Software Testing & Audit

Unit-3

Jayash Kumar Sharma
Department of Computer Science & Engineering
Hindustan Institute of Technology & Management
Mob: +91-9639325975
Email: jayash.sharma@gmail.com

Software Maintenance

- Software certainly changes, whatever well-written and designed initially it may be.

Errors	Additional Functionality	External policies and principles
Improvement in efficiency and performance	Change in existing technology	To delete some obsolete capabilities

Software Maintenance

- Software always changes in order to address the above mentioned issues.
- Changed software is required to be re-tested in order to ensure that changes work correctly and these changes have not adversely affected other parts of the software.
- This is necessary because small changes in one part of the software program may have subtle undesired effects in other seemingly unrelated parts of the software.

Regression Testing

- Regression testing is the process of re-testing the modified parts of the software.
- Ensures that no new errors have been introduced into previously tested source code due to these modifications.
- Therefore, regression testing tests both the modified source code and other parts of the source code that may be affected by the change.

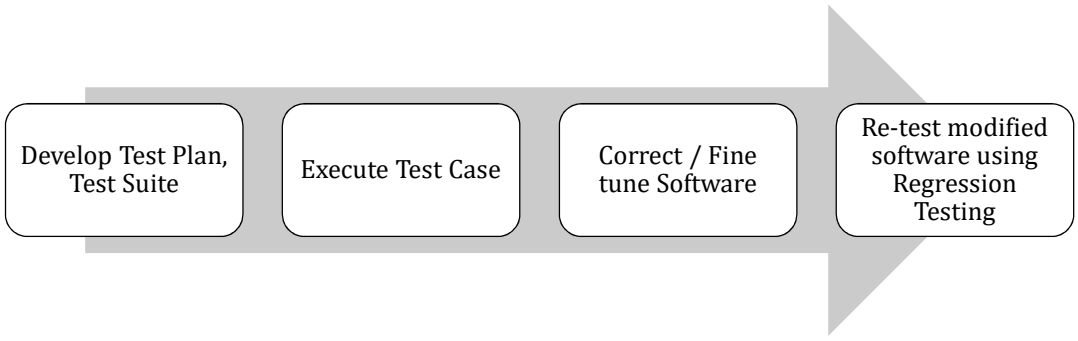
Regression Testing

- Regression testing serves several purposes like:

1. Increases confidence in the correctness of the modified program.
2. Locates errors in the modified program.
3. Preserves the quality and reliability of the software.
4. Ensures the software's continued operation.

Regression Testing

- Regression testing is thought as a software maintenance activity.
- But regression testing is also performed during the latter stage of software development.



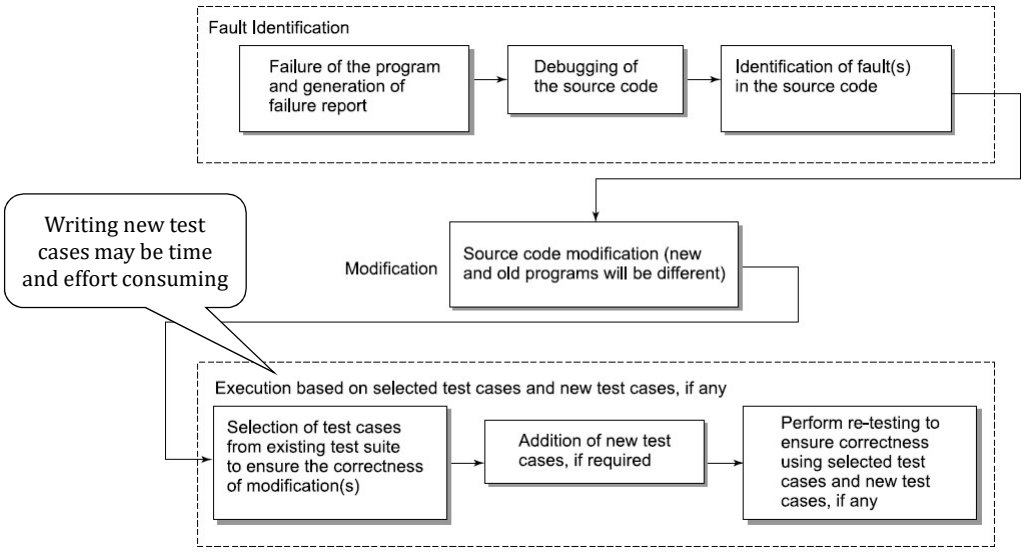
Development vs Regression Testing

Sr. No.	Development Testing	Regression Testing
1.	We write test cases.	We may use already available test cases.
2.	We want to test all portions of the source code.	We want to test only modified portion of the source code and the portion affected by the modifications.
3.	We do development testing just once in the lifetime of the software.	We may have to do regression testing many times in the lifetime of the software.
4.	We do development testing to obtain confidence about the correctness of the software	We do regression testing to obtain confidence about the correctness of the modified portion of the software.

Development vs Regression Testing

Sr. No.	Development Testing	Regression Testing
5.	Performed under the pressure of release date.	Performed in crisis situations, under greater time constraints.
6.	Separate allocation of budget and time.	Practically no time and generally no separate budget allocation.
7.	Focus is on the whole software with the objective of finding faults.	Focus is only on the modified portion and other affected portions with the objective of ensuring the correctness of the modification.
8.	Time and effort consuming activity (40% to 70%).	Not much time and effort is consumed as compared to development testing.

Regression Testing Process



Regression Testing Process

- Size of the existing test suite may be very large and it may not be possible to execute all tests.
- The greatest challenge is to reduce the size of the existing test suite for a particular failure.
- Hence, test case selection for a failure is the main key for Regression Testing.

Selection of Test Cases

- How to select an appropriate number of test cases for a failure?
- The range is from “one test case” to “all test cases”.
- A ‘regression test cases’ selection technique may help us to do this selection process.
- The effectiveness of the selection technique may decide the selection of the most appropriate test cases from the test suite.

Selection of Test Cases

Source Code

```
1.  main( )                          1.  main ( )
2.  {                                2.  {
3.  int a, b, x, y, z;               3.  int a, b, x, y, z;
4.  scanf (“%d, %d”, &a, &b);        4.  scanf (“%d, %d”, &a, &b);
5.  x = a + b ;                      5.  x = a + b;
6.  y = a* b;                        6.  y = a – b;
7.  if (x ≥ y) {                     7.  if (x ≥ y) {
8.  z = x / y ;                      8.  z = x / y ;
9.  }                                9.  }
10. else {                           10. else {
11. z = x * y ;                      11. z = x * y ;
12. }                                12. }
13. printf (“z = %d \ n”, z );       13. printf (“z = %d \ n“, z);
14. }                                14. }
```

(a) Original program with fault in line 6.

(b) Modified program with modification in line 6.

Modified Line
* is replaced by -

Selection of Test Cases

Test Suite

Set of Test Cases S. No.	Inputs a	Inputs b	Execution History
1	2	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
2	1	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
3	3	2	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14
4	3	3	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14

Selection of Test Cases

- In this example, line number 6 is modified where 'a*b' is replaced by 'a-b'.
- All four test cases of the test suite execute this modified line 6.
- We may decide to execute all four tests for the modified program.
- If we do so, test case 2 with inputs a = 1 and b = 1 will experience a ***'divide by zero'*** problem, whereas others will not.

Selection of Test Cases

- We may select all test cases which are executing the modified line.
- Here, all four test cases are executing the modified line (line number 6) and hence are selected.
- There is no reduction in terms of the number of test cases.
- However, we may like to reduce the number of test cases for the modified program.

Selection of Test Cases

- Check execution history :
  - Test case 1 and Test case 2 have the same execution history.
  - Test case 3 and Test case 4 have the same execution history.
- Choose any one test case of the same execution history to avoid repetition.
- For execution history 1 (i.e. 1, 2, 3, 4, 5, 6, 7, 8, 10, 11), if we select test case 1, the program will execute well, but if we select test case 2, the program will experience 'divide by zero' problem.
- Therefore, either test case 1 or test case 2 may have to be selected.

Selection of Test Cases

- Select test case 1, will miss the opportunity to detect the fault that test case 2 detects.
- Minimization techniques may omit some test cases that might expose fault(s) in the modified program. (Selection of test case-1 in this example)
- Hence, we should be very careful in the process of minimization of test cases and always try to use safe regression test selection technique (if at all it is possible).
- A Safe Regression Test Selection Technique should select all test cases that can expose faults in the modified program.

Regression Test Case Selection

- Select All Test Cases
- Select Test Cases Randomly
- Select Modification Traversing Test Cases

Regression Test Case Selection

Select All Test Cases

- Simplest, safest technique without any risk.
- Run all test cases for any change in the program.
- A program may fail many times and every time we will execute the entire test suite.
- This technique is practical only when the size of the test suite is small. For any reasonable or large sized test suite, it becomes impractical to execute all test cases.

Regression Test Case Selection

Select Test Cases Randomly

- Test cases may be selected randomly to reduce the size of the test suite.
- How many test cases are required to be selected depend upon time and available resources.
- Once, we decide the number, the same number of test cases is selected randomly.

Regression Test Case Selection

Select Test Cases Randomly

- If the number is large, we may get a good number of test cases for execution and testing may be of some use.
- If the number is small, testing may not be useful at all.
- In this technique, our assumption is that all test cases are equally good in their fault detection ability.
- However, in most of the situations, this assumption may not be true.

Regression Test Case Selection

Select Test Cases Randomly

- Objective is to re-test the source code for the purpose of checking the correctness of the modified portion of the program.
- Many randomly selected test cases may not have any relationship with the modified portion of the program.
- However, random selection may be better than no regression testing at all.

Regression Test Case Selection

Select Modification Traversing Test Cases

- Only those test cases are selected that execute the modified portion of the program and the portion which is affected by the modification(s).
- Other test cases of the test suite are discarded.
- All those test cases that reveal faults in the modified program are known as Fault Revealing Test Cases.

Regression Test Case Selection

Select Modification Traversing Test Cases

- No effective technique to find fault revealing test cases for the modified program.
- This is the best selection approach, which we want, but we do not have techniques for the same.
- Another lower objective may be to select those test cases that reveal the difference in the output of the original program and the modified program.

Regression Test Case Selection

Select Modification Traversing Test Cases

- Such test cases are known as Modification Revealing Test Cases.
- These test cases target that portion of the source code which makes the output of the original program and the modified program differ.
- Unfortunately, we do not have any effective technique to do this.
- Therefore, it is difficult to find fault revealing test cases and modification revealing test cases.

Regression Test Case Selection

Select Modification Traversing Test Cases

- The reasonable objective is to select all those test cases that traverse the modified source code and the source code affected by modification(s).
- These test cases are known as Modification Traversing Test Cases.
- It is easy to develop techniques for modification traversing test cases and some are available too.

Regression Test Case Selection

Select Modification Traversing Test Cases

- Out of all modification traversing test cases, some may be modification revealing test cases and out of some modification revealing test cases, some may be fault revealing test cases.
- Many modification traversing techniques are available but their applications are limited due to the varied nature of software projects.

Reducing the Number of Test Cases

- Test case reduction is an essential activity in Regression Testing.
- Select those test cases that execute the modification(s) and the portion of the program that is affected by the modification(s).
- We may minimize the test suite or prioritize the test suite in order to execute the selected number of test cases.

Reducing the Number of Test Cases

Minimization of Test Cases

- Select all those test cases that traverse the modified portion of the program and the portion that is affected by the modification(s).
- If selected number is very large, reduce this using any test case minimization technique.
- These test case minimization techniques attempt to find redundant test cases.

Reducing the Number of Test Cases

Minimization of Test Cases

- Redundant Test Case : Which achieves an objective which has already been achieved by another test case.
- The objective may be:
  - Source code coverage
  - Requirement coverage
  - Variables coverage
  - Branch coverage
  - Specific lines of source code coverage, etc.

Reducing the Number of Test Cases

Minimization of Test Cases

- A minimization technique may further reduce the size of the selected test cases based on some criteria.
- Any type of minimization is risky and may omit some fault revealing test cases.

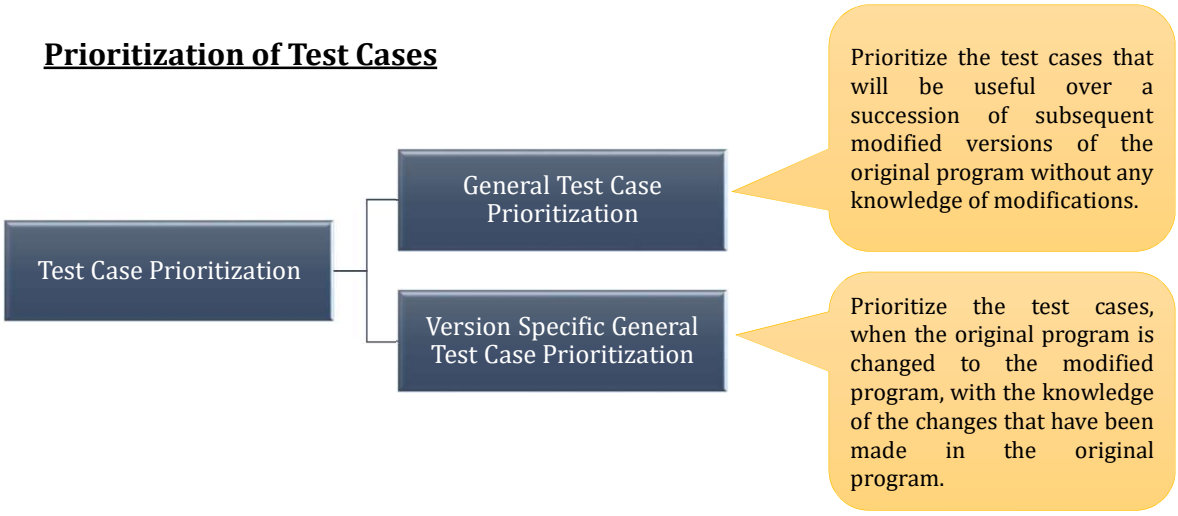
Reducing the Number of Test Cases

Prioritization of Test Cases

- Ordering of the test cases with which a test case may be addressed is known as prioritization of test cases.
- A test case with the highest rank has the highest priority and the test case with the second highest rank has the second highest priority and as so on.
- Prioritization does not discard any test case.
- The efficiency of the regression testing is dependent upon the criteria of prioritization.

Reducing the Number of Test Cases

Prioritization of Test Cases



Reducing the Number of Test Cases

Prioritization of Test Cases

Prioritization guidelines should address two fundamental issues:

1. What functions of the software must be tested?
2. What are the consequences if some functions are not tested?

- Every reduction activity has an associated risk.
- All prioritization guidelines should be designed on the basis of risk analysis.

Reducing the Number of Test Cases

Prioritization of Test Cases

- All risky functions should be tested on higher priority.
- The risk analysis may be based on complexity, criticality, impact of failure, etc.
- The most important is the 'impact of failure' which may range from 'no impact' to 'loss of human life' and must be handled very carefully.

Reducing the Number of Test Cases

Prioritization of Test Cases – Scheme-1

- Assign a priority code to every test case. (Assumption: priority code 1 is more important than test case of priority code 2).

Priority code 1	Essential test case
Priority code 2	Important test case
Priority code 3	Execute, if time permits
Priority code 4	Not important test case
Priority code 5	Redundant test case

Reducing the Number of Test Cases

Prioritization of Test Cases – Scheme-2

- Assign priorities based on customer requirements or market conditions.

Priority code 1	Important for the customer
Priority code 2	Required to increase customer satisfaction
Priority code 3	Help to increase market share of the product

Reducing the Number of Test Cases

Prioritization of Test Cases - Conclusion

- We may design any priority category scheme.
- A scheme based on technical considerations always improves the quality of the product and should always be encouraged.

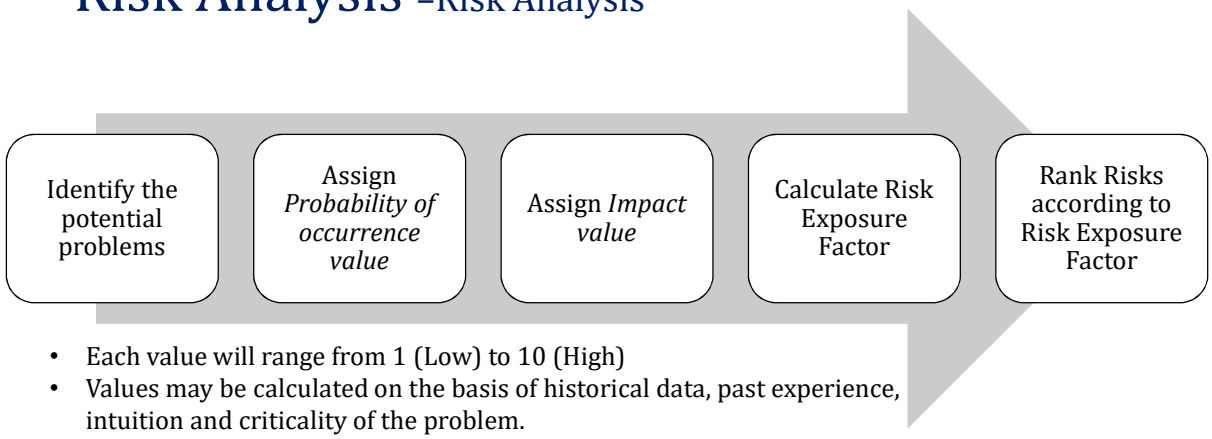
Risk Analysis

- Unexpected behaviors of a software program: Disturbing and embarrassing situation for companies and associated people.
- Developers work hard to find the solutions of the problems highlighted by these unexpected behaviors.
- These situations can be minimized if risky areas of the software minimized.
- Hence, risk analysis must be carefully conducted to minimize the risk.

Risk Analysis – What is Risk

- A problem that may cause some loss or threaten the success of the project, but, which has not happened yet is referred as Risk.
- Risk is defined as the "Probability of occurrence of an undesirable event and the impact of occurrence of that event".
- In order to consider an event as Risk, potential consequences of the occurrences / non-occurrences of that event must be understood.
- Risks may delay, over-budget a project. Risky projects may also not meet specified quality levels.

Risk Analysis –Risk Analysis



- Each value will range from 1 (Low) to 10 (High)
- Values may be calculated on the basis of historical data, past experience, intuition and criticality of the problem.

Risk Exposure Factor of the problem =
'Probability of occurrence of the problem' value * 'Impact of that problem' value

Risk Analysis –Risk Analysis

Risk Analysis Table

Sr. No.	Potential Problems	Probability of Occurrence Value	Impact Value	Risk Exposure
1.	GUI Issues	7	3	21
2.	Payment Validation	6	6	36
3.	Balance Check	5	7	35
4.	SMS Alerts	6	9	54
5.	Email Alerts	5	8	40
6.	Fund Transfer	8	10	80

Risk Analysis – What is Risk

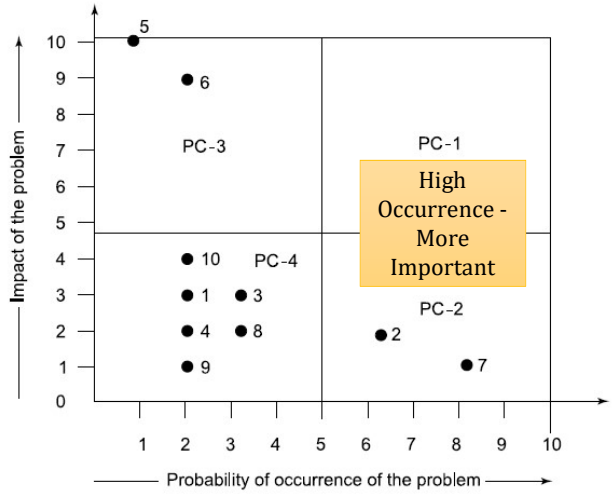
Examples:

- Output that is immediately needed during working hours is more critical than output that could be sent hours or days later.
- If large volumes of data to be sent by mail are wrong, just the cost of re-mailing may be horrible.

Risk Analysis –Risk Matrix

- Risk matrix is used to capture identified problems, estimate their probability of occurrence with impact and rank the risks based on this information.
- Risk Matrix can be used to assign thresholds that group the potential problems into priority categories.

Risk Analysis –Risk Matrix



Sample Risk Matrix with four quadrants

Each quadrant is a Priority Category

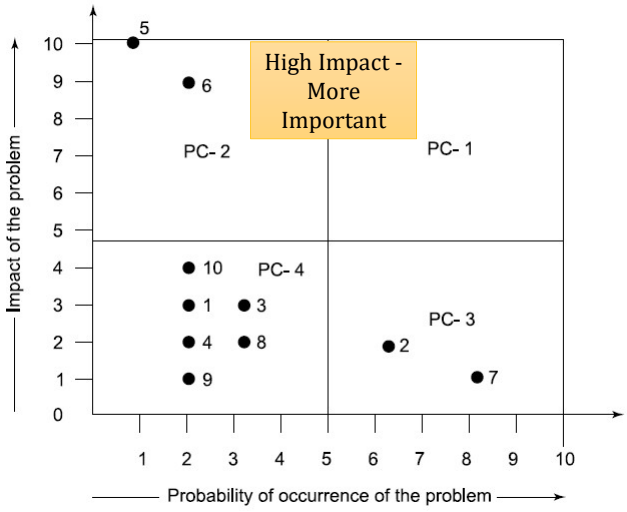
Priority Category-1:
High probability value and high impact value

Priority Category-2:
High probability value and low impact value

Priority Category-3:
Low probability value and high impact value

Priority Category-4:
Low probability value and low impact value

Risk Analysis –Risk Matrix



Sample Risk Matrix with four quadrants

Each quadrant is a Priority Category

Priority Category-1:
High probability value and high impact value

Priority Category-2:
Low probability value and high impact value

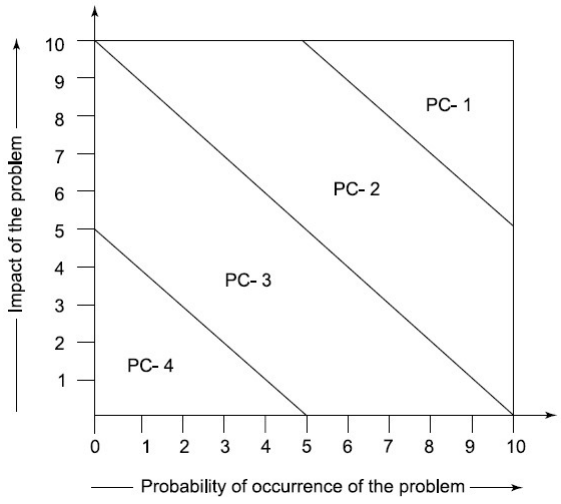
Priority Category-3:
High probability value and high impact value

Priority Category-4:
Low probability value and low impact value

Risk Analysis –Risk Matrix

- There may be situations where we do not want to give importance to any value and assign equal importance.
- In this case, the Diagonal Band Prioritization Scheme is more suitable.
- This scheme is more appropriate in situations where we have difficulty in assigning importance to either 'probability of occurrence of the problem' value or 'impact of that problem' value.

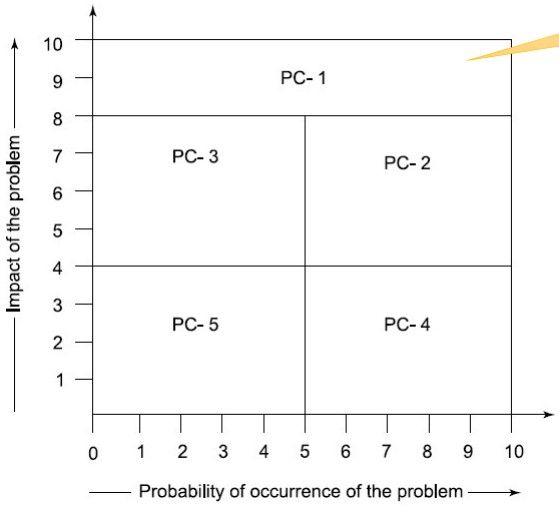
Risk Analysis –Risk Matrix



Equal Importance

Diagonal Band Prioritization Scheme

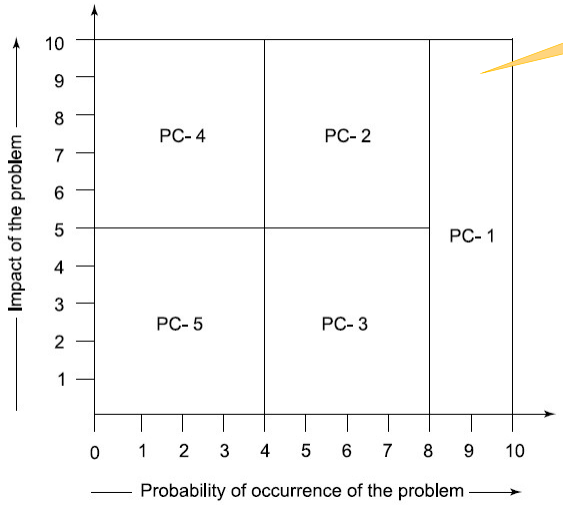
Risk Analysis –Risk Matrix



High Impact – More Important

- We may give priority to high impact value irrespective of the 'probability of occurrence' value.
- The highest priority (PC-1) is assigned to high impact value and for the other four quadrants; any prioritization scheme may be selected.

Risk Analysis –Risk Matrix



High Probability Occurrence – More Important

- We may give priority to high 'probability of occurrence' value irrespective of the 'impact' value.
- The highest priority (PC-1) is assigned to high probability of occurrence value and for the other four quadrants; any prioritization scheme may be selected.
- Not popular in practice.

Risk Analysis –Risk Matrix

- After the risks are ranked, the high priority risks are identified & managed first and then other priority risks in descending order.
- Risks should be discussed in a team and proper action should be recommended to manage these risks.
- Risk matrix is a powerful tool for designing prioritization schemes.
- Probability of occurrence value of a problem may be assigned on the scale of 1 to 10.
- The impact of the problem may be critical, serious, moderate, minor or negligible.

Code Coverage Prioritization Technique

Program **P**

```
1. main( )
2. {
3. int a, b, x, y, z;
4. scanf ("%d, %d", &a, &b);
5. x = a + b ;
6. y = a* b;
```

Modified Program **P'**

```
1. main ( )
2. {
3. int a, b, x, y, z;
4. scanf ("%d, %d", &a, &b);
5. x = a + b;
6. y = a – b;
```

Test Suite for Program **T**

Set of Test Cases S. No.	Inputs a	Inputs b	Execution History
1	2	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
2	1	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 14
3	3	2	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14
4	3	3	1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14

Objective: Execute modified portion(s) of the source code and the portion(s) affected by the modification(s) to see the correctness of modification(s).

No time to execute all test cases

Insufficient resources to execute all test cases

Reduce the size of T to T' to execute the modified portion of the source code and the portion(s) affected by modification(s).

Code Coverage Prioritization Technique

- Based on version specific test case prioritization.
- Selects **T'** from **T** which is a subset of T.
- The technique also prioritizes test cases of **T'** and recommends use of high priority test cases first and then low priority test cases in descending order till time and resources are available or a reasonable level of confidence is achieved.

Code Coverage Prioritization Technique

Test Case Selection Criteria

- Since, technique is based on version specific test case prioritization, information about changes in the program is known.
- Hence, prioritization is focused around the changes in the modified program.
- We may like to execute all modified lines of source code with a minimum number of selected test cases.

Code Coverage Prioritization Technique

Test Case Selection Criteria

- This technique identifies those test cases that:

1. Execute the modified lines of source code at least once.
2. Execute the lines of source code after deletion of deleted lines from the execution history of the test case and are not redundant.

- The technique uses two algorithms one for 'modification' and the other for 'deletion'.

Code Coverage Prioritization Technique

Test Case Selection Criteria

Following information is available with us to use this technique:

1. Program P with its modified program P'.
2. Test suite T with test cases t_1, t_2, t_3,.....t_n.
3. Execution history (number of lines of source code covered by a test case) of each test case of test suite T.
4. Line numbers of lines of source code covered by each test case are stored in a two dimensional array (t_{11}, t_{12}, t_{13},.....t_{ij}).

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm

- The 'modification' algorithm is used to minimize and prioritize test cases based on the modified lines of source code.
- Algorithm uses 7 variables.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm

S. No.	Variable name	Description
1.	T1	It is a two dimensional array and is used to store line numbers of lines of source code covered by each test case.
2.	modloc	It is used to store the total number of modified lines of source code.
3.	mod_locode	It is a one-dimensional array and is used to store line numbers of modified lines of source code.
4.	nfound	It is a one-dimensional array and is used to store the number of lines of source code matched with modified lines of each test case.
5.	pos	It is a one-dimensional array and is used to set the position of each test case when nfound is sorted.
6.	candidate	It is a one-dimensional array. It sets the bit to 1 corresponding to the position of the test case to be removed.
7.	priority	It is a one-dimensional array and is used to set the priority of the selected test case.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm

Step-1: Initialization of variables

- Consider a hypothetical program of 60 lines of code with a test suite of 10 test cases.
- The execution history is also given with test cases
- Assume that lines 1, 2, 5, 15, 35, 45, 55 are modified.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm - *Step-1*

Test case Id	Execution history
T1	1, 2, 20, 30, 40, 50
T2	1, 3, 4, 21, 31, 41, 51
T3	5, 6, 7, 8, 22, 32, 42, 52
T4	6, 9, 10, 23, 24, 33, 43, 54
T5	5, 9, 11, 12, 13, 14, 15, 20, 29, 37, 38, 39
T6	15, 16, 17, 18, 19, 23, 24, 25, 34, 35, 36
T7	26, 27, 28, 40, 41, 44, 45, 46
T8	46, 47, 48, 49, 50, 53, 55
T9	55, 56, 57, 58, 59
T10	3, 4, 60

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-1

1. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Initialize array T1[i][j] to zero
2. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
      (i) Store line numbers of line of source code covered by each test case.
3. Repeat for i=1 to number of modified lines of source code
   (a) Store line numbers of modified lines of source code in array mod_locode.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm

Step-2: Selection and Prioritization of test cases

- The second portion of the algorithm counts the number of modified lines of source code covered by each test case (nfound).

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

2. Repeat for all true cases
   (a) Repeat for i=1 to number of test cases
      (i) Initialize array nfound[i] to zeroes
      (ii) Set pos[i] =i
   (b) Repeat for i=1 to number of test cases
      (i) Initialize l to zero
      (ii) Repeat for j=1 to length of the test case
            If candidate[i] ≠ 1 then
            Repeat for k=1 to modified lines of source code
                  If t1[i][j]=mod_locode[k] then
                        Increment nfound[i] by one
                        Increment l by one

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Numbers of lines matched	Number of Matches (nfound)
T1	1, 2	2
T2	1	1
T3	5	1
T4	-	0
T5	5, 15	2
T6	15, 35	2
T7	45	1
T8	55	1
T9	55	1
T10	-	0

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

- Sort the *nfound* array
- Select the test case with the highest value of *nfound* as a candidate for selection.
- The test cases are arranged in increasing order of priority.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

(c) Initialize l to zero
(d) Repeat for i=0 to number of test cases
  (i) Repeat for j=1 to number of test cases
    If nfound[i]>0 then
      t=nfound[i]
      nfound[i]=nfound[j]
      nfound[j]=t
      t=pos[i]
      pos[i]=pos[j]
      pos[j]=t
(e) Repeat for i=1 to number of test cases
  (i) If nfound[i]=1 then
    Increment count
(f) If count = 0 then
  (i) Goto end of the algorithm
(g) Initialize candidate[pos[0]] = 1
(h) Initialize priority[pos[0]]= m+1

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Numbers of lines matched	Number of Matches (nfound)	Candidate	Priority
T1	1, 2	2	1	1
T5	5, 15	2	0	0
T6	15, 35	2	0	0
T2	1	1	0	0
T3	5	1	0	0
T7	45	1	0	0
T8	55	1	0	0
T9	55	1	0	0
T4	-	0	0	0
T10	-	0	0	0

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

- The test case with candidate=1 is selected in each iteration.
- The modified lines of source code included in the selected test case are removed from the mod_locode array.
- This process continues until there are no remaining modified lines of source code covered by any test case.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

(a) Repeat for i=1 to length of selected test cases
  (i) Repeat for j=1 to modified lines of source code
    If t1[pos[0]][i] = mod[j] then
      mod[j] = 0

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

- Since test case T1 is selected and it covers 1 and 2 lines of source code, these lines will be removed from the mod_locode array.
- mod_locode = [1, 2, 5, 15, 35, 45, 55] - [1, 2] = [5, 15, 35, 45, 55]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T5	2	5, 15	1	2
T6	2	15, 35	0	0
T3	1	5	0	0
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T5	2	5, 15	1	2
T6	2	15, 35	0	0
T3	1	5	0	0
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [5, 15, 35, 45, 55] – [5, 15] = [35, 45, 55]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T6	1	35	1	3
T7	1	45	0	0
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [35, 45, 55] – [35] = [45, 55]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T7	1	45	1	4
T8	1	55	0	0
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [45, 55] – [45] = [55]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

Test Cases	Number of matches (nfound)	Matches found	Candidate	Priority
T8	1	55	1	5
T9	1	55	0	0
T2	0	-	0	0
T3	0	-	0	0
T4	0	-	0	0
T10	0	-	0	0

mod_locode = [55] – [55] = [NIL]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Modification Algorithm – Step-2

- Hence test cases T1, T5, T6, T7 and T8 need to be executed on the basis of their corresponding priority.
- Out of ten test cases, we need to run only 5 test cases for 100% code coverage of modified lines of source code.
- This is 50% reduction of test cases.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm

- The 'deletion' portion of the technique is used to:

1. Update the execution history of test cases by removing the deleted lines of source code.
2. Identify and remove those test cases that cover only those lines which are covered by other test cases of the program.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm

S. No.	Variable	Description
1.	T1	It is a two-dimensional array. It keeps the number of lines of source code covered by each test case i.
2.	deloc	It is used to store the total number of lines of source code deleted.
3.	del_locode	It is a one-dimensional array and is used to store line numbers of deleted lines of source code.
4.	count	It is a two-dimensional array. It sets the position corresponding to every matched line of source code of each test case to 1.
5.	match	It is a one-dimensional array. It stores the total count of the number of 1's in the count array for each test case.
6.	deleted	It is a one-dimensional array. It keeps the record of redundant test cases. If the value corresponding to test case i is 1 in deleted array, then that test case is redundant and should be removed.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm

Step-1: Initialization of variables

- Consider a hypothetical program of 20 lines of code with a test suite of 5 test cases.
- The execution history is also given with test cases.
- Assume that line numbers 6, 13, 17 and 20 are modified, and line numbers 4, 7 and 15 are deleted from the source code.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm

Test case Id	Execution history
T1	1, 5, 7, 15, 20
T2	2, 3, 4, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19
T5	1, 2, 6, 8, 9, 13

delloc = 3
del_locode = [4, 7, 15]
modloc = 4
mod_locode = [6, 13, 17, 20]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-1*

Remove deleted lines of source from execution history

```
1. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to length of test case i
       (i) Repeat for l to number of deleted lines of source code
           If T1[i][j]=del_locode then
              Repeat for k=j to length of test case i
               T1[i][k]=T1[i][k+1]
              Initialize T1[i][k] to zero
              Decrement c[i] by one
```

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-1*

Execution history after remove 4, 7, 15

Test case Id	Execution history
T1	1, 5, 20
T2	2, 3, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19
T5	1, 2, 6, 8, 9, 13

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm

Step-2: Identification of redundant test cases

- Objective: To find redundant test cases.
- A test case is a redundant test case, if it covers only those lines which are covered by other test cases of the program.
- This situation may arise due to deletion of a few lines of the program.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

Initialize test case array with line numbers of lines of source code covered by each test case.

```
2. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
         (i) Initialize array t1[i][j] to zero
         (ii) Initialize array count[i][j] to zero
3. Repeat for i=1 to number of test cases
   (a) Initialize deleted[i] and match [i] to zero
4. Repeat for i=1 to number of test cases
   (a) Initialize c[i] to number of line numbers in each test case i
   (b) Repeat for j=1 to c[i]
   (c) Initialize t1[i][j] to line numbers of line of source code covered by each test case
```

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

- Compare lines covered by each test case with lines covered by other test cases.
- A two-dimensional array count is used to keep the record of line number matched in each test case.
- If all the lines covered by a test case are being covered by some other test case, then that test case is redundant and should not be selected for execution.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

```
5. Repeat for i=1 to number of test cases
   (a) Repeat for j=1 to number of test cases
       (i) If i≠j and deleted[j]≠1 then
                   Repeat for k=1 to until t1[i][k]≠0
                     Repeat for l=1 until t1[j][l]≠0
                   If t1[i][k]=t1[j][l] then
                         Initialize count [i][k]=1
   (b) Repeat for m=1 to c[i]
       (i) If count[i][m]=1 then
                 Increment match[i] with 1
   (c) If match[i]=c[i] then
       (i) Initialize deleted[i] to 1
6. Repeat for i=1 to number of test cases
   (a) If deleted[i] =1 then
       Remove test case i (as it is a redundant test case)
```

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

- On comparing all values in each test case with all values of other test cases, we found that test case 1 and test case 5 are redundant test cases.
- These two test cases do not cover any line which is not covered by other test cases.

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

Test Case	Line Number of LOC	Found In Test Case	Redundant Y/N
T1	1	T4	Y
	5	T2	Y
	20	T2	Y
T5	6	T3	Y
	8	T3	Y
	9	T3	Y
	1	T4	Y
	2	T2	Y
	13	T3	Y

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

Modified table after removing T1 and T5

Test case Id	Execution history
T2	2, 3, 5, 8, 16, 20
T3	6, 8, 9, 10, 11, 12, 13, 14, 17, 18
T4	1, 2, 5, 8, 17, 19

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

Now we will minimize and prioritize test cases using 'modification' algorithm.

Test Cases	Number of lines matched (found)	Number of matches (nfound)
T2	20	1
T3	6, 13, 17	3
T4	17	1

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

Arrange in sorted order

Test Cases	Number of matches (nfound)	Numbers of lines matched	Candidate	Priority
T3	3	6, 13, 17	1	1
T2	1	20	0	0
T4	1	17	0	0

mod_locode = [6, 13, 17, 20] – [6, 13, 17] = [20]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

Test Cases	Number of matches (nfound)	Numbers of lines matched	Candidate	Priority
T2	1	20	1	2
T4	0	-	0	0

mod_locode = [20] – [20] = [NIL]

Code Coverage Prioritization Technique

Test Case Selection Criteria – Deletion Algorithm - *Step-2*

- Hence, test cases T2 and T3 are needed to be executed
- Redundant test cases are T1 and T5.
- Out of the five test cases, we need to run only 2 test cases for 100% code coverage of modified code coverage.
- This is a 60% reduction.