UNIT IV
TESTING AND MAINTENANCE

Software testing fundamentals-Internal and external views of Testing-white box testing-basis path testing-control structure testing-black box testing- Regression Testing – Unit Testing – Integration Testing – Validation Testing – System Testing And Debugging – Software Implementation Techniques: Coding practices-Refactoring-Maintenance and Reengineering-BPR model-Reengineering process model-Reverse and Forward Engineering.

4.1 SOFTWARE TESTING FUNDAMENTALS

- Software testing is a critical element of software quality assurance and represents the ultimate review of specification design and code generation.
- Testing involves exercising the program using data like the real data processed by unexpected system outputs.

"**Software testability** is simply how easily a computer program can be tested." The following characteristics lead to testable software.

- Operability
- Observability
- Controllability
- Decomposability
- Simplicity
- Stability
- Understandability

Test Characteristics

Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

- A good test has a high probability of finding an error
- A good test is not redundant
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex

4.2 INTERNAL AND EXTERNAL VIEWS OF TESTING

There are two views of the testing.

- **Internal view(white-box testing)**
- **External view(Black-box testing)**
  - **Black-box testing** alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.
  - **White-box testing** of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

4.3 WHITE-BOX TESTING

- White-box testing, sometimes called glass-box testing, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- Using white-box testing methods, can derive test cases that

  (1) Guarantee that all independent paths within a module have been exercised at least once,

  (2) exercise all logical decisions on their true and false sides,

  (3) Execute all loops at their boundaries and within their operational bounds, and

  (4) Exercise internal data structures to ensure their validity.
- The structural testing is also called as white box testing
- In structural testing derivation of test cases is according to program structure.

  The basis path testing technique is one of a number of techniques for control structure testing.

4.3.1 BASIS PATH TESTING

4.3.2 Steps

- Basis path testing is a white-box testing technique.
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

4.3.2.1 Flow Graph Notation

- A simple notation for the representation of control flow, called a flow graph (or program graph.
- The flow graph depicts logical control flow using the notation illustrated in Figure 4.1

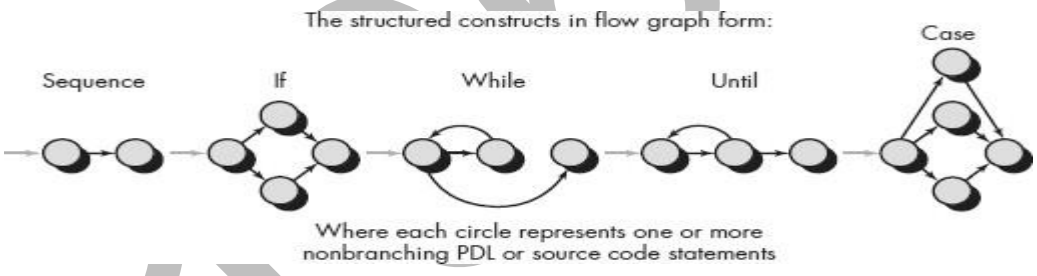


Figure 4.1 Flow Graph Notation

Flow graph notation

- Consider the procedural design representation
- A flowchart is used to depict program control structure.
- It maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).
- Each circle, called a flow graph node, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node.
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.

- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.Referring to Figure 4.3 the **program design language** (PDL) segment translates into the flow graph shown.

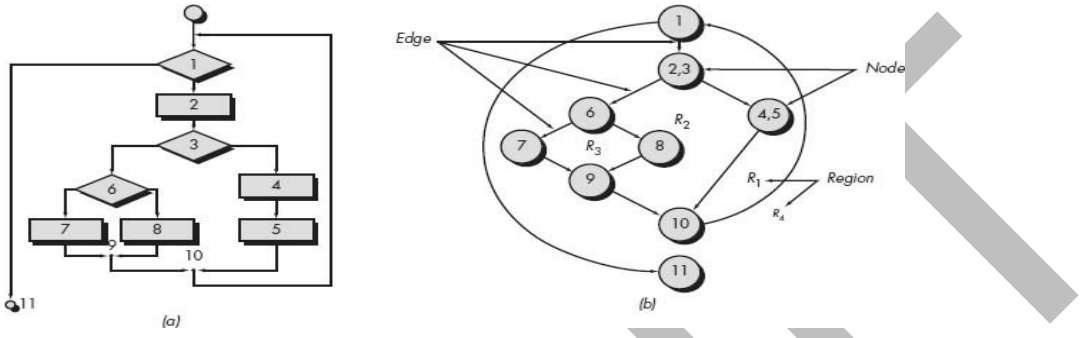


Figure 4.2 a) Flowchart and (b) flow graph

- Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a **predicate node** and is characterized by two or more edges emanating from it.

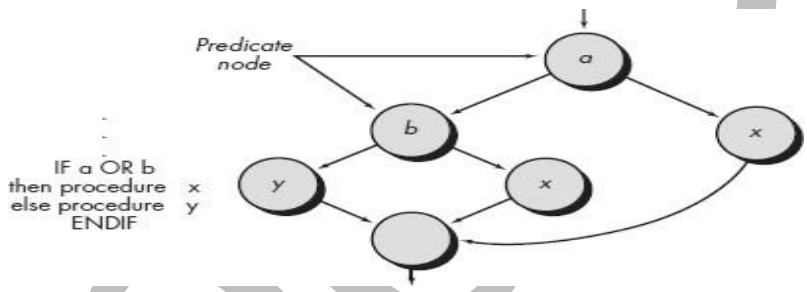


Figure 4.3 Compound logic

4.3.2.2 Independent Program Paths

- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.
- An independent path must move along at least one edge that has not been traversed before the path is defined.
- For example, a set of independent paths for the flow graph

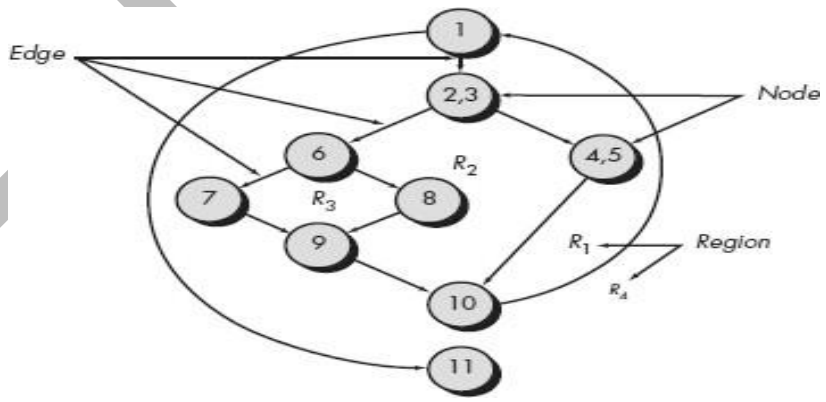


Figure 4.4 Example for Independent Program Paths

Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11
Note that each new path introduces a new edge. The path 1-
2-3-4-5-10-1-2-3-6-8-9-10-1-11
Paths 1 through 4 constitute a **basis set** for the flow graph.
Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program.Complexity is computed in one of three ways:
Definition
1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity V(G) for a flow graph G is defined
as V(G) = E - N + 2
where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as
V(G) = P +1
where P is the number of predicate nodes contained in the flow graph G.
The cyclomatic complexity can be computed using each of the algorithms just noted:
1.The flow graph has four regions.
2. V(G) = 11 edges - 9 nodes + 2 = 4.
3. V(G) =3 predicate nodes + 1 = 4.
Therefore, the cyclomatic complexity of the flow graph in Figure 4.4

4.3.2.3 Graph Matrices

- A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- A simple example of a flow graph and its corresponding graph matrix is shown inFigure 4.5.

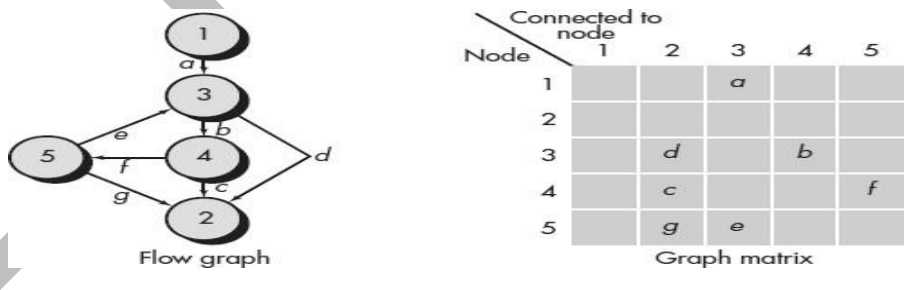


Figure 4.5 a) Flow graph b) Graph matrix

4.3.3 Control Structure Testing:

4.3.3.1 Condition Testing

- Condition testing is a test-case design method that exercises the logical conditions contained in a

- program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

  E1 <relational-operator> E2

  where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥. A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses.

4.3.3.2 Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.
- For a statement with S as its statement number,
- DEF(S) = {X | statement S contains a definition of X}
- USE(S) = {X | statement S contains a use of X}

4.3.3.3 Loop Testing

- **Loop testing** is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: **simple loops, concatenated loops, nested loops, and unstructured loops** as shown in Figure 4.6

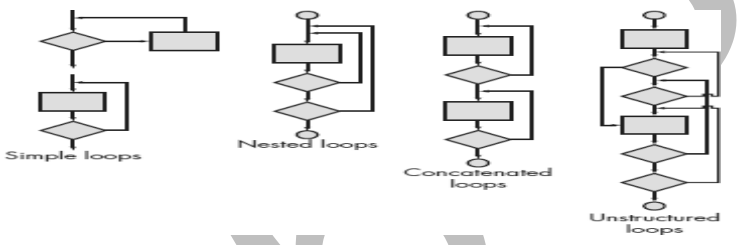


Figure 4.6 Classes of Loops

Classes of Loops

- **Simple loops:** The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

  1. Skip the loop entirely.
  2. Only one pass through the loop.
  3. Two passes through the loop.
  4. m passes through the loop where m < n.
  5. n -1, n, n + 1 passes through the loop.

- **Nested loops**

  Beizer suggests an approach that will help to reduce the number of tests:

  1. Start at the innermost loop. Set all other loops to minimum values.
  2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
  3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
  4. Continue until all loops have been tested.

- **Concatenated loops**

  Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.

- **Unstructured loops**

  Class of loops should be redesigned to reflect the use of the structured programming constructs.

4.4 BLACK-BOX TESTING

- Black-box testing, also called **behavioral testing,** focuses on the functional requirements of the software. That is, black-box testing techniques enable to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:
  (1) Incorrect or missing functions,
  (2) Interface errors,
  (3) Errors in data structures or external database access,
  (4) Behavior or performance errors, and
  (5) Initialization and termination errors.

4.4.1 Graph-Based Testing Methods

- The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects as shown in Figure

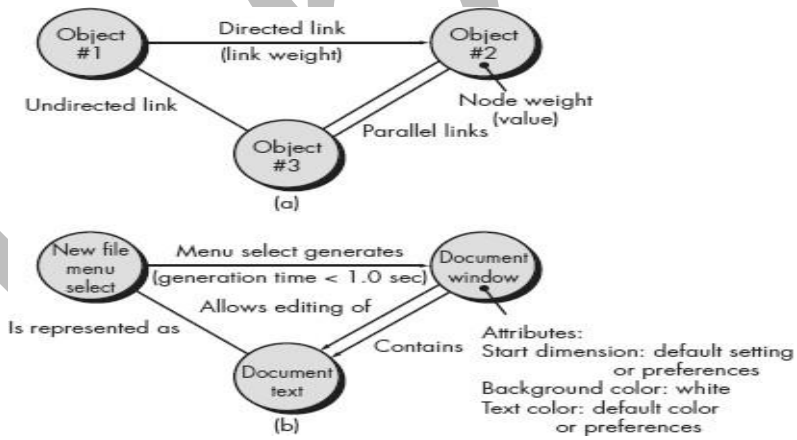


Figure 4.7(a) Graph notation; (b) simple example

- The symbolic representation of a graph is shown in Figure 4.7
- Nodes are represented as circles connected by links that take a number of different forms.
- A directed link (represented by an arrow) indicates that a relationship moves in only one direction.
- A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions.
- Parallel links are used when a number of different relationships are established between graph nodes.

4.4.2 Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.
- An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

4.4.3 Boundary Value Analysis

- Boundary value analysis leads to a selection of test cases that exercise bounding values.
- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

By applying these guidelines, boundary testing will be more complete.

4.4.4 Orthogonal Array Testing

- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.

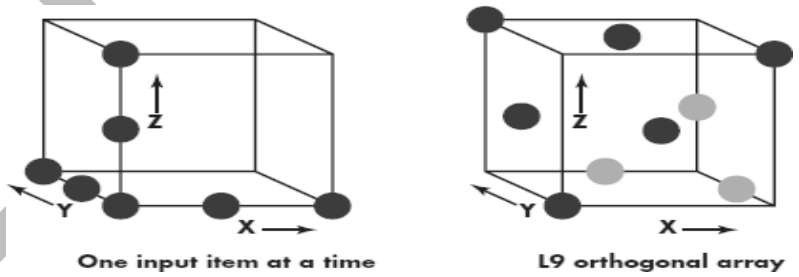


Figure 4.8 A geometric view of test cases Source

4.5 REGRESSION TESTING

- Regression testing is the re execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Regression testing is the activity that helps to ensure that changes do not introduce unintended

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
- Ensures that changes have not propagated unintended side effects
- Helps to ensure that changes do not introduce unintended behavior or additional errors
- May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed
- Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to reexecute every test for every program function once a change has occurred.

4.6 UNIT TESTING

- Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

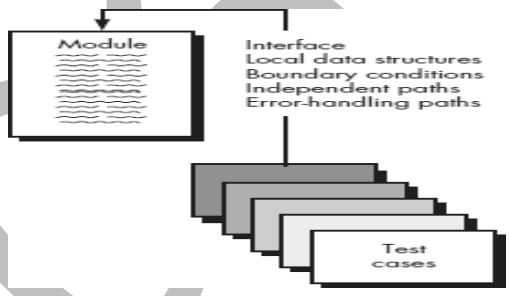


Figure 4.9 Unit test

Unit-test considerations

- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.
- Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Unit-test procedures:

- Unit testing is normally considered as an adjunct to the coding step.
- The design of unit tests can occur before coding begins or after source code has been generated.
- A review of design information provides guidance for establishing test cases that are likely to uncover errors.
- A component is not a stand-alone program; driver and/or stub software must often be developed for each unit test.
- The unit test environment is illustrated in Figure. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Unit testing is simplified when a component with high cohesion is designed.
- When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

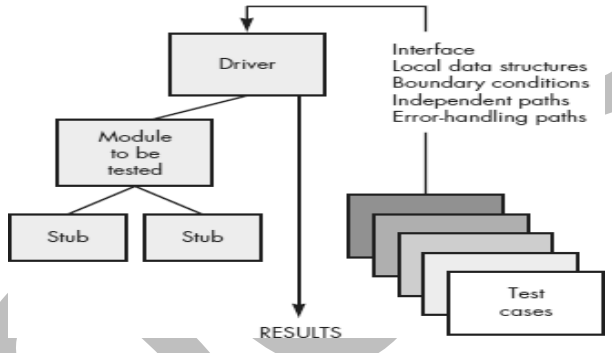


Figure 4.10 Unit-test environment

4.7 INTEGRATION TESTING

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that has been dictated by design.
- There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole.
- A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.
- The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Top-down integration:

- Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.
- Depth-first integration integrates all components on a major control path of the program structure.
- Selection of a major path is arbitrary and depends on application-specific characteristics.
- Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.
- The integration process is performed in a series of five steps:

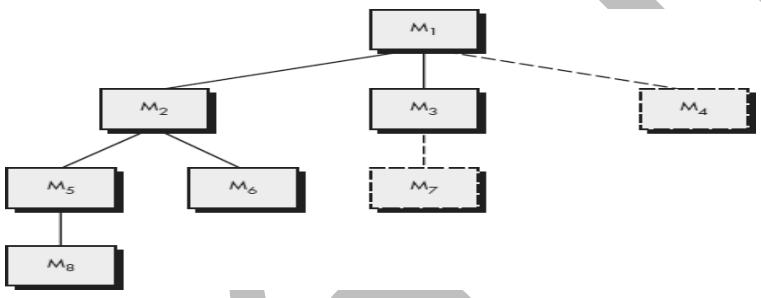


Figure 4.11 Top-down integration

Top-down integration

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The top-down integration strategy verifies major control or decision points early in the test process.

Bottom-up integration:

- Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program Structure.

- Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block).
- Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma.
- Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

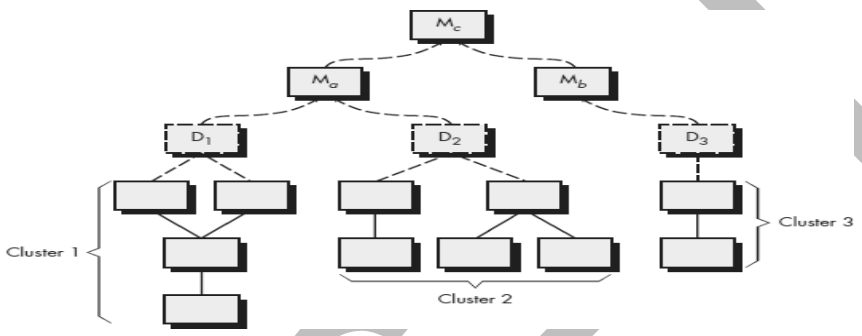


Figure 4.12 Bottom-up integration

- If the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

4.8 VALIDATION TESTING

- Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected.
- At the validation or system level, the distinction between conventional software, object-oriented software, and WebApps disappears.

4.8.1 Validation-Test Criteria

- Software validation is achieved through a series of tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., trans-portability, compatibility, error recovery, maintainability).
- After each validation test case has been conducted, one of two possible conditions exists:
  (1) The function or performance characteristic conforms to specification and is accepted or
  (2) a deviation from specification is uncovered and a deficiency list is created. Deviations or errors discovered at this stage in a project can rarely be corrected prior to scheduled delivery.

4.8.2 Configuration Review

- An important element of the validation process is a **configuration review.**
- The intent of the review is to ensure that all elements of the software configuration have been properly developed, are catalogued, and have the necessary detail to bolster the support activities.
- The configuration review, sometimes called an **audit.**

4.8.3 Alpha and Beta Testing

- Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

Alpha Testing

- The alpha test is conducted at the developer's site by a representative group of end users.
- The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.

Beta Testing

- The beta test is conducted at one or more end-user sites.
- Unlike alpha testing, the developer generally is not present.
- Therefore, the beta test is a **"live" application** of the software in an environment that cannot be controlled by the developer.
- The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
- As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base.
- A variation on beta testing, called **customer acceptance testing,** is sometimes performed when custom software is delivered to a customer under contract.

4.9 SYSTEM TESTING

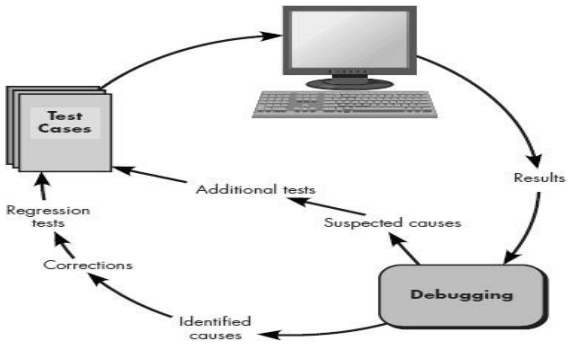
- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.
- Various types of system tests are
- Recovery Testing
- Security Testing
- Stress Testing
- Performance Testing
- Deployment Testing

4.9.1 Recovery Testing

- Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed by the system itself), re initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness.

4.9.2 Security Testing

- Security testing attempts to verify that protection mechanisms built into a system, in fact, protect it from improper penetration.



- The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

4.9.3 Stress Testing

- Stress tests are designed to confront programs with abnormal situations.
- Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- A variation of stress testing is a technique called **sensitivity testing.**
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

4.9.4 Performance Testing

- Performance testing is designed to test the run-time performance of software within the context of an integrated system.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.
- That is, it is often necessary to measure resource utilization.

4.9.5 Deployment Testing

- Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.
- In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

4.10 DEBUGGING

- Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

4.10.1 The Debugging Process

- Debugging is not testing but often occurs as a consequence of testing.
- The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.

- The debugging process will usually have one of two outcomes:
  (1) The cause will be found and corrected or
  (2) The cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Why is debugging so difficult?

1. The symptom and the cause may be geographically remote.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

4.10.2 Psychological Considerations

- Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.
- Commenting on the human aspects of debugging, Shneiderman [Shn80] states:
- Debugging is one of the more frustrating parts of programming.
- Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately corrected.

4.10.3 Debugging Strategies

In general, three debugging strategies have been proposed
(1) brute force,
(2) backtracking, and
(3) cause elimination.
Each of these strategies can be conducted manually, but modern debugging tools can make the process much more effective.

Debugging tactics.

- ✓ The **brute forcecategory** of debugging is probably the most common and least efficient method for isolating the cause of a software error.
- ✓ **Backtracking** is a fairly common debugging approach that can be used successfully in small programs.

i) Beginning at the site where a symptom has been uncovered, the source codeistraced backward (manually) until the cause is found.
ii) Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

- ✓ The third approach to debugging—**cause elimination**—is manifested by induction or deduction and introduces the concept of binary partitioning.

Automated debugging:

- Each of these debugging approaches can be supplemented with debugging tools.
- A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available.

4.10.4 Correcting the Error

- Once a bug has been found, it must be corrected.
- Van Vleck suggests three simple questions that you should ask before making the "correction" that removes the cause of a bug:

➢ Is the cause of the bug reproduced in another part of the program?
➢ What "next bug" might be introduced by the fix I'm about to make?
➢ What could we have done to prevent this bug in the first place?

4.11 SOFTWARE IMPLEMENTATION TECHNIQUES

- After detailed system design we get a system design which can be transformed into implementation model.
- The goal coding is to implement the design in the best possible manner. Coding affects both testing and maintenance very deeply.
- The coding should be done in. such a manner that the instead of getting the job of programmer simplified the task of testing and maintenance phase should get simplified.
- Various objectives of coding are –
  1. Programs developed in coding should be readable.
  2. They should execute efficiently.
  3. The program should utilize less amount of memory.
  4. The programs should not be lengthy.
- If the objectives are clearly specified before the programmers then while coding they try to achieve the specified objectives.
- To achieve these objectives some programming principles must be followed.

4.11.1 Coding Practices

There are some commonly used programming practices that help in avoiding the common errors. These are enlisted below –

1. Control construct

- The **single entry** and **single exit** constructs need to be used.
- The standard control constructs must be used instead of using wide variety of controls.

2.Use of gotos

- The goto statements make the program unstructured and it also imposes overhead on compilation process.

3. Information hiding

- Information hiding should be supported as far as possible.
- In that case only access functions to the data structures must be made visible and the information present in it must be hidden.

4. Nesting

- Nesting means defining one structure inside another.
- If the nesting is too deep then it becomes hard to understand the code. Hence as far as possible - avoid deep nesting of the code.

Removing of nesting

```
while(condition)
{
if condition then statement
else if condition then statement
else if condition then statement
}
```

```
while(condition)
{
if condition then statement if
condition then statement if
condition then statement
}
```

5.User Defined data type

- Modern programming languages allow the user to use defined data types as the enumerated types.
- Use of user defined data types enhances the readability of the code.

For example: In C we can define the name of the days using enumerated datatype -

```
enum Day
{
Sunday;
Monday;
Tuesday;
Wednesday;
Thursday;
Friday;
Saturday;
}workday;
enum Day Today=Friday;
```

6. Module size

There is no standard rule about the size of the module but the large size of the module will not be functionally cohesive.

7. Module interface

- Complex module interface must be carefully examined.
- A simple rule of thumb is that the module interface with more than five parameters must be broken into multiple modules with simple interface.

8. Side effects

- Avoid obscure side effects.
- If some part of the code is changed randomly then it will cause some side effect.
- For example if number of parameters passed to the function is changed then it will be difficult to understand the purpose of that function.

9. Robustness

- The program is said to robust if it does something even though some unexceptional condition occurs.
- In such situations the programs do not crash but it exits gracefully.

10. Switch case with defaults

- The choice being passed to the switch case statement may have some unpredictable value, and then the default case will help to execute the switch case statement without any problem.
- Hence it is a good practice to always have default case in the switch statement.

11. Empty catch block

- If the exception is caught but if there is no action then it is not a good practice.
- Therefore take some default action even if it is just writing some print statement, whenever the exception is caught.

12. Empty if and while statements

- In if and while statements some conditions are checked.
- Hence if we write some empty block on checking these conditions then those checks are proved to be useless checks.
- Such useless checks should be avoided.

13. Check for read return

- Many times the return values that we obtain from the read functions are not checked because we blindly believe that the desired result is present in the corresponding variable when the read function is performed.
- But this may cause some serious errors.
- Hence the return value must be checked for the read operation.

14. Return from Finally Block

- The return value must come from finally block whenever it is possible.
- This helps in distinguishing the data values that are returning from the try-catch statements.

15. Trusted Data sources

- Counter check should be made before accessing the input data.
- For example while reading the data from the file one must check whether the data accessed is NULL or not.

16. Correlated Parameters

- Many often there occurs co-relation between the data items.
- It is a good practice to check these co-relations before performing any operation on those data items.

17. Exceptions Handling

- If due to some input condition if the program does not follow the main path and follows an exceptional path.
- In such a situation, an exceptional path may get followed. In order to make the software more reliable, it necessary to write the code for execution.

1. **Naming Conventions**

Following are some commonly used naming conventions in the coding

- Package name and variable names should be in lower case.
- Variable names must not begin with numbers.
- The type name should be noun and it should start with capital letter.
- Constants must be in upper case (For example PI, SIZE)
- Method name must be given in lower case.
- The variables with large scope must have long name. For example count_total, sum, Variables with short scope must have short name. For example i,j.
- The prefix is must be used for Boolean type of variables. For example isEmpty or isFull

2. **Files**

Reader must get an idea about the purpose of the file by its name. In some programming language like Java –

- The file extension must be Java.
- The name of the file and the class defined in the file must have the same na
- Line length in the file must be limited to 80 characters.

3. **Commenting/Layout**

Comments are non-executable part of the code. But it is very important because it enhances the readability of the code. The purpose of the code is to explain the logic of the program.

- Single line comments must be given by //
- For the names of the variables comments must be given.
- A black of comment must be enclosed within /* and */.

4. **Statements**

There are some guidelines about the declaration and executable statements.

- Declare some related variables on same line and unrelated variables on another line.
- Class variable should never be declared public.
- Make use of only loop control within the for loop.
- Avoid make use of break and continue statements in the loop.
- Avoid complex conditional expressions. Make use of temporary variables instead.
- Avoid the use of do...while statement.

Advantages of coding standards:

1. Coding standard brings uniform appearance in system implementation.
2. The code becomes readable and hence can be understood easily.
3. The coding standard helps in adopting good programming practices.

4.11.2 Code refactoring:

- **Code refactoring** is the process of restructuring existing computer code – changing the factoring – without changing its external behavior.
- Refactoring improves nonfunctional attributes of the software.
- Advantages include improved code readability and reduced complexity to improve source code maintainability, and create a more expressive internal architecture or object model to improve extensibility.

- If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.
- Typically, refactoring applies a series of standardized basic micro-refactoring, each of which is (usually) a tiny change in a computer's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements.
- Many development environments provide automated support for performing the mechanical aspects of these basic refactoring.

There are two general categories of benefits to the activity of refactoring.

1. **Maintainability**.

- It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.
- This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods.
- It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.

2. **Extensibility**.

- It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.

4.11.2.1 List of Refactoring Techniques

- Here are some examples of micro-refactoring; so some of these may only apply to certain languages or language types.
- A longer list can be found in Fowler's Refactoring bookand on Fowler's Refactoring Website.
- Many development environments provide automated support for these micro-refactoring.
- For instance, a programmer could click on the name of a variable and then select the "Encapsulate field" refactoring from a context menu.
- The IDE would then prompt for additional details, typically with sensible defaults and a preview of the code changes.
- After confirmation by the programmer it would carry out the required changes throughout the code.

4.11.3.2. Techniques that allow for more abstraction

- Encapsulate Field – force code to access the field with getter and setter methods
- Generalize Type – create more general types to allow for more code sharing
- Replace type-checking code with State/Strategy
- Replace conditional with polymorphism

4.11.3.3. Techniques for breaking code apart into more logical pieces

- Componentization breaks code down into reusable semantic units that present clear, well- defined, simple-to-use interfaces.
- Extract Class moves part of the code from an existing class into a new class.
- Extract Method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.

- Rename Method or Rename Field – changing the name into a new one that better reveals its purpose
- Pull Up – in OOP, move to a superclass
- Push Down – in OOP, move to a subclass

4.12 MAINTENANCE AND REENGINEERING

4.12.1Software Maintenance

The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Maintenance is thus concerned with

- correcting errors found after the software has been delivered
- adapting the software to changing requirements, changing environments, ...

4.12.1.1 Types of Software Maintenance

- In a software lifetime, types of maintenance may vary based on its nature.
- It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature.
- Following are some types of maintenance based on their characteristics:

a) Corrective Maintenance
b) Adaptive Maintenance
c) Perfective Maintenance
d) Preventive Maintenance

a) Corrective Maintenance

- This includes modifications and updations done in order to **correct or fix problems**, which are either discovered by user or concluded by user error reports.
- Corrective maintenance deals with the repair of faults or defects found in day-to-day system functions.
- A defect can result due to errors in software design, logic and coding.
- Design errors occur when changes made to the software are incorrect, incomplete,wrongly communicated, or the change result is misunderstood.

b) Adaptive Maintenance

- This includes modifications and updations applied to keep the software product up-to-date and tuned to the ever changing world of technology and business environment
- Adaptive maintenance is the implementation of changes in a part of the system, which has been affected by a change that occurred in some other part of the system.
- It consists of **adapting software to changes in the environment such as the hardware or the operating system.**

c) Perfective Maintenance:

- This includes modifications and updations in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.

- Perfective maintenance involves making functional enhancements to the system in addition to the activities to increase the system's performance even when the changes have not been suggested by faults.
- This includes enhancing both the function and efficiency of the code and changing the functionalities of the system as per the users' changing needs.

d) Preventive Maintenance:

- This includes modifications and updations to prevent future problems of the software.
- It aims to attend problems, which are not significant at this moment but may cause serious issues in future.
- Preventive maintenance involves performing activities to prevent the occurrence of errors.
- It tends to reduce the software complexity thereby **improving program understandability and increasing software maintainability**.
- It comprises documentation updating, code optimization, and code restructuring.
- Documentation updating involves modifying the documents affected by the changes in order to correspond to the present state of the system.

4.12.1.2 Major cause for maintenance problem

- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation

4.12.1.3 Maintenance Control

- Configuration control:
- Identify, classify change requests
- Analyze change requests
- Implement changes
- Fits in with ***iterative enhancement model*** of maintenance (first analyze, then change)
- As opposed to ***quick-fix model*** (first patch, then update design and documentation, if time permits)

4.12.2 Re-Engineering

- **When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering.**
- It is a thorough process where the design of software is changed and programs are re-written.
- Legacy software cannot keep tuning with the latest technology available in the market.
- As the hardware become obsolete, updating of software becomes a headache.
- Even if software grows old with time, its functionality does not.
- For example, initially UNIX was developed in assembly language. When language C came into existence, UNIX was re-engineered in C, because working in assembly language was difficult.
- Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.

When to re-engineer?

- When system changes are mostly confined to part of the system then re-engineer that part.
- When hardware or software support becomes obsolete.
- When tools to support re-structuring are available.

Advantages of Re-Engineering:

- ***Reduced risk***
  There is a high risk in new software development. There may be development problems, staffing problems and specification problems
- ***Reduced cost***
  The cost of re-engineering is often significantly less than the costs of developing new software.

4.13 BPR MODEL (BUSINESS PROCESS RE-ENGINEERING):

- Concerned with re-designing business processes to make them more responsive and more efficient.
- Often reliant on the introduction of new computer systems to support the revised processes
- May force software re-engineering as the legacy systems are designed to support existing processes

4.14 RE-ENGINEERING PROCESS MODEL Steps involved in Re-Engineering Process

Decide what to re-engineer. Is it whole software or a part of it?
Perform Reverse Engineering, in order to obtain specifications of existing software. **Restructure** Program if required. For example, changing function-oriented programs into object-oriented programs. Re-structure data as required.
Apply Forward engineering concepts in order to get re-engineered software.

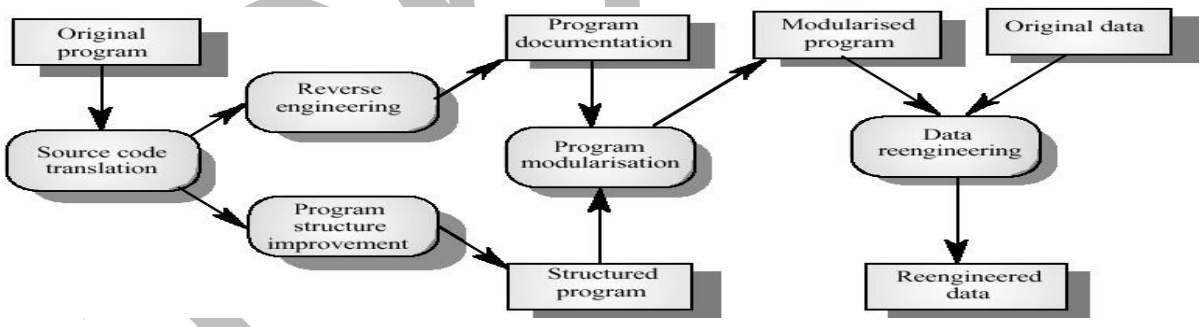


Figure 4.12 Re-Engineering Process

I Source Code Translation

- Involves converting the code from one language (or language version) to another
- May be necessary because of:
  - Hardware platform update
  - Staff skill shortages
  - Organisational policy changes
- Only realistic if an automatic translator is available.

II Reverse Engineering

- Analysing software with a view to understanding its design and specification

- May be part of a re-engineering process but may also be used to re-specify a system for re-implementation.

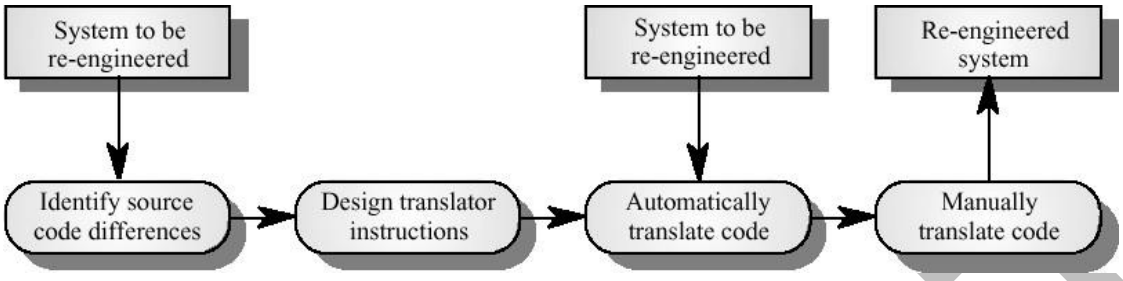


Figure 4.13 Program Translation Process

- Builds a program data base and generates information from this.
- Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process.

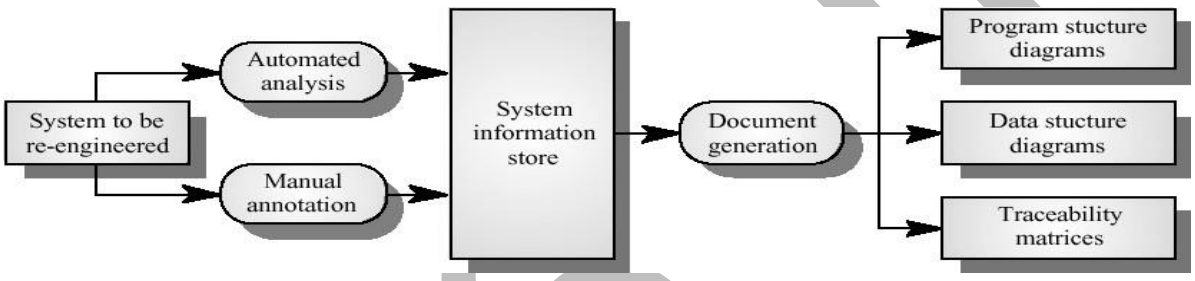


Figure 4.14Reverse Engineering Process

- Reverse engineering often precedes re-engineering but is sometimes worthwhile in its own right
  - The design and specification of a system may be reverse engineered so that they can be an input to the requirements specification process for the system's replacement.
  - The design and specification may be reverse engineered to support program maintenance.

III Program Structure Improvement

- Maintenance tends to corrupt the structure of a program. It becomes harder and harder to understand
- The program may be automatically restructured to remove unconditional branches
- Conditions may be simplified to make them more readable

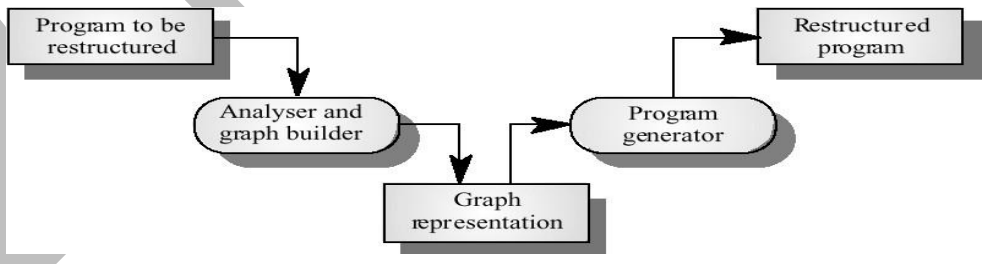


Figure 4.15 Automatic program restructuring

IV Restructuring problems

- Problems with re-structuring are:

- Loss of comments
- Loss of documentation
- Heavy computational demands

- Restructuring doesn't help with poor modularisation where related components are dispersed throughout the code
- The understandability of data-driven programs may not be improved by re-structuring

V Program modularisation

- The process of re-organising a program so that related program parts are collected together in a single module
- Usually a manual process that is carried out by program inspection and re-organisation

Module Types

- Data abstractions
  - Abstract data types where datastructures and associated operations are grouped
- Hardware modules
  - All functions required to interface with a hardware unit
- Functional modules
  - Modules containing functions that carry out closely related tasks
- Process support modules
  - Modules where the functions support a business process or process fragment

Recovering data abstractions

- Many legacy systems use shared tables and global data to save memory space
- Causes problems because changes have a wide impact in the system
- Shared global data may be converted to objects or ADTs
  - Analyse common data areas to identify logical abstractions
  - Create an ADT or object for these abstractions
  - Use a browser to find all data references and replace with reference to the data abstraction.

VI Data Re-Engineering

- Involves analysing and reorganising the data structures (and sometimes the data values) in a program
- May be part of the process of migrating from a file-based system to a DBMS-based system or changing from one DBMS to another
- Objective is to create a managed data environment

Approaches to data re-engineering

Approach	Description
Data cleanup	The data records and values are analysed to improve their quality. Duplicates are removed, redundant information is deleted and a consistent format applied to all records. This should not normally require any associated program changes.
Data extension	In this case, the data and associated programs are re-engineered to remove limits on the data processing. This may require changes to programs to increase field lengths, modify upper limits on the tables, etc. The data itself may then have to be rewritten and cleaned up to reflect the program changes.
Data migration	In this case, data is moved into the control of a modern database management system. The data may be stored in separate files or may be managed by an older type of DBMS.

Table 4.1 Data re-engineering process

Data Problems

- End-users want data on their desktop machines rather than in a file system. They need to be able to download this data from a DBMS
- Systems may have to process much more data than was originally intended by their designers
- Redundant data may be stored in different formats in different places in the system

Data Conversion

- Data re-engineering may involve changing the data structure organisation without changing the data values.
- Data value conversion is very expensive. Special-purpose programs have to be written to carry out the conversion.

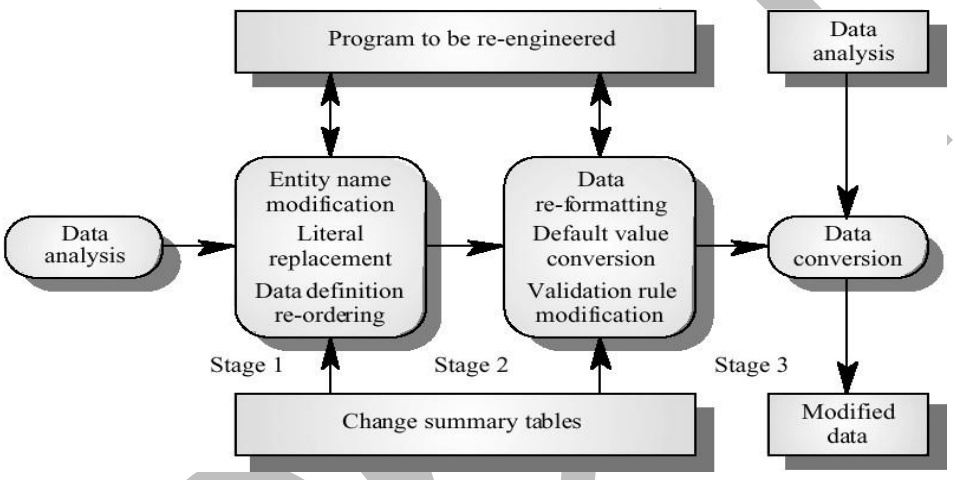


Figure 4.16 Data re-engineering process

Data inconsistency	Description
Inconsistent default values	Different programs assign different default values to the same logical data items. This causes problems for programs other than those that created the data. The problem is compounded when missing values are assigned a default value that is valid. The missing data cannot then be discovered.
Inconsistent units	The same information is represented in different units in different programs. For example, in the US or the UK, weight data may be represented in pounds in older programs but in kilograms in more recent systems. A major problem of this type has arisen in Europe with the introduction of a single European currency. Legacy systems have been written to deal with national currency units and data has to be converted to euros.
Inconsistent validation rules	Different programs apply different data validation rules. Data written by one program may be rejected by another. This is a particular problem for archival data which may not have been updated in line with changes to data validation rules.
Inconsistent representation semantics	Programs assume some meaning in the way items are represented. For example, some programs may assume that upper-case text means an address. Programs may use different conventions and may therefore reject data which is semantically valid.
Inconsistent handling of negative values	Some programs reject negative values for entities which must always be positive. Others, however, may accept these as negative values or fail to recognise them as negative and convert them to a positive value.

Table 4.2 Data value inconsistencies

4.15 REVERSE AND FORWARD ENGINEERING

4.15.1 Forward Engineering:

- Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

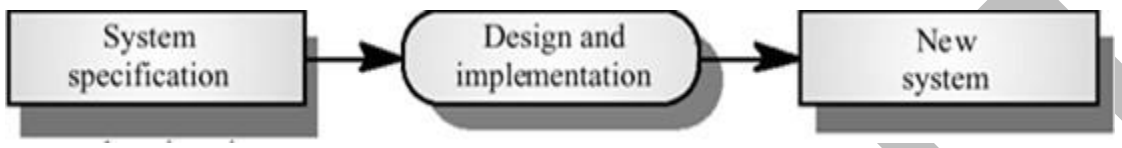


Figure 4.16 Forward engineering

- Forward engineering is same as software engineering process with only one difference it is carried out always after reverse engineering.
- The forward engineering process applies software engineering principles, concepts, and methods to re- create an existing application. In most cases, forward engineering does not simply create a modern equivalent of an older program.
- Rather, new user and technology requirements are integrated into the reengineering effort.
- The redeveloped program extends the capabilities of the older application.

4.15.2 Reverse Engineering

- Analysing software with a view to understanding its design and specification
- May be part of a re-engineering process but may also be used to re-specify a system for re-implementation.
- Builds a program data base and generates information from this.
- Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process.

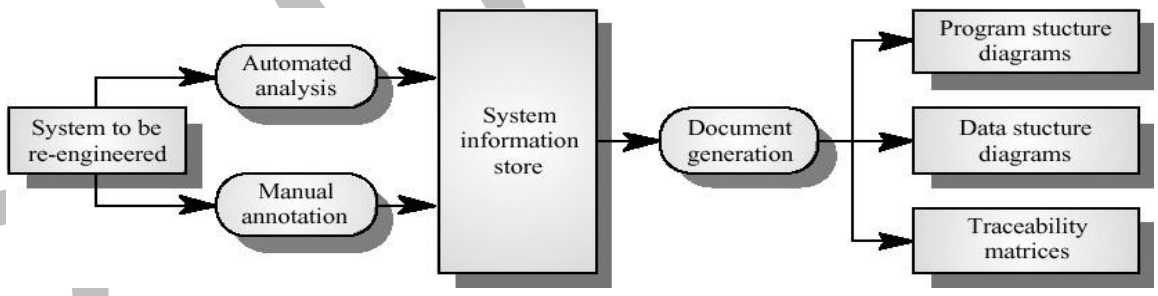


Figure 4.17 Reverse Engineering Process

- Reverse engineering often precedes re-engineering but is sometimes worthwhile in its own right
  - The design and specification of a system may be reverse engineered so that they can be an input to the requirements specification process for the system's replacement
  - The design and specification may be reverse engineered to support program maintenance