	0.05
RPFV	2.25	4.77	4.15	2.47	1.0

Inference

- In this table, R2 has the highest RPFV among all requirements.
- So, the prioritization order of these requirements is R2, R3, R4, and R1.
- The value of RPFV depends on the values of pfvalue and pfweight.

Prioritization of modules

- It is a process of mapping between prioritized requirements and their corresponding modules.
- If there is more than one module then the modules are prioritized.
- It uses criteria like Cyclometic complexity and non-dc path.

Prioritization of modules cont …

- Definition-clear path (dc-path): A dc-path with respect to a variable v is a path between the definition node and the usage node s.t. no other node in the path is a defining node of variable v. Non-dc paths are more error prone.
- The test cases of higher priority modules are prioritized first and executed.
- For each module, a module prioritize value (MPV) is calculated by adding cyclometic complexity and the number of non-dc paths.

Example

Factors	M1	M2	M3	M4
Cyclomatic complexity	8	4	4	5
Non-dc path	7	5	6	3
MPV	15	9	10	8

Example

- The table shows the prioritization of 4 sample modules on the basis of MPV for each module.
- The order of prioritization of modules on the basis of MPV is M1, M3, M2 and M4.

Test case prioritization process

- It is used to prioritize and schedule the test cases corresponding to prioritized modules.
- Some weight factors are used for test case prioritization such as
  - test case complexity,
  - requirements coverage,
  - dependency of the test cases and
  - test impact.

Test Case Complexity

- It shows how difficult it is to execute a test case.
- It also shows how much effort is required to execute the test case.
- This factor is assigned a value between 1 and 10.

Requirements Coverage

- It shows how many requirements are covered by executing the test case.
- This factor is scaled between 1 to 10.
- A higher value shows maximum requirements are covered by the test case.
- Higher the number of requirements coverage, higher is the priority of the test case to be executed first.

Dependency

- It shows the dependency of test cases on some pre-requisities.
- It shows how many pre-requisites are required for each test case before execution of the test case.
- This factor is assigned a value between 1 and 10.

Test Impact

- It is the most critical factor in test case prioritization.
- It shows the impact of test case on a system if it is not executed.
- This factor assesses importance of the test case.
- This factor is assigned a value between 1 and 10.

Test Case Weight Prioritization

- TCWP is calculated as follows

$$\text{TCWP} = \sum_{j=1}^{n} (fvalue\ ij \times fweight\ j)$$

where,

- TCWP = weight prioritization for each test case calculated from the four factors.
- fvalue = value assigned to each test case.
- fweight = weight assigned to each factor.

Test Case Weight Prioritization cont …

- After calculating the value of each test case, test cases are ordered by TCWP such that maximum TCWP gives a test case the highest priority and executed.
- Suppose, a set of four test cases TC1, TC2, TC3, and TC4 are to be prioritized. For these test cases TCWP is calculated by using the above equation and are prioritized on the basis of values of TCWP.

Example

S. No.	Factor	TC1	TC2	TC3	TC4	Weight
1	Test impact	4	8	7	9	0.4
2	Test case complexity	8	7	5	9	0.3
3	Requirement coverage	6	2	4	4	0.2
4	Dependency	7	6	6	8	0.1
	TCWP	5.90	6.30	5.70	7.90	1.0

So, the final prioritized order of test cases is TC4, TC2, TC1, TC3

Reference

1. Naresh Chauhan, Software Testing: Principles and Practices, Second Edition, (Chapter 12), Oxford Univerity Press, 2018.

Thank you

Efficient Test Suite Management cont…

Dr. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela

Data Flow Based Test Case Prioritization

- This is a white box testing technique used to detect improper use of data values due to coding errors.
- Data usage for a variable affects the white box testing and thereby the regression testing.
- If the prioritization of regression test suite is based on this concept, the rate of detection of faults will be high and critical bugs can be detected earlier.

Data Flow Based Test Case Prioritization cont…

J.Rummel et al. proposed an approach to regression test prioritization that leverages the all-du's (definition-use) test adequacy criterion that focuses on the definition and use of variables within the program under test.

Data Flow Based Test Case Prioritization cont…

- Kumar et al. proposed an approach for test case prioritization using *du* path as well as definition clear (dc) paths.

  Idea: *du* paths which may not be *dc*, may be very problematic as non-dc paths may be subtle source of errors.

Data Flow Based Test Case Prioritization cont…

- In another work, it was indicated that for testing modified programs, the prioritized test suite for original program may not work because after modification, new *du* paths may get introduced and some of these *du* paths may also not be definition clear (dc).
- These paths may also be more prone to errors.

Data Flow Based Test Case Prioritization cont…

- So, a list of all such newly introduced non-dc paths is prepared and test cases corresponding to these paths at the highest priority are placed in the test suite for a modified program. This set of test cases is referred to as 'Set-1'.
- Because of modification in the program, some existing *dc* paths may become *non-dc*. The set of test cases corresponding to these paths is taken at the next priority and this set of test cases may be referred as 'Set-2'.

Data Flow Based Test Case Prioritization cont…

- Further, there may be some *non-dc* paths of the original program, which are still *non-dc* after modification of the program. The set of test cases for such paths may be referred to as 'Set-3'.
- Finally, there are some test cases which do not cover non-dc paths, i.e., these test cases cover dc-paths. This may be referred to as 'Set-4'.
- Using these 4 concepts (sets), an algorithm has been designed for prioritizing the test suite.

Algorithm for prioritizing the test suite

Step 1:

If a test case covers more number of paths in Set-1

Then place it at the highest priority.

Step 2:

If 2 test cases (in Step1)cover an equal number of paths in Set-1

Then the test case which covers more number of paths in Set-2 in the modified program is placed at the next priority.

Step 3:

If 2 test cases (in Step 2) cover equal number of paths in Set-2 in the modified program,

Then the test case which covers maximum number of lines of code in the modified program is placed at the next priority.

Module Coupling Slice-Based Test Case Prioritization

- This technique is based on dependence and coupling between the modules, and proceeds in 2 steps.

Steps for Module Coupling Slice-Based Test Case Prioritization

Step 1: Find the highly affected module whenever there is a change in a module.

Step 2: Prioritize the test cases of the affected module identified in Step 1.

- In both steps, coupling information between modules is considered.

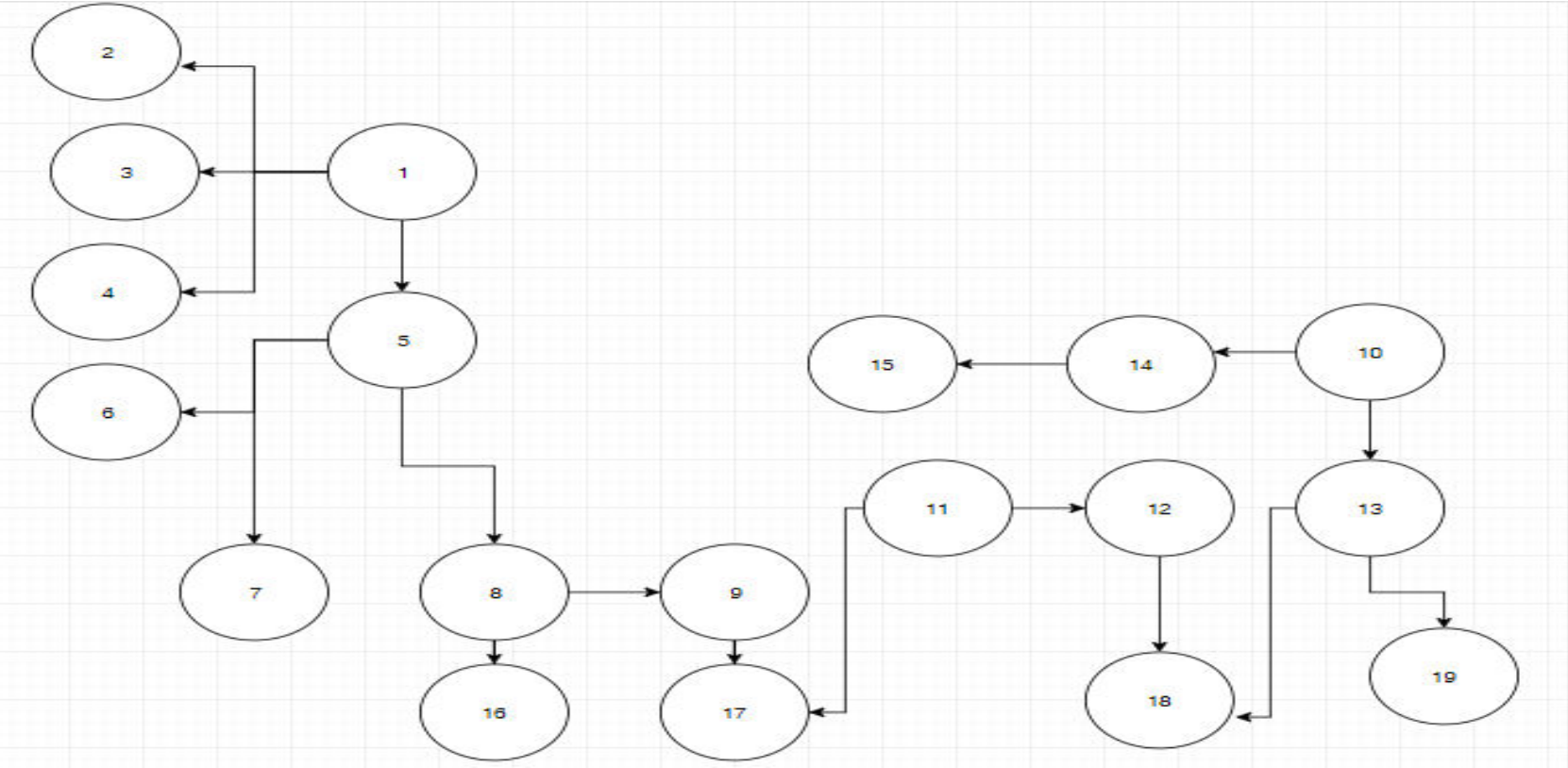
Module Coupling Slice-Based Test Case Prioritization cont…

- When there is a change in a module, there will be some effect on other modules which are coupled with this module.
- Based on the coupling information between the modules, the highly affected module can be identified.

Module-Coupling Effect

- The effect is worse if there is high coupling between the modules causing high probability of errors. This is called as *Module-Coupling Effect*.
- If regression test case prioritization is done based on this module coupling effect, there will be high percentage of detecting **critical errors** that have been propagated to other modules due to any change in a module.

Example Call Graph



Module-Coupling Effect cont…

- Modules 17 and 18 are being called by multiple modules.
- If there is any change in these modules 17 & 18, then modules 9,11and 12,13 will be affected respectively.
- If there is no prioritization, then according to regression testing, all test cases of all the affected modules will be executed, thus increasing the testing time and effort.
- If the coupling type between modules is known, then a prioritization scheme can be developed based on this coupling information.

Module Coupling Slice-Based Test Case Prioritization cont…

- Modules having worst type of coupling will be prioritized over other modules and their test cases.
- Module dependence can be identified by coupling and cohesion.
- A quantitative measure of the dependence of modules will be useful to find out the stability of the design.
- The work is based on the premise that different values can be assigned for various types of module coupling and cohesion.

Module Coupling Slice-Based Test Case Prioritization cont…

Table 1: Coupling types and their values

Coupling Types	Value
Content	0.95
Common	0.70
External	0.60
Control	0.50
Stamp	0.35
Data	0.20

Module Coupling Slice-Based Test Case Prioritization cont…

Table 2: Cohesion types and their values

Cohesion Types	Value
Coincidental	0.95
Logical	0.40
Temporal	0.60
Procedural	0.40
Communicational	0.25
Sequential	0.20
Functional	0.20

Module Coupling Slice-Based Test Case Prioritization cont…

- A matrix can be obtained by using the 2 tables, which gives the dependence among all the modules in a program.
- This dependence matrix describes the probability of having to change module i, given that module j has been changed.
- Module Dependence Matrix is derived using the following 3 steps:

Steps for Module Dependence Matrix

Step 1:

(i) Determine the coupling among all the modules in the program.

(ii) Construct an mxm coupling matrix, where m is the number of modules. Using Table 1, fill the elements of matrix. Element $\mathbf{C_{ij}}$ represents coupling between modules i and j.

(iii) This matrix is symmetric, i.e. $\mathbf{C_{ij} = C_{ji}}$ for all i and j.

(iv) Elements on the diagonal are all 1 (C_{ij} = 1for all i).

Steps for Module Dependence Matrix cont…

- **Step 2:**

(i) Determine strength of each module in the program.

(ii) Using Table 2, record the corresponding numerical values of cohesion in module cohesion matrix (**S**).

Steps for Module Dependence Matrix cont…

- **Step 3:**

(i) Construct the Module Dependence Matrix **D** using the following equation:

$D_{ij}=0.15\ (S_i + S_j)+ 0.7\ C_{ij}$, where C_{ij} is not equal to 0,

$= 0$, where $C_{ij} = 0$ $D_{ii} = 1$ for all i.

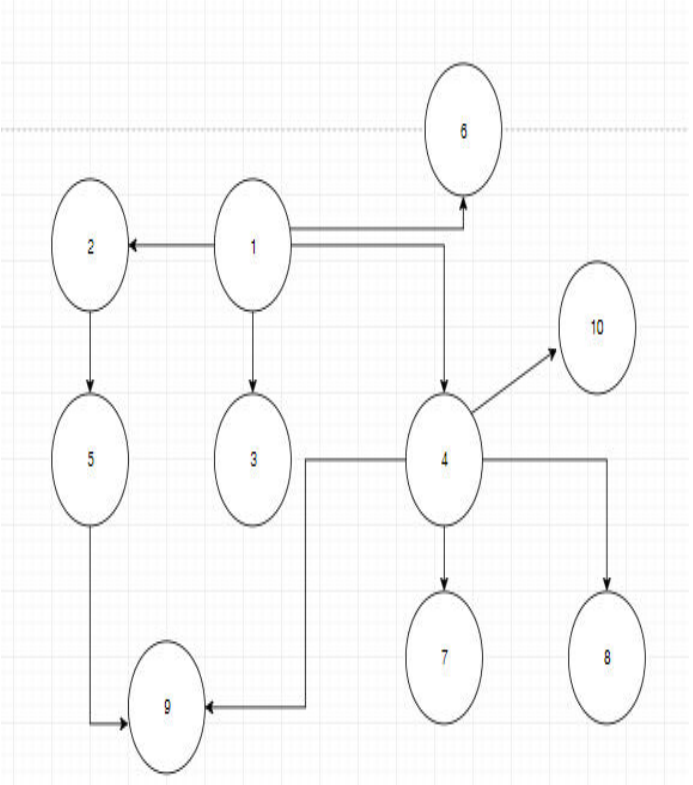
Prioritization of modules can de done by comparing non zero entries of matrix **D**.

Steps for Module Dependence Matrix cont…

- For example, if module i has been modified, find all the existing parent modules(j,k,l,…) of that changed module(i) and after that compare first-order dependence matrix entries for particular links, like i-j, i-k,i-l and so on.
- The link having the highest module dependence matrix value will get the highest priority and the link with the lowest module dependence matrix value will get the lowest priority.

Example

A Software consists of 10 modules. The coupling and cohesion information of these modules are given in tables shown in next slide. Let us find out the badly affected module in this software.



Call Graph

Example

Coupling information

Type of Coupling	No. of Modules in Relation	Examples
Data Coupling	3	1-2,1-4,1-6
Stamp Coupling	1	1-3
Control Coupling	4	4-7,4-8, 4-9,4-10
Common Coupling	2	2-5,5-9
Message (Content) Coupling	0	-

cont…

Cohesion information

Module Number	Cohesion Type
1	Coincidental
2	Functional
3	Communicational
4	Logical
5	Procedural
6	Functional
7	Functional
8	Functional
9	Functional
10	Functional

Example cont…

By using coupling values among different modules, a module coupling matrix is given below:

1.0	0.2	0.35	0.2	0.0	0.2	0.0	0.0	0.0	0.0
0.2	1.0	0.0	0.0	0.70	0.0	0.0	0.0	0.0	0.0
0.35	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.2	0.0	0.0	1.0	0.0	0.0	0.50	0.50	0.50	0.50
0.0	0.2	0.0	0.0	1.0	0.0	0.0	0.0	0.70	0.0
0.2	0.2	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.2	0.50	0.0	0.0	1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.50	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.50	0.70	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.50	0.0	0.0	0.0	0.0	0.0	1.0

Example cont…

By using the value of cohesion among different modules a Cohesion Matrix (**S**) is designed as shown below:

0.95	0.2	0.25	0.4	0.4	0.2	0.2	0.2	0.2	0.2

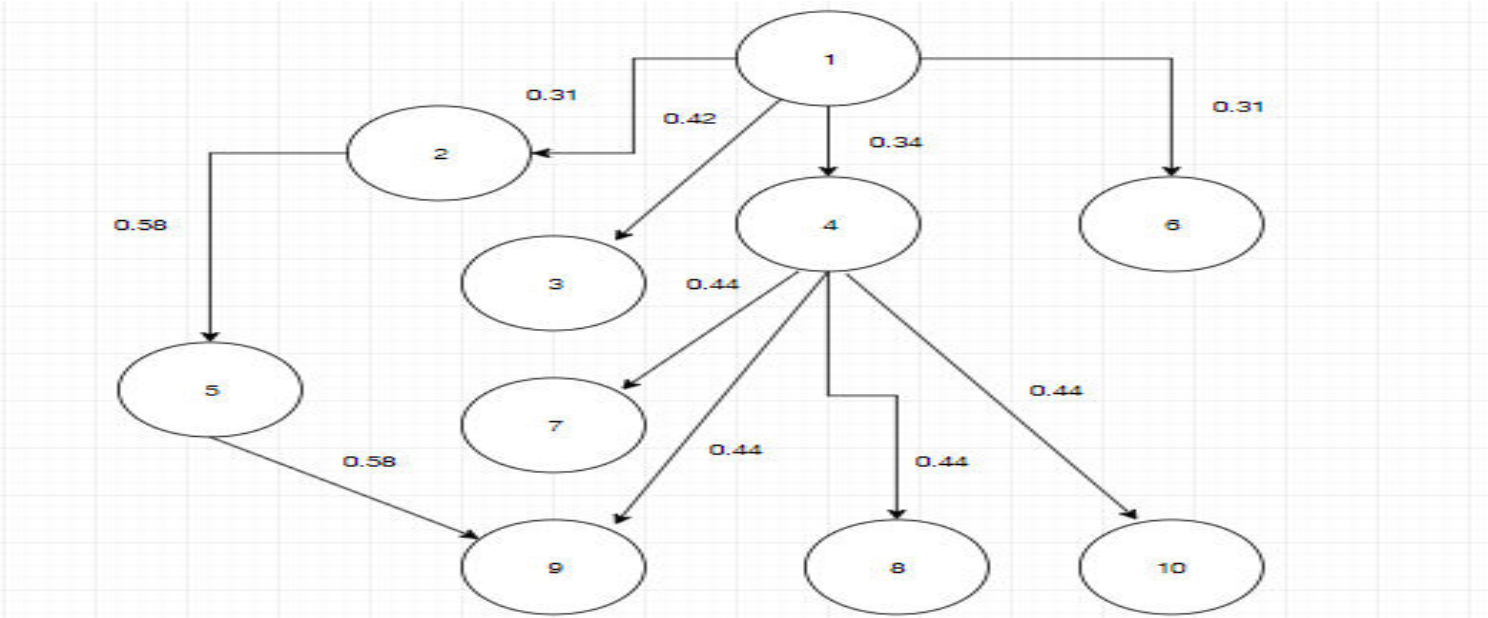
Example cont…

A Module Dependence Matrix (**D**)is designed, (using the previous equation).

1.0	0.31	0.42	0.34	0.0	0.31	0.0	0.0	0.0	0.0
0.31	1.0	0.0	0.0	0.58	0.0	0.0	0.0	0.0	0.0
0.42	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.34	0.0	0.0	1.0	0.0	0.0	0.44	0.44	0.44	0.44
0.0	0.58	0.0	0.0	1.0	0.0	0.0	0.0	0.58	0.0
0.31	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.0	0.44	0.58	0.0	0.0	0.0	1.0	0.0
0.0	0.0	0.0	0.44	0.0	0.0	0.0	0.0	0.0	1.0

Example cont…

From the module dependence values obtained from call graph as shown below and from Module Dependence matrix, we conclude that change in module 4 propagates to modules 7,8,9 and 10.



Example cont…

- Modules 7,8,9 and 10 have the same module dependence value(0.44), so the order of prioritization of test cases for these modules is the same.
- Similarly, the change in module 1 propagates to modules 2,3,4 and 6.
- The module dependence values for these modules show that module 3 is the more affected module when compared to modules 2,4 and 6.
- So, the test cases for module 3 have to be prioritized first as compared to modules 2,4 and 6.

Module Coupling Slice-Based Test Case Prioritization cont…

- After identifying the highly affected module due to a change in a module, there is a need to execute the test cases corresponding to this affected module.
- However, there may be a large number of test cases in this module.
- Therefore, there is a need to prioritize these test cases so that critical bugs can be found easily.
- Based on the coupling information between the changed module and the affected module, a **coupling slice** can be prepared that helps in prioritizing the test cases.

Module Coupling Slice-Based Test Case Prioritization cont…

- All the statement numbers present in the execution history of a program comprise the execution slice of a program.
- Coupling information can be helpful to decide which variables are affected in the caller module.
- Depending upon this information the statement numbers of affected variables can be found. This may be called a **coupling slice**.

Module Coupling Slice-Based Test Case Prioritization cont…

- At last, these statement numbers are matched in the execution slice.
- The test cases for these statements will be given high priority and executed first.

THANK YOU