SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part II-Software Testing

Structural Testing

Lecture # 22, 23, 24
1,2,4 Nov

TODAY'S OUTLINE

- Structural Testing
  - Control Flow Testing
    - Branch testing
    - Statement testing
    - Condition testing
    - Path testing
  - Guest session

STRUCTURAL TESTING

- More technical than functional testing.
- It attempts to design test cases from the source code and not from the specifications.
- The source code becomes the base document which is examined thoroughly in order to understand the internal structure and other implementation details.
- Structural testing techniques are also known as white box testing techniques
- Many structural testing techniques are available
  - control flow testing,
  - data flow testing,
  - slice based testing and
  - mutation testing.

CONTROL FLOW TESTING

- Identify paths of the program and write test cases to execute those paths. PATHS?
- There may be too many paths in a program and it may not be feasible to execute all of them. As the number of decisions increase in the program, the number of paths also increase accordingly.
- Every path covers a portion of the program. We define 'coverage' as a 'percentage of source code that has been tested with respect to the total source code available for testing'.
- Write test cases to achieve a reasonable level of coverage using control flow testing.
- The most reasonable level may be to test every statement of a program at least once before the completion of testing.
- Testing techniques based on program coverage criterion may provide an insight about the effectiveness of test cases.

CONTROL FLOW TESTING

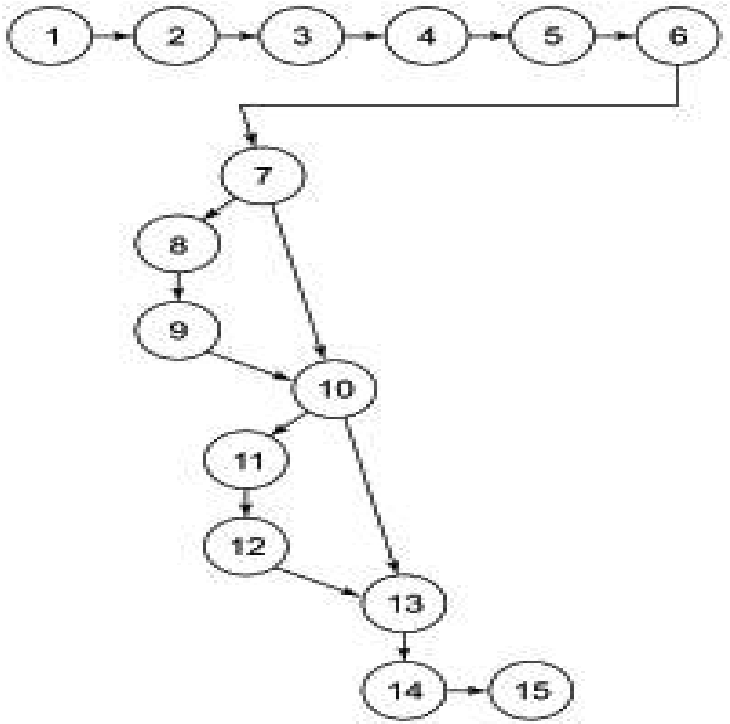
- Some of such techniques are discussed which are part of control flow testing.
- Statement Coverage
- Branch Coverage
- Condition Coverage

STATEMENT COVERAGE

- We want to execute every statement of the program in order to achieve 100% statement coverage.
- Consider the following portion of a source code along with its program graph.

```
#include<stdio.h>
#include<conio.h>

1.  void main()
2.  {
3.  int a,b,c,x=0,y=0;
4.  clrscr();
5.  printf("Enter three numbers:");
6.  scanf("%d %d %d",&a,&b,&c);
7.  if((a>b)&&(a>c)){
8.          x=a*a+b*b;
9.  }
10. if(b>c){
11.         y=a*a-b*b;
12. }
13. printf("x= %d y= %d",x,y);
14. getch();
15. }
```



TEST CASE

- a=9, b=8, c=7, all statements are executed and we have achieved 100% statement coverage by only one test case. The total paths of this program graph are given as:
  - 1–7, 10–15
  - 1–7, 10, 13–15
  - 1–10, 13–15
  - 1–15
- The cyclomatic complexity of this graph is:
- V(G) = e – n + 2P = 16 – 15 + 2 = 3
- Hence, independent paths are three and are given as:
  - 1–7, 10, 13–15
  - 1–10, 13–15
  - 1–7, 10–15
- Only one test case may cover all statements but will not execute all possible four paths and not even cover all independent paths.

BRANCH COVERAGE

- We want to test every branch of the program. Hence, we wish to test every 'True' and 'False' condition of the program.
- If we select a = 9, b = 8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions): Path = 1–15
- We also want to select all false conditions with the following inputs:
- a = 7, b = 8, c = 9, the path followed is Path = 1–7, 10, 13–15
- These two test cases out of four are sufficient to guarantee 100% branch coverage. The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

CONDITION COVERAGE

- Condition coverage is better than branch coverage because we want to test every condition at least once. However, branch coverage can be achieved without testing every condition.
- Considering the example on slide 6, statement number 7 has two conditions (a>b) and (a>c). There are four possibilities namely:
  - First is true, second is false
  - Both are true
  - First is false, second is true
  - Both are false
- If a > b and a > c, then the statement number 7 will be true (first possibility). However, if a < b, then second condition (a > c) would not be tested and statement number 7 will be false (third and fourth possibilities). If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition. Selected inputs may be given as:
- a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
- a = 9, b = 8, c = 7 (first possibility when both are true)
- a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)
- Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

PATH COVERAGE

- In this coverage criteria, we want to test every path of the program. There are too many paths in any program due to loops and feedback connections. It may not be possible to achieve this goal of executing all paths in many programs. If we do so, we may be confident about the correctness of the program. If it is unachievable, at least all independent paths should be executed.
- 1–7, 10–15
- 1–7, 10, 13–15
- 1–10, 13–15
- 1–15

TEST CASES FOR ALL PATHS

- Execution of all these paths increases confidence about the correctness of the program. Inputs for test cases are given as:

S. No.	Paths Id.	Paths	Inputs			Expected Output
			a	b	c	
1.	Path-1	1-7,10, 13-15	7	8	9	x=0 y=0
2.	Path-2	1-7, 10-15	7	8	6	x=0 y=-15
3.	Path-3	1-10, 13-15	9	7	8	x=130 y=0
4.	Path-4	1-15	9	8	7	x=145 y=17

PATH TESTING

- Path testing guarantee statement coverage, branch coverage and condition coverage.

EXAMPLE: PERFORM PATH TESTING

```
public static void search ( int key, int 0elemArray, Result r ) {
1. int bottom =0 ;
2. int top =elemArray.length - 1 ; int mid;
3. r.found =false ;
4. r.index =-1 ;
5. while ( bottom <= top ) {
6 mid =(top + bottom) / 2 ;
7 if (elemArray [mid] = key) {
8 rindex = mid;
9 r.found =true ;
10 return ; } // if part
else {
11 if (elemArray [mid] < key)
12 bottom = mid + 1 ;
else
13 top = mid - 1 ; }
} //while loop
14. } //search
```

That is all

SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

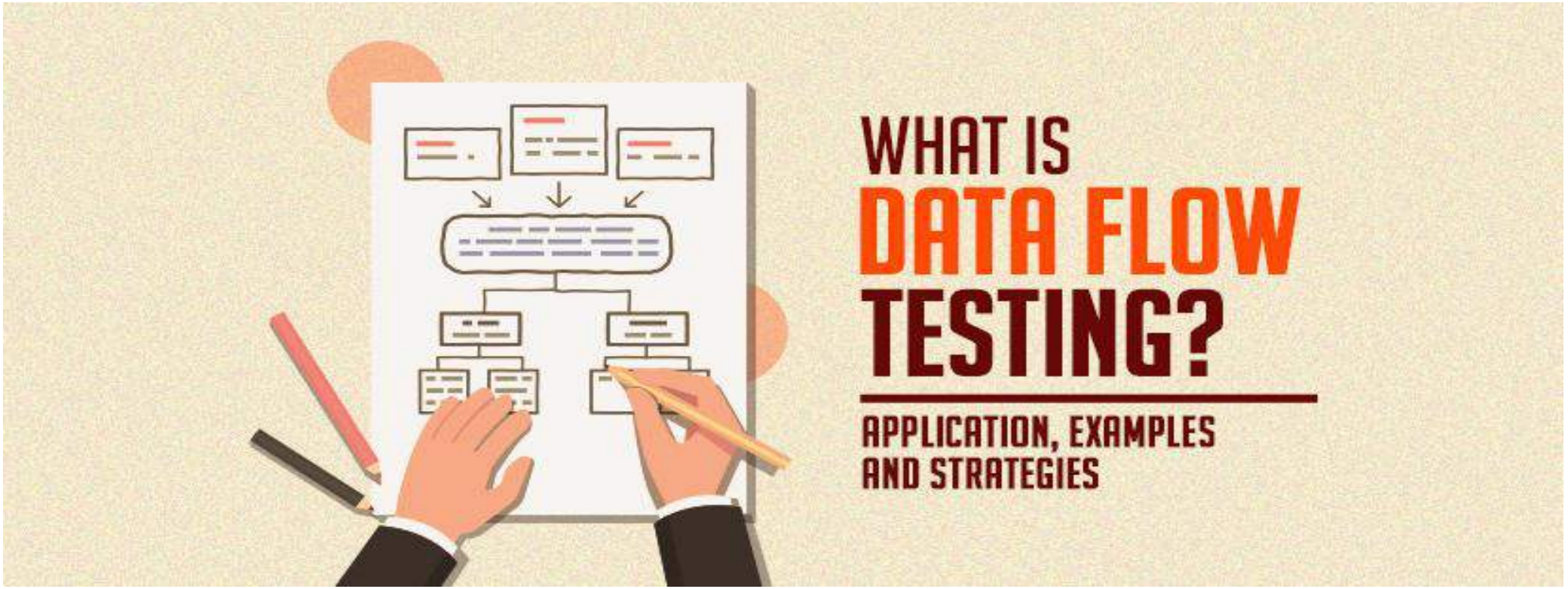
Part II-Software Testing

Structural Testing

Lecture # 25, 26, 27
8,9,11 Nov

TODAY'S OUTLINE

- Structural Testing
  - Data Flow Testing
  - Slice based testing
  - Mutation testing


WHAT IS
DATA FLOW
TESTING?
APPLICATION, EXAMPLES
AND STRATEGIES

DATA FLOW TESTING

- In control flow testing, we find various paths of a program and design test cases to execute those paths.
- We may like to execute every statement of the program at least once before the completion of testing.
- Consider the following program:

```
1. # include < stdio.h>
2. void main ()
3. {
4. int a, b, c;
5. a = b + c;
6. printf ("%d", a);
7. }
```

DATA FLOW TESTING

- Data flow testing may help us to minimize such mistakes. It is done to cover the path testing and branch testing gap.
- It has nothing to do with dataflow diagrams. It is based on variables, their usage and their definition(s) (assignment) in the program.
- The main points of concern are:
  - Statements where these values are used (referenced).
  - Statements where variables receive values (definition).
- Data flow testing focuses on variable definition and variable usage.
- The process is conducted to detect the bugs because of the incorrect usage of data variables or data values.

DEFINE/REFERENCE ANOMALIES

- Some of the define / reference anomalies are given as:
  - A variable is defined but never used / referenced.
  - A variable is used but never defined.
  - A variable is defined twice before it is used.
  - A variable is used before even first-definition.
- Define / reference anomalies may be identified by static analysis of the program.

DATA FLOW TESTING TERMS DEFINITIONS

- A program is first converted into a program graph.
- Defining node
- A node of a program graph is a defining node for a variable , if and only if, the value of the variable is defined in the statement corresponding to that node. It is represented as DEF (, n) where is the variable and n is the node corresponding to the statement in which is defined.
- Usage node
- A node of a program graph is a usage node for a variable , if and only if, the value of the variable is used in the statement corresponding to that node. It is represented as USE (, n), where ‘ ’ is the variable and ‘n’ in the node corresponding to the statement in which ‘ ’ is used.
- A usage node USE (, n) is a predicate use node (denoted as P-use), if and only if, the statement corresponding to node ‘n’ is a predicate statement otherwise USE (, n) is a computation use node (denoted as C-use).

DATA FLOW TESTING TERMS DEFINITIONS

- Definition use Path
- A definition use path (denoted as du-path) for a variable ‘ ’ is a path between two nodes ‘m’ and ‘n’ where ‘m’ is the initial node in the path but the defining node for variable ‘ ’ (denoted as DEF (, m)) and ‘n’ is the final node in the path but usage node for variable ‘ ’ (denoted as USE (, n)).

IDENTIFICATION OF DU PATHS

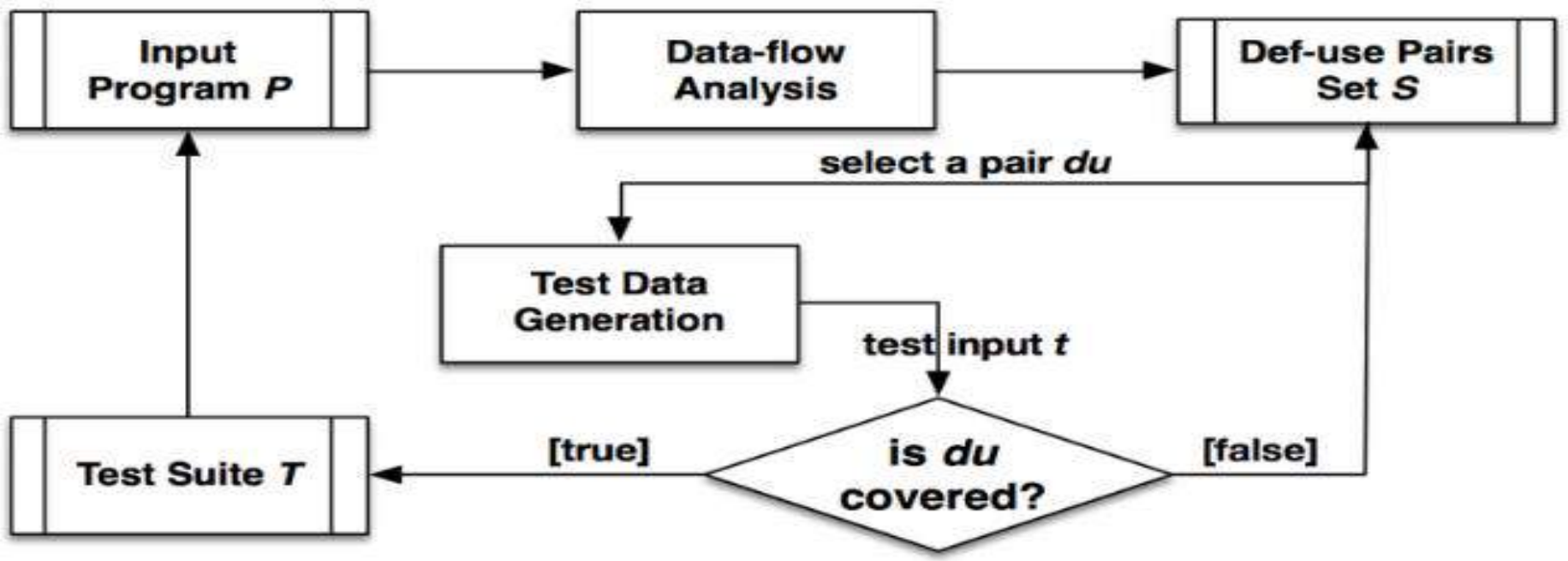
- The various steps for the identification of du and dc paths are given as:
- Draw the program graph of the program.
- Find all variables of the program and prepare a table for define / use status of all variables using the following format:

S. No.	Variable(s)	Defined at node	Used at node

- Generate all du-paths from define/use variable table of above step using the following

S. No.	Variable	du-path(begin, end)

DATA FLOW TESTING CYCLE



TESTING STRATEGIES USING DU-PATHS

- We want to generate test cases which trace every definition to each of its use and every use is traced to each of its definition. Some of the testing strategies are given as:
- **Test all du-paths**
- All du-paths generated for all variables are tested. This is the strongest data flow testing strategy covering all possible du-paths.
- **Test all uses**
- Find at least one path from every definition of every variable to every use of that variable which can be reached by that definition.
- For every use of a variable, there is a path from the definition of that variable to the use of that variable.

TESTING STRATEGIES USING DU-PATHS

- **Test all definitions**
- Find paths from every definition of every variable to at least one use of that variable;
- The first requires that each definition reaches all possible uses through all possible du-paths, the second requires that each definition reaches all possible uses, and the third requires that each definition reaches at least one use.

TYPES OF DATA FLOW TESTING

- Static Data Flow Testing
- No actual execution of the code is carried out in Static Data Flow testing. Generally, the definition, and usage pattern of the data variables is scrutinized through a control flow graph.
- Dynamic Data Flow Testing
- The code is executed to observe the transitional results. Dynamic data flow testing includes:
  - Identification of definition and usage of data variables.
  - Identifying viable paths between definition and usage pairs of data variables.
  - Designing & crafting test cases for these paths.

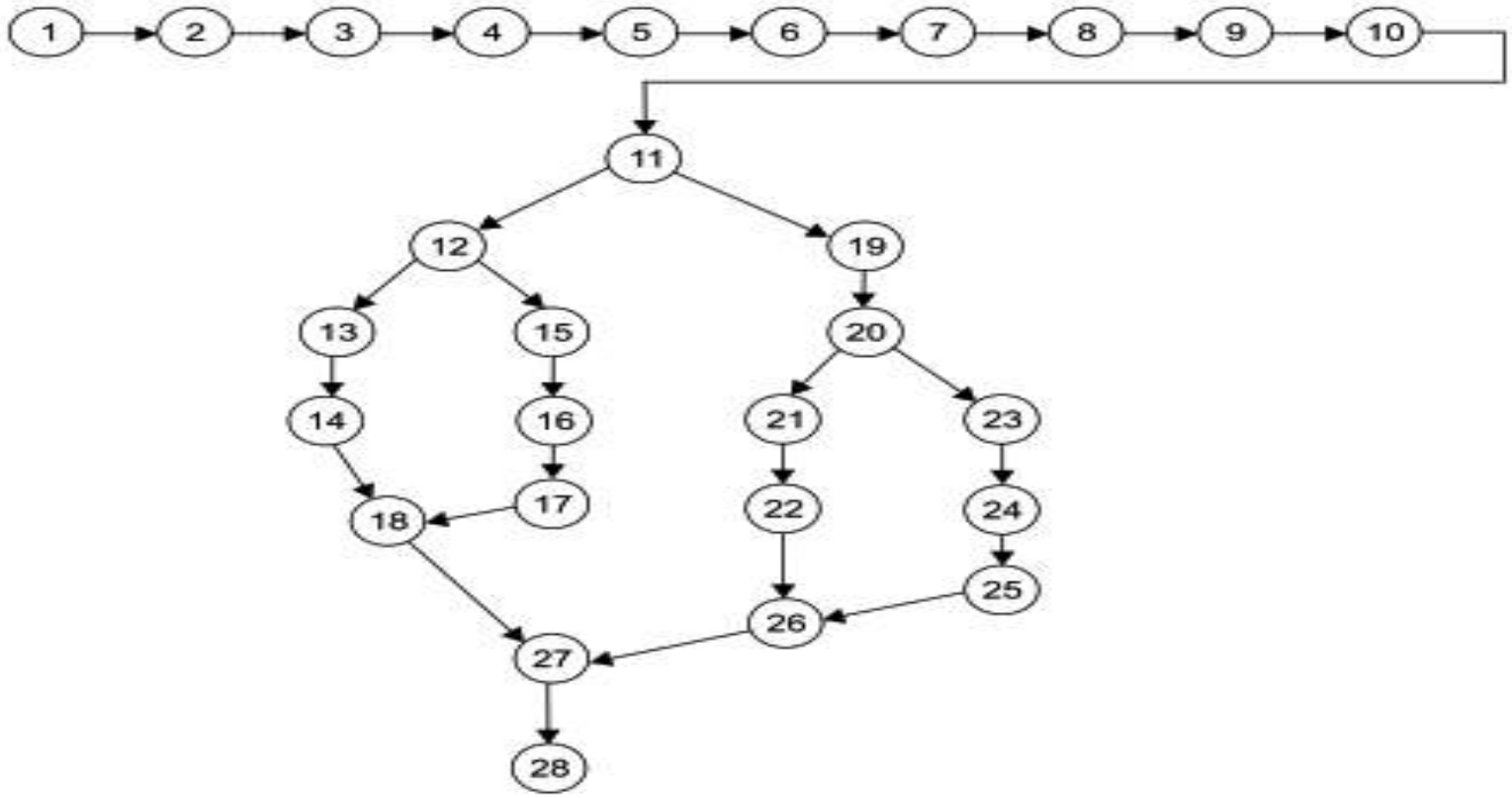
DATA FLOW TESTING LIMITATIONS

- Testers require good knowledge of programming.
- Time-consuming
- Costly process.

EXAMPLE: FIND THE LARGEST NUMBER AMONGST THREE NUMBERS.

```
#include<stdio.h>
#include<conio.h>
1.  void main()
2.  {
3.  float A,B,C;
4.  clrscr();
5.  printf("Enter number 1:\n");
6.  scanf("%f", &A);
7.  printf("Enter number 2:\n");
8.  scanf("%f", &B);
9.  printf("Enter number 3:\n");
10. scanf("%f", &C);
    /*Check for greatest of three numbers*/
11. if(A>B) {
12. if(A>C) {
13.         printf("The largest number is: %f\n",A);
14.         }
15. else  {
16.         printf("The largest number is: %f\n",C);
17.         }
18. }
19. else {
20. if(C>B) {
21.         printf("The largest number is: %f\n",C);
22.         }
23.  else  {
24.         printf("The largest number is: %f\n",B);
25.         }
26.  }
27.  getch();
28.  }
```

STEP I



STEP II & III

S. No.	Variable	Defined at node	Used at node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

The du-paths with beginning node and end node are given as:

Variable	du-path (Begin, end)
A	6, 11 6, 12 6, 13
B	8, 11 8, 20 8, 24
C	10, 12 10, 16 10, 20 10, 21

TEST CASES

Test all du-paths

S. No.	Inputs A	B	C	Expected Output	Remarks
1.	9	8	7	9	6–11
2.	9	8	7	9	6–12
3.	9	8	7	9	6–13
4.	7	9	8	9	8–11
5.	7	9	8	9	8–11, 19, 20
6.	7	9	8	9	8–11, 19, 20, 23, 24
7.	8	7	9	9	10–12
8.	8	7	9	9	10–12, ,15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19–21

SLICE BASED TESTING

- we prepare various subsets (called slices) of a program with respect to its variables and their selected locations in the program.
- Each variable with one of its location will give us a program slice.
- A large program may have many smaller programs (its slices), each constructed for different variable subsets.
- "Program slicing is a technique for restricting the behaviour of a program to some specified subset of interest. A slice S(, n) of program P on variable , or set of variables, at statement n yields the portions of the program that contributed to the value of just before statement n is executed. S (, n) is called a slicing criteria.
- Slices can be computed automatically on source programs by analyzing data flow.
- Hence, slices are smaller than the original program and may be executed independently.
- Only two things are important here, variable and its selected location in the program.

GUIDELINES FOR SLICING

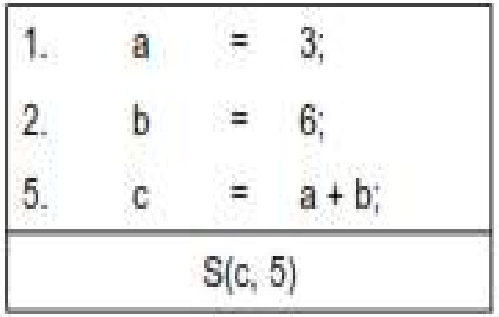
- All statements where variables are defined and redefined should be considered.
- All statements where variables receive values externally should be considered.
- All statements where output of a variable is printed should be considered.
- The status of all variables may be considered at the last statement of the program.

CREATION OF PROGRAM SLICES

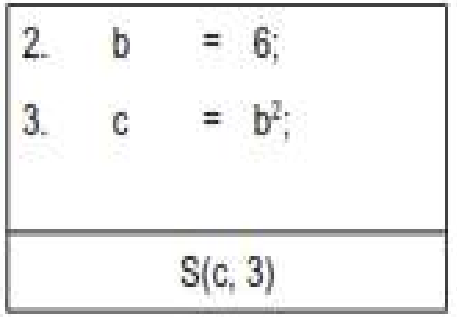
- Consider the portion of a program given in Figure 4.2 for the identification of its slices.
- a = 3;
- b = 6;
- c = b^2;
- d = a^2+ b^2;
- c = a + b;

PROGRAM SLICES

- We identify two slices for variable ‘c’ at statement number 3 and statement number 5.



Variable ‘c’ at statement 5



Variable ‘c’ at statement 5

EXAMPLE PROGRAM

- Consider the following program.

```
1. void main ( )
2. {
3. int a, b, c, d, e;
4. printf (“Enter the values of a, b and c \ n”);
5. scanf (“%d %d %d”, & a, &b, &c);
6. d = a+b;
7. e = b+c:
8. printf (“%d”, d);
9. printf (“%d”, e);
10. }
```

SOME SLICES

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
7.  e = b + c;
9.  printf ("%d", e);
10. }
```

Slice on criterion S (e, 10) = (1, 2, 3, 4, 5, 7, 9, 10)

```
1. main ( )
2. {
3. int a, b, c, d, e;
4. printf ("Enter the values of a, b and c \ n");
5. scanf ("%d %d %d", &a, &b, &c);
7.  e = b + c;
10. }
```

Slice on criterion S (e,7) = (1, 2, 3, 4, 5, 7,10)

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
10. }
```

Slice on criterion S (a, 5) = (1, 2, 3, 4, 5, 10)

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
6.  d = a + b;
8.  printf ("%d", d);
10. }
```

Slice on criterion S (d,10) = (1, 2, 3, 4, 5, 6, 8, 10)

```
1.  main ( )
2.  {
3.  int a, b, c, d, e;
4.  printf ("Enter the values of a, b and c \ n");
5.  scanf ("%d %d %d", &a, &b, &c);
6.  d = a + b;
10. }
```

Slice on criterion S (d,6) = (1, 2, 3, 4, 5, 6, 10)

GENERATION OF TEST CASES

- Every slice should be independently executable and may cover some lines of source code of the program.
- The test cases for the slices of the program given
- Generated slices to find the largest number amongst three numbers are S(A, 6), S(A, 13), S(A, 28), S(B, 8), S(B, 24), S(B, 28), S(C, 10), S(C, 16), S(C, 21), S(C, 28).

TEST CASES

Test cases using program slices of program to find the largest among three numbers

S. No.	Slice	Lines covered	A	B	C	Expected output
1.	S(A, 6)	1–6, 28	9			No output
2.	S(A, 13)	1–14, 18, 27, 28	9	8	7	9
3.	S(A, 28)	1–14, 18, 27, 28	8	8	7	9
4.	S(B, 8)	1–4, 7, 8, 28		9		No output
5.	S(B, 24)	1–11, 18–20, 22–28	7	9	8	9
6.	S(B, 28)	1–11, 19, 20, 23–28	7	9	8	9
7.	S(C, 10)	1–4, 9, 10, 28			9	No output
8.	S(C, 16)	1–12, 14–18, 27, 28	8	7	9	9
9.	S(C, 21)	1–11, 18–22, 26–28	7	8	9	9
10.	S(C, 28)	1–11, 18–22, 26–28	7	8	9	9

LIMITATIONS OF SLICE BASED TESTING

- Slice based testing is a popular structural testing technique and focuses on a portion of the program with respect to a variable location in any statement of the program.
- Hence slicing simplifies the way of testing a program's behaviour with respect to a particular subset of its variables.
- But slicing cannot test a behaviour which is not represented by a set of variables or a variable of the program.

MUTATION TESTING

- Popular technique to assess the effectiveness of a test suite.
- A large number of test cases for any program can be generated but cant execute all of them due to time or resources.
- So the idea is to select a few test cases using any testing technique and prepare a test suite.
- But how to assess the effectiveness of a selected test suite? Is this test suite adequate for the program?
- If the test suite is not able to make the program fail, there may be one of the following reasons:
- The test suite is effective but hardly any errors are there in the program. How will a test suite detect errors when they are not there?
- The test suite is not effective and could not find any errors. Although there may be errors, they could not be detected due to poor selection of test suite. How will errors be detected when the test suite is not effective?

MUTATION AND MUTANTS

- Mutation testing may help us to assess the effectiveness of a test suite and may also enhance the test suite, if it is not adequate for a program.
- Mutation is the process of changing a program.
- This change may be limited to one, two or very few changes in the program.

MUTATION AND MUTANTS

- To mutate a program means to change a program. We generally make only one or two changes in order to assess the effectiveness of the selected test suite
- We may make many mutants of a program by making small changes in the program.
- Every mutant will have a different change in a program.
- Every change of a program may give a different output as compared to the original program.
- The original program and mutant are syntactically correct and should compile correctly.

EXAMPLE: FIND THE LARGEST NUMBER AMONGST THREE NUMBERS.

```
#include<stdio.h>
#include<conio.h>
1.  void main()
2.  {
3.  float A,B,C;
4.  clrscr();
5.  printf("Enter number 1:\n");
6.  scanf("%f", &A);
7.  printf("Enter number 2:\n");
8.  scanf("%f", &B);
9.  printf("Enter number 3:\n");
10. scanf("%f", &C);
    /*Check for greatest of three numbers*/
11. if(A>B) {
12. if(A>C) {
13.         printf("The largest number is: %f\n",A);
14.         }
15. else  {
16.         printf("The largest number is: %f\n",C);
17.         }
18. }
19. else {
20. if(C>B) {
21.         printf("The largest number is: %f\n",C);
22.         }
23.     else  {
24.         printf("The largest number is: %f\n",B);
25.         }
26.     }
27.     getch();
28.     }
```

FIRST ORDER MUTANT OF THE EXAMPLE PROGRAM

- Many changes can be made in the program.
- Mutant M_1 is obtained by replacing the operator '>' of line number 11 by the operator ' ='.
- Mutant M_2 is obtained by changing the operator '>' of line number 20 to operator '<'.
- These changes are simple changes. Only one change has been made in the original program to obtain mutant M_1 and mutant M_2.

```
/*Check for greatest of three numbers*/
11.  if(A>B){  ← if(A=B) { mutated statement ('>' is replaced by '=')
12.  if(A>C) {
```

M_1: First order mutant

```
19.  else {
20.  if(C>B) {  ← if(C<B) { mutated statement ('>' is replaced by '<')
21.          printf("The largest number is: %f\n",C);
```

M_2: First order mutant

HIGH ORDER MUTANT

- The mutants generated by making only one change are known as first order mutants.
- We may obtain second order mutants by making two simple changes in the program and third order mutants by making three simple changes, and so on.
- The second order mutant (M_3) of the example program can be obtained by making two changes in the program and thus changing operator '>' of line number 11 to operator '<' and operator '>' of line number 20 to ' >=' .
- The second order mutants and above are called higher order mutants.
- Generally, in practice, we prefer to use only first order mutants in order to simplify the process of mutation.

SECOND ORDER MUTANT

```
11.   if(A>B) {   ← if(A<B) { mutated statement (replacing '>' by '<')
12.   if(A>C) {
13.           printf("The largest number is: %f\n",A);
14.           }
15.   else  {
16.           printf("The largest number is: %f\n",C);
17.           }
18.   }
19.   else {
20.   if(C>B) {   ← if(C≥B) { mutated statement (replacing '>'by '≥')
21.           printf("The largest number is: %f\n",C);
```

MUTATION OPERATORS

- Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The changed expression should be grammatically correct as per the used language.
- If one or more mutant operators are applied to all expressions of a program, we may be able to generate a large set of mutants.
- We should measure the degree to which the program is changed. If the original expression is x + 1, and the mutant for that expression is x + 2, that is considered as a lesser change as compared to a mutant where the changed expression is (y * 2) by changing both operands and the operator.
- If x – y is changed to x – 5 to make a mutant, then we should not use the value of y to be equal to 5. If we do so, the fault will not be revealed.

MUTATION OPERATORS

- Some of the mutant operators for object oriented languages like Java, C++ are given as:
  - Static modifier change
  - Changing the access modifier, like public to private.
  - Argument order change
  - Super Keyword change
  - Operator change
  - Any operand change by a numeric value.

MUTATION SCORE

- When we execute a mutant using a test suite, we may have any of the following outcomes:
  - The results of the program are affected by the change and any test case of the test suite detects it. If this happens, then the mutant is called a killed mutant.
  - The results of the program are not affected by the change and any test case of the test suite does not detect the mutation. The mutant is called a live mutant.
- The mutation score associated with a test suite and its mutants is calculated as:

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

MUTATION SCORE

- The mutation score measures how sensitive the program is to the changes and how accurate the test suite is.
- A mutation score is always between 0 and 1.
- A higher value of mutation score indicates the effectiveness of the test suite although effectiveness also depends
- on the types of faults that the mutation operators are designed to represent.
- The live mutants are important for us and should be analyzed thoroughly.
- Why is it that any test case of the test suite not able to detect the changed behaviour of the program?
- One of the reasons may be that the changed statement was not executed by these test cases. If executed, then also it has no effect on the behaviour of the program.

EXAMPLE: PROGRAM TO FIND THE LARGEST OF THREE NUMBERS

- Generate five mutants (M_1 to M_5) and calculate the mutation score of this test suite.

S. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20

FIVE MUTANTS

Mutated statements

Mutant No.	Line no.	Original line	Modified Line
M_1	11	if(A>B)	if (A<B)
M_2	11	if(A>B)	if(A>(B+C))
M_3	12	if(A>C)	if(A<C)
M_4	20	if(C>B)	if(C=B)
M_5	16	printf("The Largest number is:%f\n",C);	printf("The Largest number is:%f\n",B);

PROGRAM RESULTS AFTER EXECUTING MUTANTS

Actual output of mutant M_1

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	6
2.	10	6	2	10	6
3.	6	2	10	10	10
4.	6	10	20	20	20

Actual output of mutant M_2

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	20

PROGRAM RESULTS AFTER EXECUTING MUTANTS

Actual output of mutant M_3

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	2
3.	6	2	10	10	6
4.	6	10	20	20	20

Actual output of mutant M_4

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	10

Actual output of mutant M_5

Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	2
4.	6	10	20	20	20

MUTATION SCORE

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$
$$= \frac{4}{5}$$
$$= 0.8$$

That is all

SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part III-Software Inspection

Lecture # 31, 32, 33
29, 30 Nov, 2 Dec

TODAY'S OUTLINE

- Software Inspection
  - Fagan inspections
  - Software reviews
  - Inspection checks and metrics
- Presentations

SOFTWARE INSPECTION

- Most commonly performed software quality assurance (QA) activity besides testing.
- Inspection directly detects and corrects software problems without resorting to execution, therefore it can be applied to many types of software artifacts.
- Depending on various factors, such as the techniques used, software artifacts inspected, the formality of inspection, number of people involved, etc., inspection activities can be classified and examined individually, and then compared to one another.
- Software inspection deals with finding software defects through critical examination by human inspectors.
- As a result of this direct examination, the detected software defects are typically precisely located, and therefore can be fixed easily in the follow-up activities.

THE CASE FOR INSPECTION

- The main difference between the object types of inspection and testing, namely executable programs for testing and all kinds of software artifacts for inspection,
- The primary reason for the existence of inspection: One does not have to wait for the availability of executable programs before one can start performing inspection.
- Consequently, the urgent need for QA and defect removal in the early phases of software development can be supported by inspection, but not by testing.
- In addition, various software artifacts available late in the development can be inspected but not tested, including product release and support plans, user manuals, project schedule and other management decisions, and other project documents.
- Basically, anything tangible can be inspected.

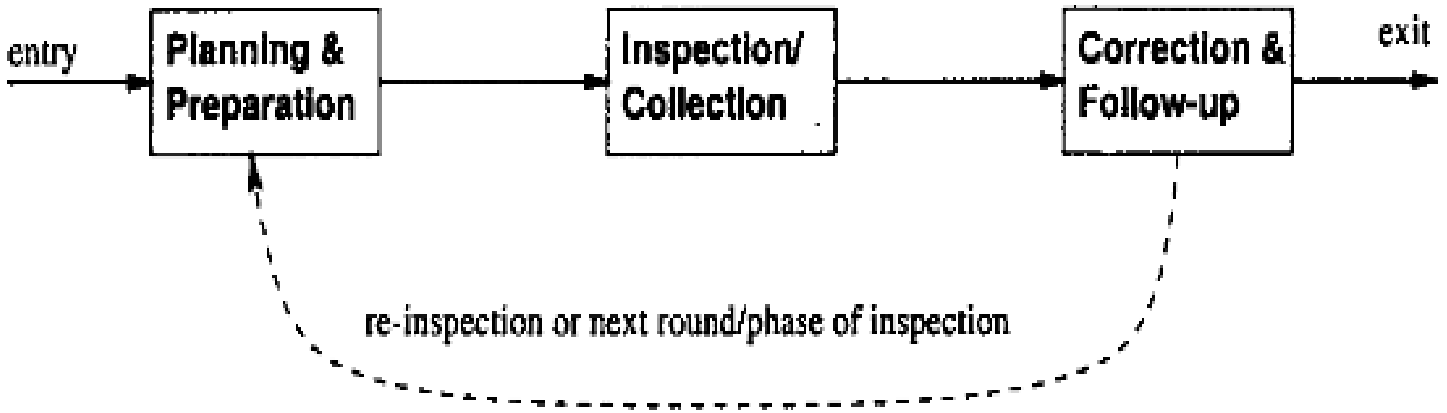
INSPECTION TECHNIQUES

- Wide variety of objects for inspection, so inspection techniques also vary considerably.
- For example, code inspection can use the program implementation details as well as product specifications and design documents to help with the inspection.
- Inspection of test plans may benefit from expected usage scenarios of the software product by its target customers.
- Inspection of product support plans must take into account the system configuration of the product in operation and its interaction with other products and the overall operational environment.
- Consequently, different inspection techniques need to be selected to perform effective inspection on specific objects.

DEGREE OF FORMALITY

- Similarly, there are different degrees of formality, ranging from informal reviews and checks to very formal inspection techniques associated with precisely defined individual activities and exact steps to follow.
- Even at the informal end, some general process or guidelines need to be followed so that some minimal level of consistency can be assured, and adequate coverage of important areas can be guaranteed.
- In addition, some organizational and tool support for inspection is also needed.

GENERIC INSPECTION PROCESS



Generic inspection process

PLANNING AND PREPARATION

- Inspection planning needs to answer the general questions about the inspection, including:
  - What are the objectives or goals of the inspection?
  - What are the software artifacts to be inspected or the objects of the inspection?
  - Who are performing the inspection?
  - Who else need to be involved, in what roles, and with what specific responsibilities?
  - What are the overall process, techniques, and follow-up activities of the inspection?

INSPECTION OR COLLECTION

- This step roughly corresponds to the execution of QA activities in generic quality engineering process.
- This step is also referred to as collection or collection meeting.
- The focus of this step is to detect faults in the software artifacts inspected, and record the inspection results so that these faults can be resolved in the next step.

CORRECTION AND FOLLOW-UP

- The discovered faults need to be corrected by people who are responsible for the specific software artifacts inspected. For example, in design or code inspection, the responsible designer or programmer, often labeled as design or code "owners" in industry, need to fix the design or code.
- There should be some follow-up activities to verify the fix.
- Sometimes, new inspection rounds can be planned and carried out,

FAGAN INSPECTION

- The earliest and most influential work in software inspection is Fagan inspection (Fagan, 1976), which is almost synonymous with the term "inspection" itself.
- Fagan inspection has been used widely across different industrial boundaries and on many different software artifacts, although most often on program code.
- Almost all the other inspection processes and techniques can be considered as derivatives of Fagan inspection, by enhancing, simplifying, or modifying it in various ways to fit specific application environment or to make it more effective or efficient with respect to certain criteria.

PROCESS AND PARTICIPANTS

- Planning: Deciding what to inspect, who should be involved, in what role, and if inspection is ready to start.
- Overview meeting: The author meets with and gives an overview of the inspection object to the inspectors. Assignment of individual pieces among the inspectors is also done.
- Preparation: Individual inspection is performed by each inspector, with attention focused on possible defects and question areas.

PROCESS AND PARTICIPANTS

- Inspection meeting to collect and consolidate individual inspection results. Fault identification in this meeting is carried out as a consensus building process.
- Rework: The author fixes the identified problems or provides other responses.
- Follow-up: Closing the inspection process by final validation.

GENERIC INSPECTION PROCESS AND FAGAN INSPECTION PROCESS

- We can adapt the generic inspection program (slide 7) to depict Fagan inspection in the following:
- The **"planning and preparation"** block can be expanded into three sequential steps, "planning", "overview", and "preparation" in Fagan inspection.
- The **"inspection/collection"** is directly mapped to the "inspection" step.
- The **"correction and follow-up"** block can be expanded into two sequential steps, "correction", and "follow-up".
- The dotted link for the next round of inspection is eliminated.

INSPECTION TEAM

- Fagan inspection typically involves about four people in the inspection team
- The potential inspectors are identified in the planning stage (Step 1) from those designers, developers, testers, or other software professionals or managers, who are reasonably familiar with the software artifacts to be inspected, but not necessarily those who directly work on it.
- An ideal mix would include people with different roles, background, experience, and different personal or professional characteristics, to bring diverse views and perspectives to the inspection.

INSPECTION WORK

- The assignment of individual pieces for inspection among the inspectors needs to take two issues into consideration: overall coverage and areas of focus.
- On the one hand, different inspectors will be assigned different pieces so as not to unnecessarily duplicate inspection effort.
- On the other hand, some important or critical pieces may need the focused attention of more than one inspector.
- The inspection meeting should be an organized event, with one inspector identified as the leader or moderator, who oversees the meeting and ensures that it fulfills its main purpose of defect identification and consolidation.
- The meeting typically lasts two hours or less.

INSPECTION WORK

- The focus is on defect detection and consolidation only, but not on defect resolution,
- A group of people working together would find and confirm problems that individuals may not.
- However, each individual must be fully prepared and bring forward candidate problems for the team to examine together.
- In this group process, false alarms will be eliminated, and consolidated defects will be confirmed, recorded, and handed over for authors to fix.

GENERAL OBSERVATIONS AND FINDINGS

- The importance of preparation
- Variations with team size, moderator role, and session coordination
- Defect detection techniques used in inspection
- Additional use of inspection feedback

OTHER INSPECTIONS AND RELATED ACTIVITIES

- Variations to Fagan inspection have been proposed and used to effectively conduct inspection under different environments.
- Some of them are direct responses to some of the general findings of Fagan inspection described above. We organize these inspection techniques and processes along two dimensions:
  - size and scope of the inspection,
  - formality of the inspection.

That is all

SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part III-Other Quality Assurance Techniques
Software review, walk through
Lecture # 34, 35
6, 7 Dec

TODAY'S OUTLINE

- Presentations
- Gilb inspection
- Desk check
- Review
- Walkthrough
- Comparison of different QA techniques w.r.t
  - Defect perspective
  - Problem type
  - Interpretation
  - Level of difficulty

OTHER INSPECTIONS AND RELATED ACTIVITIES

- Variations to Fagan inspection have been proposed and used to effectively conduct inspection under different environments.
- Some of them are direct responses to some of the general findings of Fagan inspection described above.
- We organize these inspection techniques and processes along two dimensions:
  - size and scope of the inspection,
  - formality of the inspection.

INSPECTIONS OF REDUCED SCOPE OR TEAM SIZE

- Fagan inspection teams typically consist of four members.
- However, some software artifacts are small enough to be inspected by one or two inspectors.
- Similarly, such reduced-size inspection teams can be used to inspect software artifacts of limited size, scope, or complexity.
- This so-called two-person inspection is the simplification form of Fagan inspection, with an author-inspector pair, but following essentially the same process for Fagan inspection.
- This technique is cheaper and more suitable for smaller-scale programs, small increments of design and/or code in the incremental or iterative development, or other software artifacts of similarly smaller size.

INSPECTIONS OF REDUCED SCOPE OR TEAM SIZE

- Another implementation of two-person inspection is the reversible author- inspector pair, that is, the individuals in the pair complement their roles by inspecting each other's software artifacts.
- easier to manage technique because of the mutual benefit to both individuals instead of the asymmetric relation in Fagan inspection, where the author is the main beneficiary while the inspectors are performing "service" to others or to the company.
- The idea of two-person inspection is also found in the new development paradigm called agile development and extreme programming, where the so-called paired programming resembles the author-inspector pair.
- Informal inspection
- Reduce cost

INSPECTIONS OF ENLARGED SCOPE OR TEAM SIZE

- A common extension to Fagan inspection is based on the observation that during Fagan inspection meeting, people tend to linger on discovered defects and try to both find the causes for them and suggest fixes.
- These additional activities in the meeting would interfere with the main task of defect detection and confirmation in Fagan inspection and tend to prolong the meeting.
- On the other hand, these activities do add valuable information to the feedback that can be used to improve the overall inspection process and product quality.
- A solution to this problem is proposed in the Gilb inspection.

GILB INSPECTION

- In Gilb inspection an additional step, called "process brainstorming", is added right after the inspection meeting in Fagan inspection.
- The focus of this step is root cause analysis aimed at preventive actions and process improvement in the form of reduced defect injections for future development activities.
- There are several other special features to Gilb inspection, as characterized below:
- The input to the overall inspection process is the product document, rules, checklists, source documents, and kin documents. The emphasis is that any technical documentation, even diagrams, can be inspected.
- The output from the overall inspection process is the inspected (and corrected) input documents, change requests, and suggested process improvements.
- The inspection process forms a feedback loop, with the forward part resembling Fagan inspection but with the added step for process brainstorming, and the feedback part consisting of inspection statistics and adjustment to inspection strategies.
- Multiple inspection sessions are likely through this feedback loop

GILB INSPECTION

- The Gilb inner inspection steps are as follows (with Fagan inspection equivalent given inside parenthesis):
  - 1. planning (same),
  - 2. kickoff (overview),
  - 3. individual checking (preparation),
  - 4. logging meeting (inspection),
  - 5a. edit (rework),
  - 5b. process brainstorming (),
  - 6. edit audit (follow-up).
- 5a and 5b are carried out in parallel in Gilb inspection.
- The team size is typically about four to six.
- Checklists are extensively used, particularly for step 3, individual checking.

PHASED INSPECTION

- Another variation to the above is the phased inspection, where the overall inspection is divided into multiple phases with each focusing on a specific area or a specific class of problems.
- These problems not only include the defects (correctness problems), but also issues with portability, maintainability, etc.
- This inspection is typically supported by some form of checklist and related software tools.
- The dynamic team make-up reflects the different focus and skill requirements for individual phases.

INFORMAL DESK CHECKS, REVIEWS, AND WALKTHROUGHS

- **Desk check** typically refers to informal check or inspection of technical documents produced by oneself, which is not too different from proofreading one's own writings to catch and correct obvious mistakes.
- Advance software tools can detect things as mis-spelling, format, syntactical errors.
- Instead, desk checks should focus on logical and conceptual problems, to make effective use of the valuable time of software professionals.
- **Review** typically refers to informal check or inspection of technical

REVIEW

- Review typically refers to informal check or inspection of technical documents, but in this case, produced by someone else, either organized as individual effort, or as group effort in meetings, conference calls, etc.
- The focus of these reviews should be similar to desk checks, that is, on logical and conceptual problems.
- The differences in views, experience, and skill set are the primary reasons to use some reviews to complement desk checks.
- In most companies, the completion of a development phase or sub-phase and important project events or milestones are typically accompanied by a review, such as requirement review, design review, code review, test case review, etc.

WALKTHROUGH

- A special form of review is called walkthrough,
- a more organized review typically applied to software design and code.
- Meetings are usually used for these walkthroughs.
- The designer or the code owner usually leads the meeting, explaining the intentions and rationales for the design or the code, and the other reviewers (meeting participants) examine these design code for overall logical and environmental soundness and offer their feedback and suggestions.
- Defect detection is not the focus.
- Typically, these meetings require less time and preparation by the participants except for the owners.
- In practical applications, these informal checks, reviews, and walkthroughs can be used in combination with formal inspections.

EFFECTIVENESS COMPARISON

- Different QA alternatives can be compared by examining the specific perspectives of defect they are dealing with, what kind of problems they are good at addressing, their suitability to different defect levels and pervasiveness, and their ability to provide additional information for quality improvement.

EFFECTIVENESS COMPARISON : DEFECT PERSPECTIVE

- Among the different defect related perspectives and concepts, the QA alternatives can be compared by examining whether they are dealing with error sources, errors, faults, failures, or accidents.
- This examination can be broken down further into two parts:
- Detection or observation of specific problems from specific defect perspectives during the performance of specific QA activities.
- Types of follow-up actions that deal with the observed or detected problems in specific ways as examined from the defect perspectives.

EFFECTIVENESS COMPARISON: DEFECT PERSPECTIVE

QA Alternative	Defect Perspective	
	At Observation	At Follow-up (& Action)
testing	failures	fault removal
defect prevention	errors & error sources	reduced fault injection
inspection	faults	fault removal
formal verification	(absence of) faults	fault absence verified
fault tolerance	local failures	global failures avoided
failure containment	accidents	hazards resolution & damage reduction

EFFECTIVENESS COMPARISON: PROBLEM TYPES

- Different QA alternative might be effective for different types of problems, including dealing with different perspectives of defects, ranging from different errors and error sources, various types of faults, and failures of different severity and other characteristics.

Main problem types dealt with by different QA alternatives

QA Alternative	Problem Types
testing	dynamic failures & related faults
defect prevention	systematic errors or conceptual mistakes
inspection	static & localized faults
formal verification	logical faults, indirectly
fault tolerance	operational failures in small areas
failure containment	accidents and related hazards

EFFECTIVENESS COMPARISON: PROBLEM TYPES

- Defect prevention works to block some errors or to remove error sources to prevent the injection of related faults. Therefore, it is generally good at dealing with conceptual mistakes made by software designers and programmers.
- Once such conceptual mistakes can be identified as error sources, they can be effectively eliminated.
- One key difference between inspection and testing is the way faults are identified: inspection identifies them directly by examining the software artifact, while failures are observed during testing and related faults are identified later by utilizing the recorded execution information.
- This key difference leads to the different types of faults commonly detected using these two techniques.
- Inspection is usually good at detecting static and localized faults which are often related to some common conceptual mistakes, while testing is good at detecting dynamic faults involving multiple components in interactions.

THE REASONS OF DIFFERENCES BETWEEN THE TWO TYPES OF QA ALTERNATIVES:

- Inspection involves static examination while testing involves dynamic executions. Therefore, static problems are more likely to be found during inspection, while dynamic problems are more likely to be found during testing.
- It is hard for human inspectors to keep track of multiple components and complicated interactions over time, while the same task may not be such a difficult one for computers. Therefore, testing is generally better at detecting interaction problems involving multiple components.
- Human inspectors can focus on a small area and perform in-depth analysis, leading to effective detection of localized faults.

EFFECTIVENESS COMPARISON

- Formal verification deals with logical (or mathematical) correctness, and can be interpreted as extremely formalized inspection. Therefore, it shares some of the characteristics of inspection in dealing with static and logical problems.
- Problem identification is only a side-effect of failing to produce a correctness proof.
- Fault tolerance and failure containment are designed to work with dynamic operational problems that may lead to global failures or accidents.
- Fault tolerance techniques are good at isolating faults to only cause local failures but not global ones, while failure containment works to contain failures that may lead to accidents by dealing with hazards or reducing damage related to accidents.

EFFECTIVENESS COMPARISON: DEFECT LEVEL AND PERVASIVENESS

- Different QA techniques may be suitable for different defect levels or pervasiveness.

Defect levels where different QA alternatives are suitable

QA Alternative	Defect Level	
testing	low – medium	
defect prevention	low – high	(particularly pervasive problems)
inspection	medium – high	
formal verification	low	
fault tolerance	low	
failure containment	lowest	

EFFECTIVENESS COMPARISON: RESULT INTERPRETATION AND CONSTRUCTIVE INFORMATION

- Ease of result interpretation plays an important role in the application of specific QA techniques. A good understanding of the results is a precondition to follow-up actions.
- For example, both inspection and testing are aimed at defect removal. However, inspection results are much easier to interpret and can be used directly for defect removal. Testing results need to be analyzed by experienced software professionals to locate the faults that caused the failures observed during testing, and only then can these faults be removed.
- Result interpretation for formal verification, fault tolerance, and failure containment is harder than that for inspection and testing. A significant amount of effort is needed to analyze these results to support follow-up actions.
- For example, in a fault tolerant system using recovery blocks, repeated failures need to be dealt with off-line by analyzing the dynamic records. Much information related to unanticipated environment and usage not covered in the pre-planned testing activities may be included in these records.
- Similarly, failure containment results typically need additional analysis support.

EFFECTIVENESS COMPARISON: RESULT INTERPRETATION AND CONSTRUCTIVE INFORMATION

Ease of result interpretation for different QA alternatives and amount of constructive information/measurements

QA Alternative	Result Interpretation	Information/Measurement
testing	moderate	executions & failures
defect prevention	(intangible)	experience
inspection	easy	faults, already located
formal verification	hard	fault absence verified
fault tolerance	hard	(unanticipated) environments/usages
failure containment	hard	accident scenarios and hazards

COMPARISON SUMMARY

General comparison for different QA alternatives

QA Alternative	Applicability	Effectiveness	Cost
testing	code	occasional failures	medium
defect prevention	known causes	systematic problems	low
inspection	s/w artifacts	scattered faults	low – medium
formal verification	formal spec.	fault absence	high
fault tolerance	duplication	rare-cond. failures	high
failure containment	known hazards	rare-cond. accidents	highest

That is all

SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part IV-QUANTIFIABLE QUALITY IMPROVEMENT
Feedback Loop and Activities for Quantifiable Quality Improvement
Lecture # 36
9 Dec

TODAY'S OUTLINE

- Feedback Loop and Activities for Quantifiable Quality Improvement
- Feedback Loop and Overall Mechanism
- Monitoring and Measurement
- Analysis and Feedback
- Tool and Implementation Support

QUANTIFIABLE QUALITY IMPROVEMENT BASIC ELEMENTS

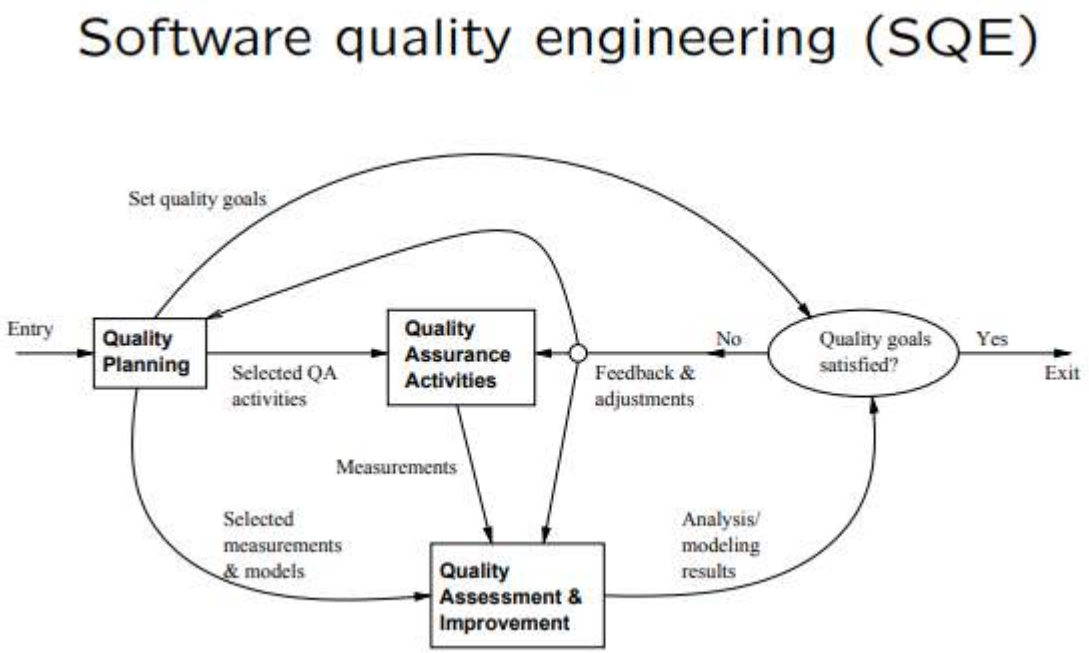
- Part IV is quantifiable quality improvement, which includes two basic elements:
- Quantification of quality through quantitative measurements and models so that the quantified quality assessment results can be compared to the pre-set quality goals for quality and process management.
- Quality improvement through analyses and follow-up activities by identifying quality improvement possibilities, providing feedback, and initiating follow-up actions.

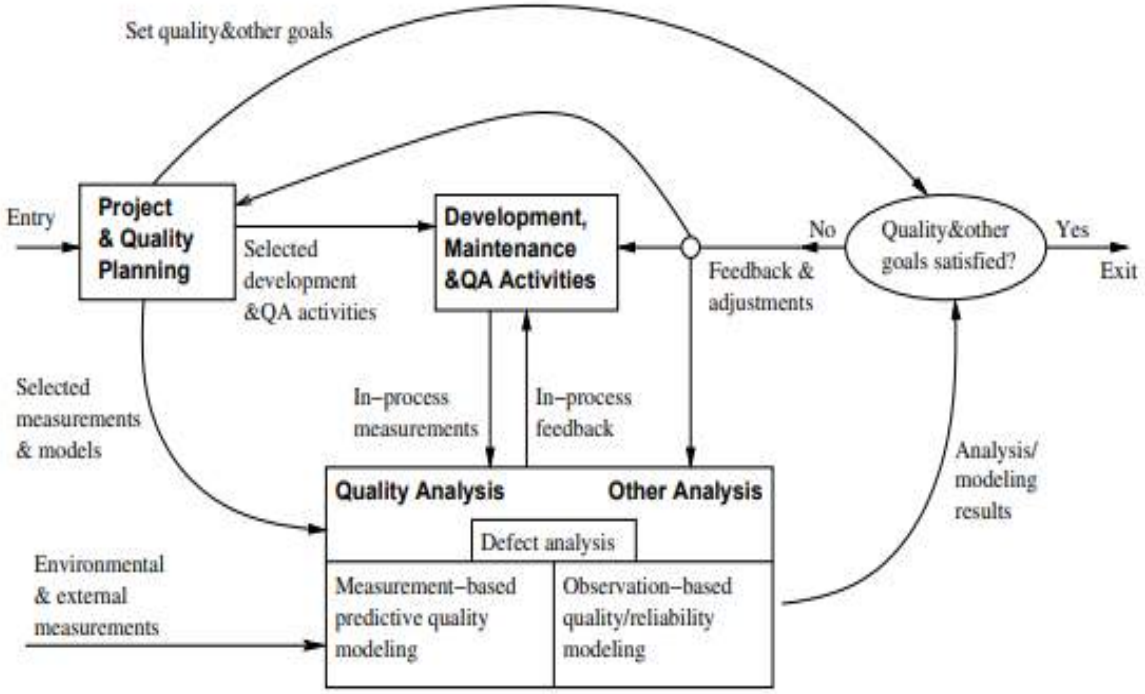
QUANTIFIABLE QUALITY IMPROVEMENT

- To support quantifiable quality improvement, various parallel and follow-up activities to the main quality assurance (QA) activities are needed, including:
- Monitoring the specific QA activities and the overall software development or maintenance activities, and extracting relevant measurement data.
- Analyzing the data collected above for quality quantification and identification of quality improvement opportunities.
- Providing feedback to the QA and development/maintenance activities and carrying out follow-up actions based on the analysis results above.

QUANTIFIABLE QUALITY IMPROVEMENT

- These activities also close the quality engineering feedback loop discussed in part I and refine it into Following:





IMPORTANCE OF FEEDBACK LOOP

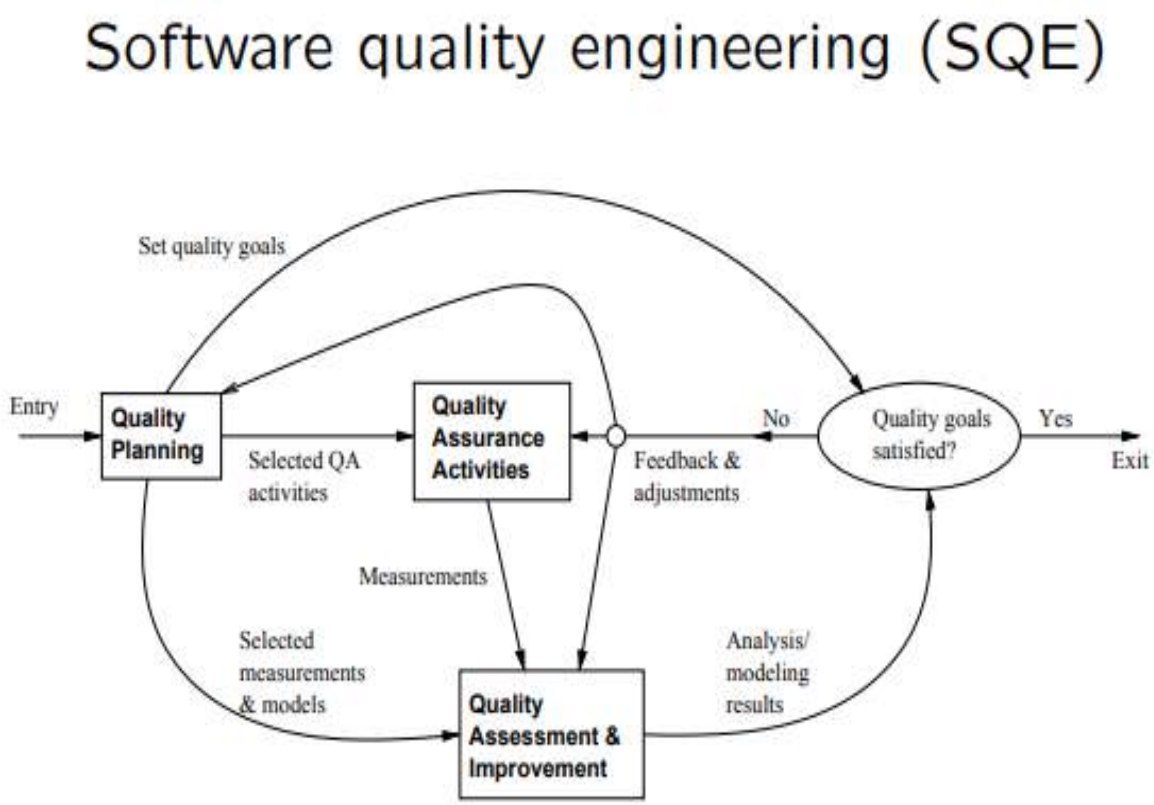
- All QA activities covered in Part II and Part III need additional support: .
- Planning and goal setting
- Management via feedback loop
  - When to stop?
  - Adjustment and improvement, etc.
  - All based on assessments/predictions

IMPORTANCE OF FEEDBACK LOOP

- Feedback loop for quantification/improvement: i.e. Focus of Part IV
  - mechanism and implementation
  - models and measurements.
  - defect analyses and techniques
  - risk identification techniques
  - software reliability engineering

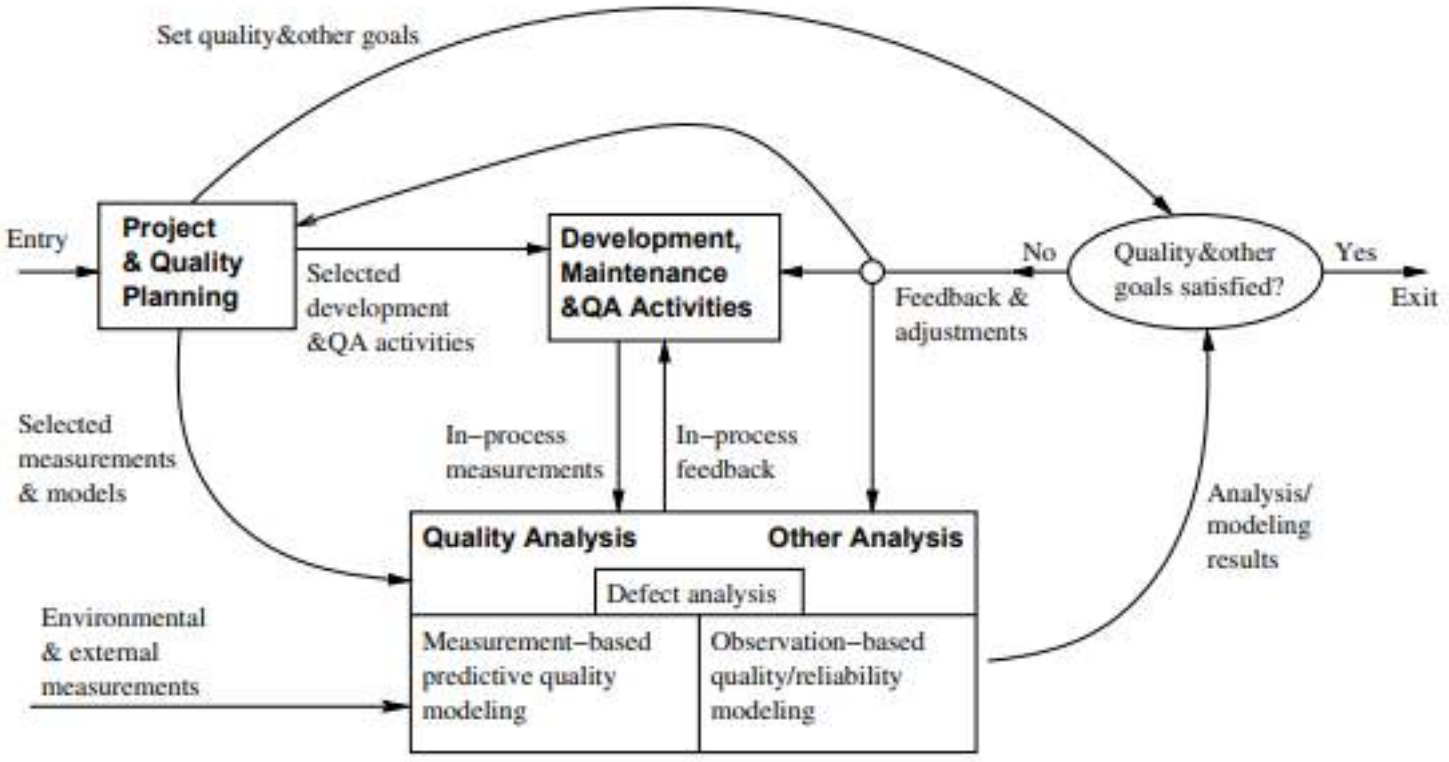
QE ACTIVITIES AND PROCESS REVIEW

- Major activities:
- Pre-QA planning (Part I)
- QA (Part II and Part III)
- Post-QA analysis & feedback – Part IV (maybe parallel instead of "post-")



QE ACTIVITIES AND PROCESS REVIEW

- Feedback loop zoom-in:
- Multiple measurement sources.
- Many types of analysis performed.
- Multiple feedback paths.



FEEDBACK LOOP RELATED ACTIVITIES

- Monitoring and measurement: .
  - defect monitoring ∈ process management
  - defect measurement ∈ defect handling
  - many other related measurements.
- Analysis modeling:
  - Historical baselines and experience
  - Choosing models and analysis techniques
  - Focus on defect/risk/reliability analyses
  - Goal: assessment/prediction/improvement.
- Feedback and follow-up
  - Frequent feedback: assessment/prediction
  - Possible improvement areas identified
  - Overall management and improvement.

QUALITY MONITORING AND MEASUREMENTS

- Quality monitoring needs:
  - Quality as a quantified entity over time
  - Able to assess, predict, and control
  - Various measurement data needed
- Some directly in quality monitoring
- Others via analyses to provide feedback.
- Direct quality measurements:
  - Result, impact and related info. e.g., success vs. failure
- Defect information: directly monitored.
- Mostly used in quality monitoring.

INDIRECT QUALITY MEASUREMENTS

- Indirect quality measurements: Why?
- Other quality measurements (reliability) need additional analyses/data.
- Unavailability of direct quality measurements early in the development cycle ⇒ early (indirect) indicators.
- Used to assess/predict/control quality. (to link to or affect various direct quality measurements)
- Types of indirect quality measurements:
  - Environmental measurements.
  - Product internal measurements.
  - Activity measurements.

INDIRECT MEASUREMENTS: ENVIRONMENT

- Process characteristics
  - Entities and relationships
  - Preparation, execution and follow-up
  - Techniques used
- People characteristics
  - Skills and experience
  - Roles: planners/developers/testers
  - Process management and teams
- Product characteristics
  - Product/market environment
  - Hardware/software environment

INDIRECT MEASUREMENTS: INTERNAL

- Product internal measurements: most studied/understood in SE
- Software artifacts being measured:
  - Mostly code-related
  - Sometimes SRS, design, docs etc.
- Product attributes being measured:
  - Control: e.g., McCabe complexity
  - Data: e.g., Halstead metrics
  - Presentation: e.g., indentation rules

INDIRECT MEASUREMENTS: ACTIVITY

- Execution/activity measurements:
  - Overall: e.g., cycle time, total effort.
  - Phased: profiles/histograms.
  - Detailed: transactions
- Testing activity examples:
  - Timing during testing/usage
  - Path verification (white-box)
  - Usage-component mapping (black-box)
  - Measurement along the path
- Usage of observations/measurements: observation-based and predictive models

IMMEDIATE FOLLOW-UP AND FEEDBACK

- Immediate (without analyses): Why?
  - Immediate action needed right away:
    - critical problems ⇒ immediate fixing
    - most other problems: no need to wait
  - Some feedback as built-in features in various QA alternatives and techniques.
  - Activities related to immediate actions.
- Testing activity examples:
  - Shifting focus from failed runs/areas.
  - Re-test to verify defect fixing.
  - Other defect-related adjustments.
- Defect and activity measurements used.

ANALYSES, FEEDBACK, AND FOLLOW-UP

- Most feedback/followup relies on analyses.
- Types of analyses:
  - Product release decision related.
  - For other project management decisions, at the phase or overall project level.
  - Longer-term or wider-scope analyses.
- Types of feedback paths:
  - Shorter vs. longer feedback loops.
  - Frequency and time duration variations.
  - Overall scope of the feedback.
  - Data source refinement.
  - Feedback destinations.

ANALYSIS FOR PRODUCT RELEASE DECISIONS

- Most important usage of analysis results
  - Prominent in SQE and modified SQE Figure
  - Related to: "when to stop testing?"
- Basis for decision making:
  - Without explicit quality assessment:
    - implicit: planned activities,
    - indirect: coverage goals,
    - other factors: time/$-based.
  - With explicit quality assessment:
    - failure-based: reliability,
    - fault-based: defect count & density.
- Criteria preference:
  - reliability – defect – coverage – activity.

ANALYSES FOR OTHER DECISIONS

- Transition from one (sub-)phase to another:
  - Later ones: similar to product release.
  - Earlier ones: reliability undefined
    - defects – coverage – activity,
    - inspection and other early QA
- Other decisions/management-activities:
  - Schedule adjustment.
  - Resource allocation and adjustment.
  - Planning for post-release support.
  - Planning for future products or updates.
- These are product-level or sub-product-level decisions and activities

OTHER FEEDBACK AND FOLLOWUP

- Other (less frequent) feedback/followup:
  - Goal adjustment (justified/approved).
  - Self-feedback (measurement & analysis)
  - Longer term, project-level feedback.
  - May even carry over to followup projects.
- Beyond a single-project duration/scope:
  - Future product quality improvement
    - overall goal/strategy/model/data,
    - especially for defect prevention.
- Process improvement.
- More experienced people

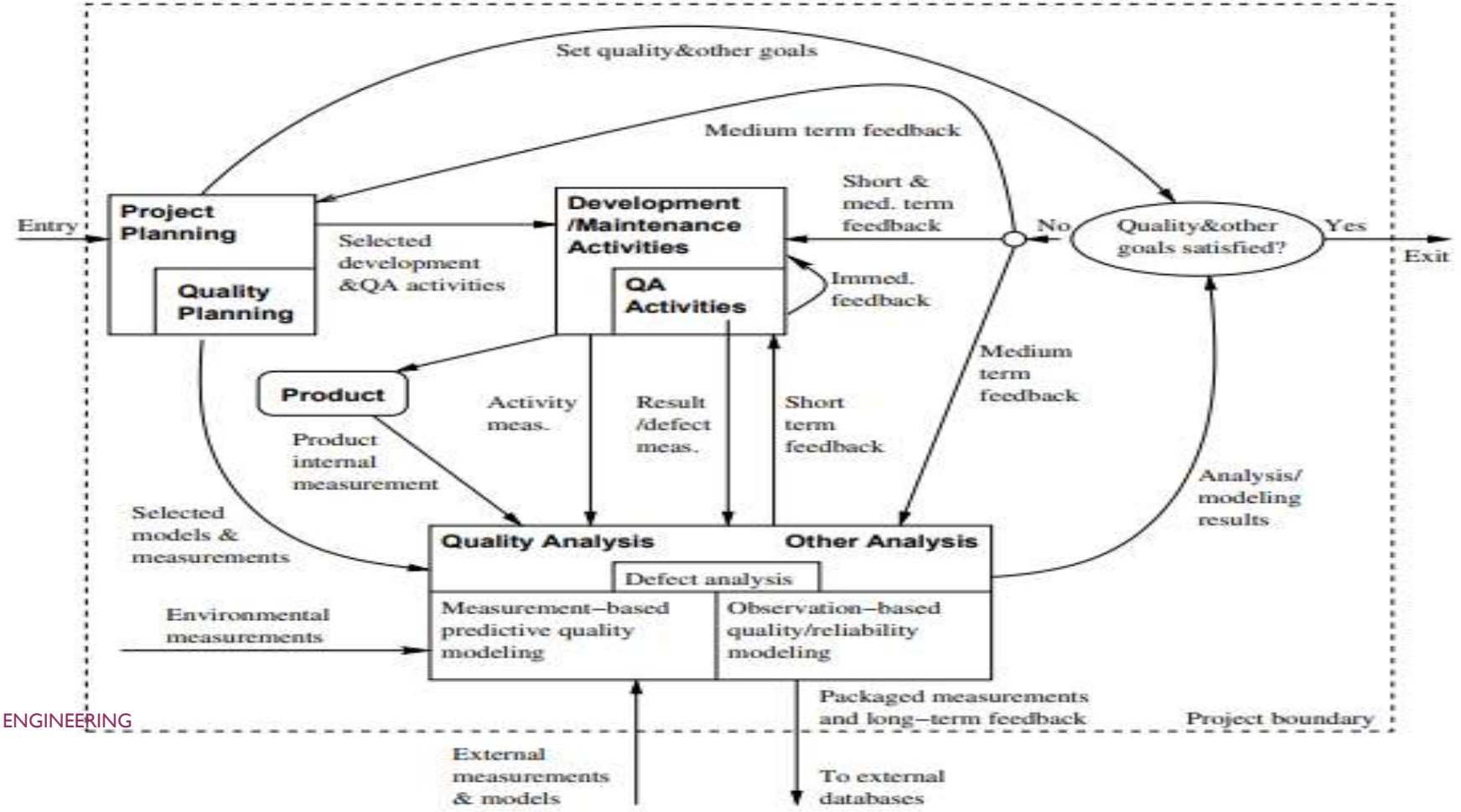
FEEDBACK LOOP IMPLEMENTATION

- Key question: sources and destinations. (Analysis and modeling activity at center.)
- Sources of feedback loop = data sources:
  - Result and defect data:
    - the QA activities themselves.
  - Activity data:
    - both QA and development activities.
  - Product internal data: product. (produced by development activities)
  - Environmental data: environment.
- Additional sources of feedback loop:
  - From project/QA planning.
  - Extended environment: measurement data and models beyond project scope.

FEEDBACK LOOP IMPLEMENTATION

- Feedback loop at different duration/scope levels.
- Immediate feedback to current development activities (locally).
- Short-term or sub-project-level feedback:
  - most of the feedback/followup
  - transition, schedule, resource,
  - destination: development activities.
- Medium-term or project-level feedback:
  - overall project adjustment and release
  - destination: major blocks
- Longer-term or multi-project feedback:
  - to external destinations

FEEDBACK LOOP IMPLEMENTATION



IMPLEMENTATION SUPPORT TOOLS

- Type of tools:
  - Data gathering tools.
  - Analysis and modeling tools.
  - Presentation tools.
- Data gathering tools:
  - Defects/direct quality measurements:
    - from defect tracking tools.
  - Environmental data: project db.
  - Activity measurements: logs.
  - Product internal measurements:
    - commercial/home-build tools.
  - New tools/APIs might be needed.

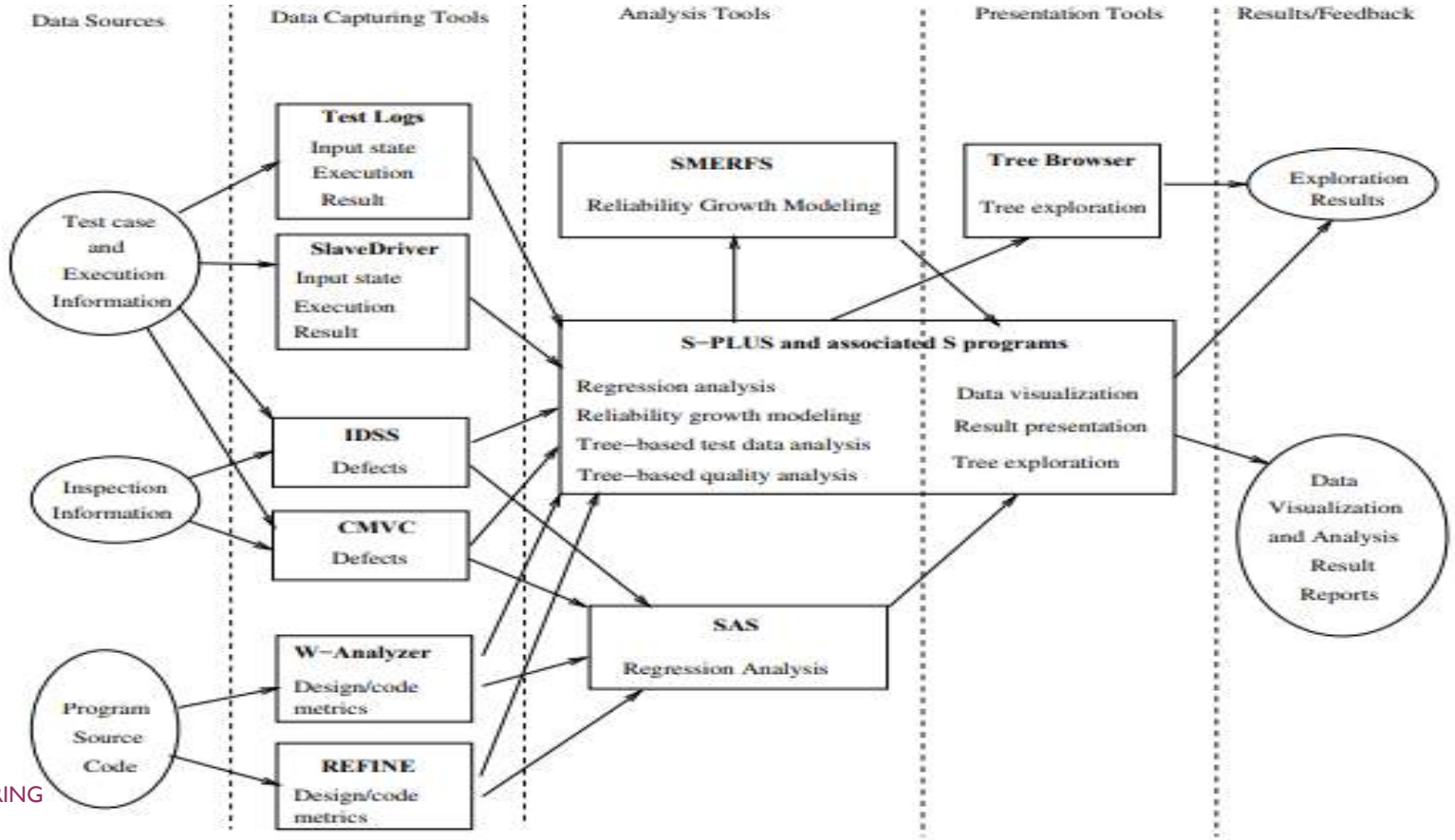
IMPLEMENTATION SUPPORT TOOLS

- Analysis and modeling tools:
  - Dedicated modeling tools:
    - e.g., SMERFS and CASRE for SRE
  - General modeling tools/packages:
    - e.g., multi-purpose S-Plus, SAS.
  - Utility programs often needed for data screening and processing.
- Presentation tools:
  - Aim: easy interpretation of feedback ⇒ more likely to act on.
  - Graphical presentation preferred.
  - Some "what-if"/exploration capability.

STRATEGY FOR TOOL SUPPORT

- Using existing tools ⇒ cost↓:
  - Functionality and availability/cost.
  - Usability.
  - Flexibility and programmability.
  - Integration with other tools.
- Tool integration issues:
  - Assumption: multiple tools used. (All-purpose tools not feasible/practical.)
  - External rules for inter-operability,
    - common data format and repository.
  - Multi-purpose tools.
  - Utilities for inter-operability.

TOOL SUPPORT EXAMPLE



That is all

SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part IV-QUANTIFIABLE QUALITY IMPROVEMENT
Quality models and measurements, SCM
Lecture # 37, 38, 39
13, 14, 16 Dec

TODAY'S OUTLINE

- Analysis → Feedback loop→ Follow-up
- Quality models and measurements
  - Types of Quality Assessment Models.
  - Comparing Quality Assessment Models.
  - Data Requirements and Measurement
  - Measurement and Model Selection.
- Software Configuration Management (SCM)
  - Why SCM?
  - SCM concepts and Terminology
  - Software Configuration Management Activities
  - Change Management
  - Build and Release Management
  - System Release
- Revision

QUALITY MEASUREMENT

- The primary purpose of the measurement and analysis activities is to provide feedback and useful information to manage software quality, the quality engineering process, and the overall software development/maintenance process and activities.
- The feedback and information provided are based on the analysis results using various models on the data collected from the quality assurance (QA) and the general development activities.
- In this lecture we examine and classify these models, relate them to the required measurements, compare the different models, and outline a general strategy to select appropriate models and measurements to satisfy specific quality assessment and improvement goals under specific application environments.

QA DATA AND ANALYSIS

- Generic testing process:
  - Test planning and preparation.
  - Execution and measurement.
  - Test data analysis and follow-up.
  - Related data ⇒ quality ⇒ decisions
- Other QA activities:
  - Similar general process.
  - Data from QA/other sources
  - Models used in analysis and follow-up:
    - provide timely feedback/assessment
    - prediction, anticipating/planning
    - corrective actions ⇒ improvement

QA MODELS AND MEASURES

- General approach
  - Adapt GQM-paradigm.
  - Quality: basic concept and ideas.
  - Compare models ⇒ taxonomy.
  - Data requirements ⇒ measurements.
  - Practical selection steps.
  - Illustrative examples.
- Quality attributes and definitions:
  - Q models: data ⇒ quality
  - Correctness vs. other attributes
  - Our definition/restriction: being defect-free or of low-defect
  - Examples: reliability, safety, defect count/density/distribution/etc.

QUALITY ANALYSIS

- Analysis and modeling:
- . Quality models: data ⇒ quality
  - a.k.a. quality assessment models or quality evaluation models
- Various models needed for .
  - Assessment, prediction, control
  - Management decisions
  - Problematic areas for actions
  - Process improvement
- Measurement data needed
- Direct quality measurements: success/failure (& defect info)
- Indirect quality measurements:– activities/internal/environmental. Indirect but early quality indicators.
- All described in Chapter 18

QUALITY MODELS

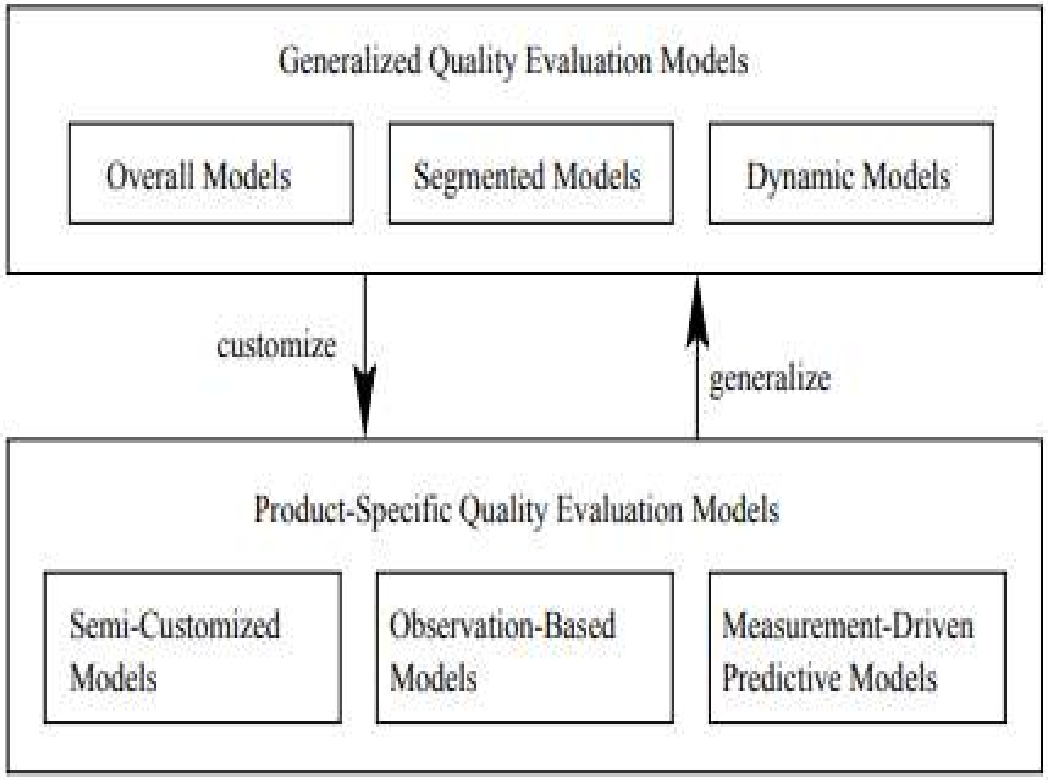
- Practical issues:
  - Applicability vs. appl. environment
  - Goal/Usefulness: information/results?
  - Data: measurement data required
  - Cost of models and related data
- Type of quality models
  - Generalized: averages or trends
  - Product-specific: more customized
  - Relating to issues above

MODELS FOR QUALITY ASSESSMENT

- Quality assessment models are analytical models that provide quantitative assessment of selected quality characteristics or sub-characteristics based on measurement data from software projects.
- Such models can help us obtain an objective assessment of our current product quality, in contrast to the often unreliable subjective assessment based on personal judgment or imprecise qualitative assessment.
- When applied over time, these models can provide us with an accurate prediction of the future quality, which can be used to help us make project scheduling, resource allocation, and other management decisions.

GENERALIZED MODELS: OVERALL

- Model taxonomy:
- Generalized:
  - overall, segmented, and dynamic
- Product-specific:
  - semi-customized: product history
  - observation-based: observations
  - measurement-driven: predictive



GENERALIZED MODELS: OVERALL

- Key characteristics
  - Industrial averages/patterns: ⇒ (single) rough estimate.
  - Most widely applicable.
  - Low cost of use.
- Examples: Defect density.
  - Estimate total defect with sizing model.
  - (counting in-field unique defect only)
- Non-quantitative overall models:
  - As extension to quantitative models.
  - Examples: 80:20 rule, and other general observations.

GENERALIZED MODELS: SEGMENTED

- Key characteristics:
- Estimates via product segmentation.
- Model: segment → quality.
- Multiple estimates provided.
- Other applications.
- Commonly used in software estimation.
- Example: COCOMO models.

A segmented model for reliability level estimation

Product Type	Failure Rate (per hour)	Reliability Level
safety-critical software	$< 10^{-7}$	ultra-high
commercial software	10^{-3} to 10^{-7}	moderate
auxiliary software	$> 10^{-3}$	low

GENERALIZED MODELS: DYNAMIC

- Example: Putnam model
- Rayleigh curve for failure rate: $r = 2Bate^{-at2}$
- Overall/average trend over time.
- Often expressed as a mathematical function or an empirical curve.
- Combined models possible,
- e.g., segmented dynamic models.

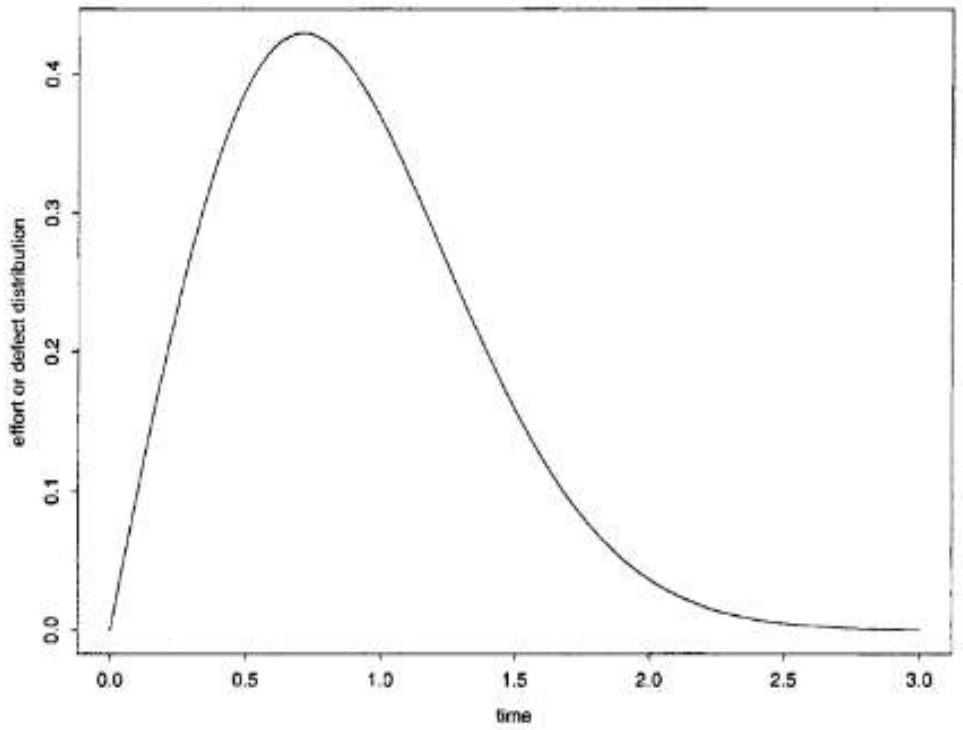


Figure 19.2 Effort or defect profile in the Putnam Model

PRODUCT-SPECIFIC MODELS (PSM)

- Product-specific info. Used (vs. none used in generalized models)
- Better accuracy/usefulness at cost ↑
- Three types:
  - semi-customized
  - observation-based
  - measurement-driven predictive
- Connection to generalized models (GMs):
  - Customize GMs to PSMs with new/refined models and additional data.
  - Generalize PSMs to GMs with empirical evidence and general patterns.

SEMI-CUSTOMIZED MODELS

- Semi-customized models:
  - Project level model based on history.
  - Data captured by phase.
  - Both projections and actual.
  - Linear extrapolation.
- The information from these semi-customized models can be directly used to predict defect distribution for the current release.

Table 19.2 DRM (defect removal model): defect distribution for previous releases of a product

Requirement	Design	Coding	Testing	Support
5%	10%	35%	40%	10%

PSM: OBSERVATION-BASED

- Observation-based models relate observations of the software system behavior to information about related activities to provide more precise quality assessments.
- Examples of such models include various software reliability growth models (SRGMs) where observed failures and associated time intervals are fitted to SRGMs to evaluate product reliability.
- Focus on the effect/observations
- Assumptions about the causes
- Assessment-centric
- Example: Goel-Okumoto SRGM
  - functional relation: $m(t) = N(1-e^{-bt})$
  - observed failures over time
  - curve fitting
  - reliability assessment/prediction
  - management decisions: exit criteria

PSM: PREDICTIVE

- Measurement-driven predictive models
  - Establish predictive relations between quality and other measurement on the historical data
  - Provide early prediction of quality , and identify problems early so that timely actions can be taken to improve product quality.
  - Modeling techniques: regression, TBM etc.
  - Risk assessment and management
- Model characteristics:
  - Response: chief concern
  - Predictors: observable/controllable
  - Linkage quantification

Table 19.3 High-defect modules for two products identified by tree-based

Product	Subset	#Modules	Mean-DF
LS	lrrr	16	9.81
	rlr	53	10.74
	rr	17	22.18
	whole product	1296	1.8
NS	rlll	8	55.0
	rr	5	77.0
	whole product	995	7.9

MODEL COMPARISON AND INTERCONNECTIONS

- Different types of quality assessment models and their relations can be compared by looking at their ability to provide useful information, their applicability to different project environments, and their inter-connections.
- we can compare the following:
  - Usefulness of the modeling results, in terms of how accurate the quality estimates are and the applicability of the models to different environments.
  - Model inter-connections, which can be examined in two opposite directions:
    - Customizution of generalized quality models to provide better quality estimates when product-specific information is available.
    - Generalization of product-specific models when enough empirical evidence from different products or projects is accumulated.

SUMMARY OF QUALITY ASSESSMENT MODELS AND THEIR APPLICATIONS

The usefulness of a model mainly depends on two factors:

- Is it applicable to the specific development environment and the specific product under development or maintenance?
- If so, how accurate the model is in its quality estimates?

Table 19.4 Summary of quality assessment models and their applications

Model Type	Sub-Type	Primary Result	Applicability
generalized models		rough quality estimates	all or by industry
	overall	overall product quality	across industries
	segmented	industry-specific quality	within an industry
	dynamic	quality trend over time	trend in all
product-specific quality models		better quality estimates	specific product
	semi-customized	quality extrapolation	prev→cur release
	observation-based	quality assessments	current product
	measurement-driven	quality predictions	both above

MODEL COMPARISON AND INTERCONNECTIONS

- This usefulness can be weighted against its cost, in particular, the cost of collecting the required measurement data, which is typically the dominant part of the modeling cost.
- Generalized models provide rough quality estimates based on empirical data from industry.
- Product-specific models provide more precise quality assessments using product-specific measurements.
- However, generalized models are more widely applicable and less expensive to use than product-specific models, because they do not require product-specific measurements.
- Consequently, generalized models may be more useful in the product planning stage, and in the early phases of product development, when most product-specific data are unavailable.

MODEL COMPARISON AND INTERCONNECTIONS

- One exception to this general rule is when there exist historical data for the previous releases of the current product. Semi-customized models can be used to provide better estimates under this situation.

- As development or maintenance activities progress, more measurement data can be collected and various detailed quality models in the category of product-specific models, such as observation-based models and measurement-based predictive models, can be used to better manage the QA activities as well as the overall software development or maintenance processes.

SOFTWARE TESTING METRICS

- **Software Testing Metrics** are the quantitative measures used to estimate the progress, quality, productivity and health of the software testing process.
- Metric defines in quantitative terms the degree to which a system, system component, or process possesses a given attribute. E.g. Total number of defects
- **Types of Test Metrics**
- **Process Metrics:** It can bae used to improve the process efficiency of the SDLC (Software Development Life Cycle)
- **Product Metrics:** It deals with the quality of the software product
- **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members

TEST METRICS GLOSSARY

- **Rework Effort Ratio =** (Actual rework efforts spent in that phase/ total actual efforts spent in that phase) X 100
- **Requirement Creep =** (Total number of requirements added/No of initial requirements)X100
- **Schedule Variance =** (Actual Date of Delivery – Planned Date of Delivery)
- **Cost of finding a defect in testing =** (Total effort spent on testing/ defects found in testing)
- **Schedule slippage =** (Actual end date – Estimated end date) / (Planned End Date – Planned Start Date) X 100
- **Passed Test Cases Percentage** = (Number of Passed Tests/Total number of tests executed) X 100
- **Failed Test Cases Percentage** = (Number of Failed Tests/Total number of tests executed) X 100
- **Blocked Test Cases Percentage** = (Number of Blocked Tests/Total number of tests executed) X 100
- **Fixed Defects Percentage** = (Defects Fixed/Defects Reported) X 100
- **Accepted Defects Percentage** = (Defects Accepted as Valid by Dev Team /Total Defects Reported) X 100
- **Defects Deferred Percentage** = (Defects deferred for future releases /Total Defects Reported) X 100
- **Critical Defects Percentage** = (Critical Defects / Total Defects Reported) X 100
- **Average time for a development team to repair defects** = (Total time taken for bugfixes/Number of bugs)
- **Number of tests run per time period** = Number of tests run/Total time
- **Test design efficiency** = Number of tests designed /Total time
- **Test review efficiency** = Number of tests reviewed /Total time
- **Bug find rote or Number of defects per test hour** = Total number of defects/Total number of test hours

SOFTWARE QUALITY MODELS

- Software Quality Models are a standardised way of measuring a software product.
- With the increasing trend in software industry, new applications are planned and developed everyday.
- This eventually gives rise to the need for reassuring that the product so built meets at least the expected standards.
- During this course, we have discussed the following main categories of software quality models:
- Quality Definition Models:
  - McCall
  - FURPS
  - ISO 9126
  - GQM
  - CMMI
- Your task is to compare the above models and provide a critical analysis based on comparisons.

SOFTWARE CONFIGURATION MANAGEMENT

- **Configuration Management is the *control* of the evolution of complex software.**
- **SCM is the discipline that enable us to keep evolving software products under control, and thus contributes to satisfying quality and delay constraints.**
- **There is a need for storing the different components of a software product (artifacts) and all their versions safely.**

SCM DEFINITION

- A discipline applying technical and administrative direction and surveillance to:
  - **Identify and document the functional and physical characteristics of a configuration item**
  - **Control changes to those characteristics**
  - **Record and report change processing and implementation status**
  - **Verify compliance with specified requirements**

Definition from IEEE

SOFTWARE CONFIGURATION MANAGEMENT

- **In software engineering, software configuration management (SCM) is the task of *tracking* and *controlling* changes in the software.**
- **SCM practices include *revision control* and the establishment of *baselines*.**
- **If something goes wrong, SCM can determine *what was* changed and *who* changed it**

VERSION CONTROL

- A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information.
- Changes are usually identified by a number termed the "revision number", "revision level", or simply "revision".
- For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on.
- Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

SOFTWARE CONFIGURATION MANAGEMENT

- **When you develop software, change is inevitable.**
- **Change increase the level of confusion among development team.**
- **Confusion arise when changes are not analyzed before they made, recorded before they are implemented.**

SOFTWARE CONFIGURATION MANAGEMENT

Why client is running the wrong version of the software . . .

Oops! a bug that was fixed in the software suddenly reappears . . .

I don't know why a developed and tested functionality is missing . . .

SCM CONCEPTS

- **Configuration**
- **Configuration Control**
- **Software Configuration Item (SCI)**
- **Baselines**
- **Version (or variants)**

SCM CONCEPTS

Configuration

- **The functional and *physical characteristics* of hardware or software as set forth in technical documentation or achieved in a product.** **(ANSI/IEEE 828, 1990)**

SCM CONCEPTS

Configuration Control

- **An element of configuration management, consisting of evaluation, coordination, approval or disapproval and implementation of changes to configuration items *after formal establishment* of their configuration identifications.**

(ANSI/IEEE 828,1990)

SCM CONCEPTS

- **Configuration Item**

"An aggregation of hardware, software, or both, that is designated for configuration management and treated as a single entity in the configuration management process."

- **Software configuration items are not only program code segments but all type of documents according to development, e.g**
  - all type of code files
  - drivers for tests
  - analysis or design documents
  - user or developer manuals
  - system configurations (e.g. version of compiler used)

- **In some systems, not only software but also hardware configuration items (CPUs, bus speed frequencies) exist!**

34

SCM CONCEPTS

Baseline

- **Baseline is a specification or product that has been formally *reviewed* and *agreed* upon, that thereafter serves as the basis for further development and that can be changed only through formal change procedures.**

- **e.g. SRS, Design Doc, Source Code etc**

SOFTWARE CONFIGURATION MANAGEMENT - WHY

Management

- New versions of software systems are created as they change
  - For different machines/OS
  - Offering different functionality
  - Tailored for particular user requirements

- Configuration management is concerned with managing evolving software systems
  - System change is a team activity
  - CM aims to control the costs and effort involved in making changes to a system

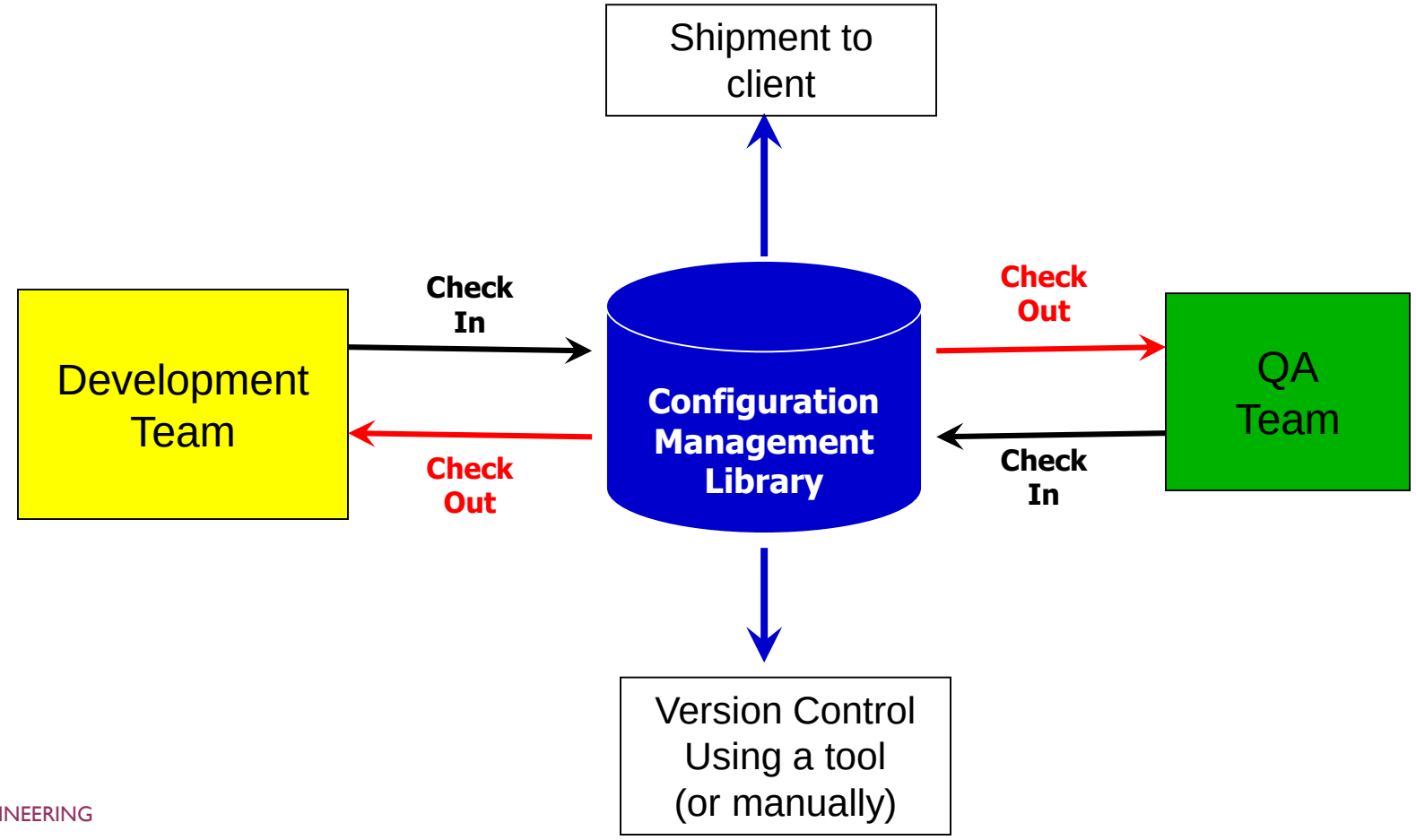
SOFTWARE CONFIGURATION MANAGEMENT – WHY?

- **Involves the development and application of procedures and standards to manage an evolving software product**
- **May be seen as part of a more general quality management process**
- **When released to CM, software systems are sometimes called baselines as they are a starting point for further development**

CONFIGURATION MANAGEMENT LIBRARY

- **The CM Library describes the central storage facility for all CI's.**
- **The CM Library contains the;**
  - **current baseline project,**
  - **approved documentation,**
  - **artifact files to keep the project running smoothly**

SCM CONCEPT


Shipment to client
Development Team
Check In
Check Out
Configuration Management Library
Check Out
Check In
QA Team
Version Control Using a tool (or manually)

SCM ACTIVITIES

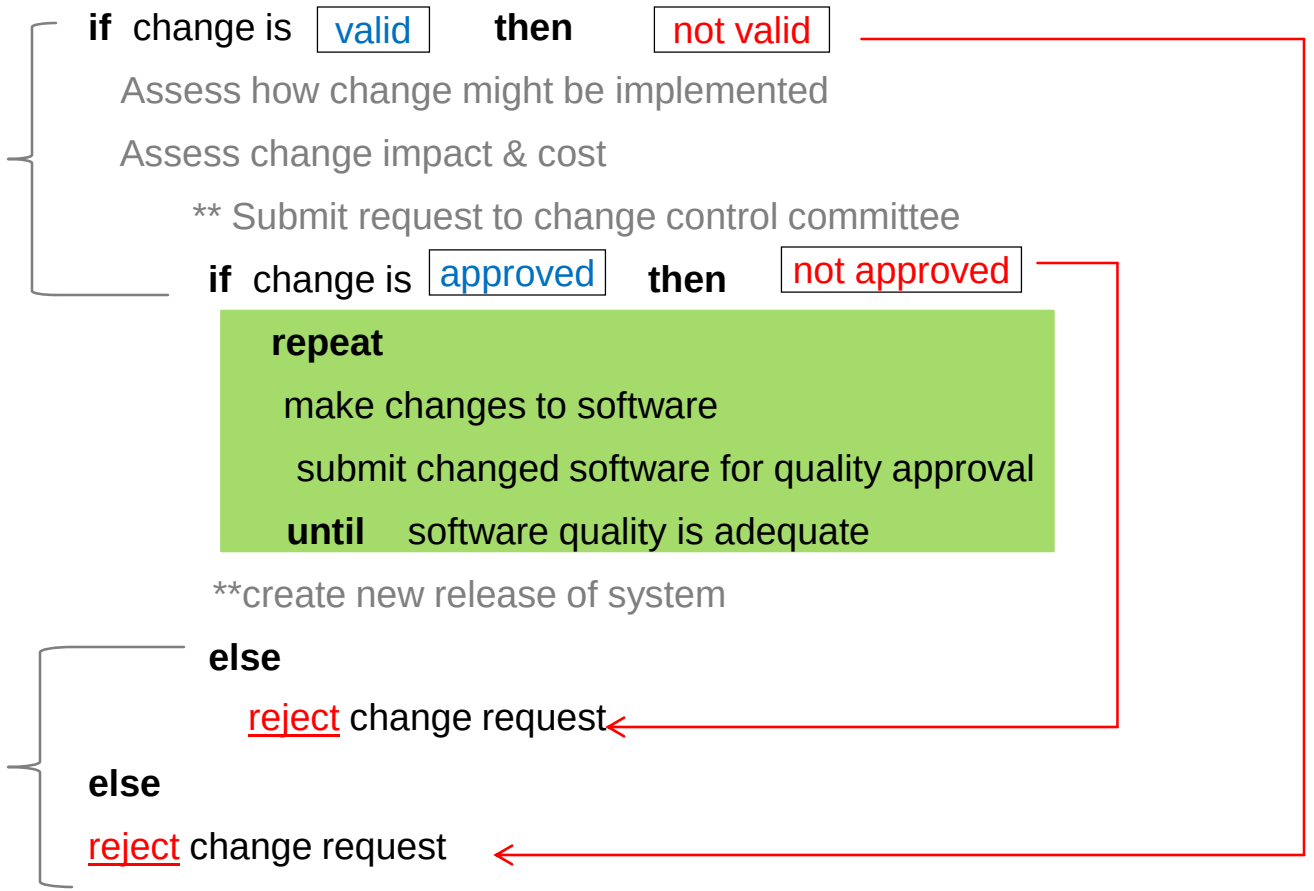
1. **Identify the changes**
2. **Control the change**
3. **Ensure that change is properly implemented**
4. **Report changes to others involved in the project**

CHANGE MANAGEMENT

- **Software systems are subject to continual *change requests***
  - **From users**
  - **From developers**
  - **From market forces**

- **Change management is concerned with keeping track of these changes and ensuring that they are implemented in the most cost-effective way**

CHANGE MANAGEMENT PROCESS



SOFTWARE CHANGE STRATEGIES

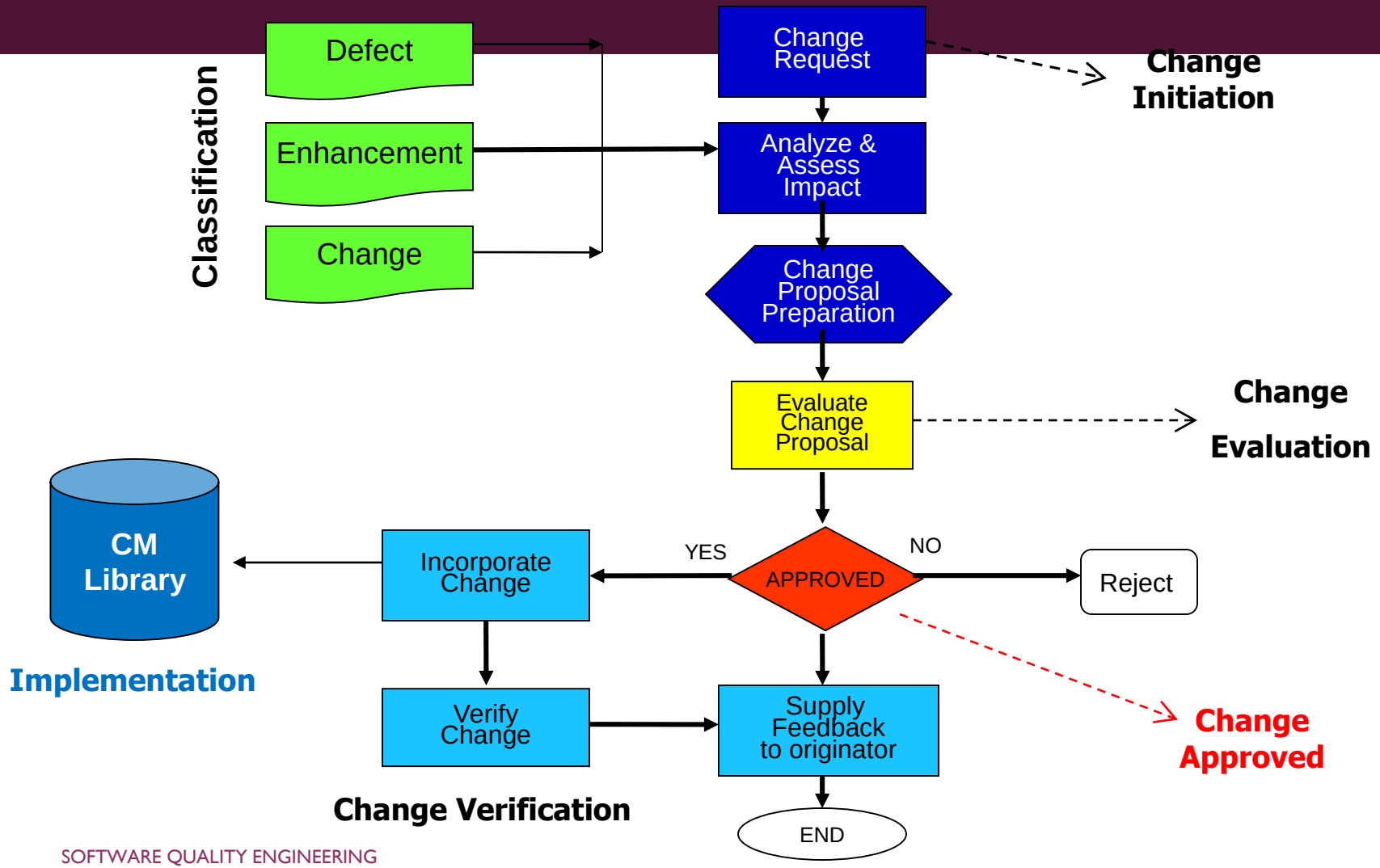
- **The software change strategies that could be applied separately or together are:**
- **Software maintenance:The changes are made in the software due to requirements.**

  Types of maintenance?
- **Architectural transformation: It is the process of changing one architecture into another form.**
- **Software re-engineering: New feature can be added to existing system and then the system is reconstructed for better use of it in future.**

TYPES OF MAINTENANCE

- Maintenance is defined as the process in which changes are implemented by either modifying the existing system's architecture or by adding new components to the system.
- **Corrective Maintenance**: Means the maintenance for correcting the software faults.
- **Adaptive maintenance**; Means maintenance for adapting the change in environment.
- **Perfective maintenance**: Means modifying or enhancing the system to meet the new requirements.
- **Preventive maintenance**: Means changes made to improve future maintainability.

CHANGE CONTROL



That is all