## TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process.

A number of software-testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

-To perform effective testing, a software team should conduct effective formal technical reviews. By doing this, many errors will be eliminated before testing commences.

-Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.

-Different testing techniques are appropriate at different points in time.

-Testing is conducted by the developers of the software and (for large projects) an independent test group.

-Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source-code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy must provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems must surface as early as possible. Software testing has been defined as:

**The process of analyzing a software item to detect the differences between existing and required conditions (i.e., bugs) and to evaluate the features of the software items. Or it is process of analyzing a program with the intent of finding errors.**

## 2. Abstraction.

An abstraction of a component describes the external behaviour of that component without bothering with the internal details that produce the behaviour.

Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase.

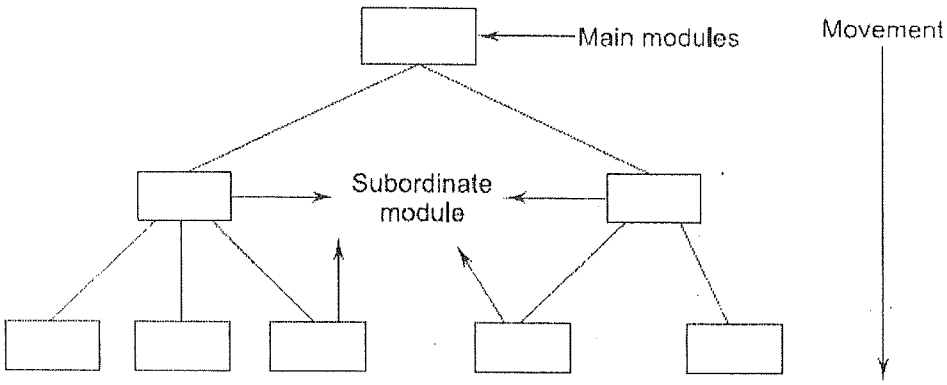
There are two common abstraction mechanisms for software systems: Functional abstraction and data abstraction.

## 3. Top-down and Bottom-up Design.

A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level components correspond to the total system.

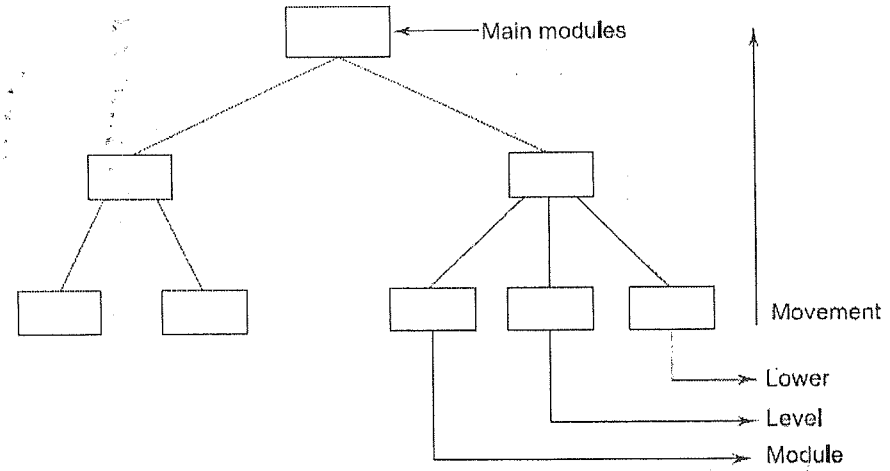
To design such hierarchies there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved.




(Movement in top-down approach)

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-levelcomponents. Bottom-up methods work with layers of abstraction.




(Movement in bottom-up approach)

## ARCHITECTURAL DESIGN

Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystem control and communication is called architectural design.

Architectural design represents the structure of data and program components that are required to build a computer-based system.

Architectural design methods have a look into various architectural styles for designing a system. These are:

-Data-centric architecture
-Data-flow architecture
-Object-oriented architecture
-Layered architecture

## Objectives of Architectural Design

The objective of architectural design is to develop a model of software architecture, which gives an overall organization of the program module in the software product. Software architecture encompasses two aspects of structures of the data and hierarchical structures of the software components.

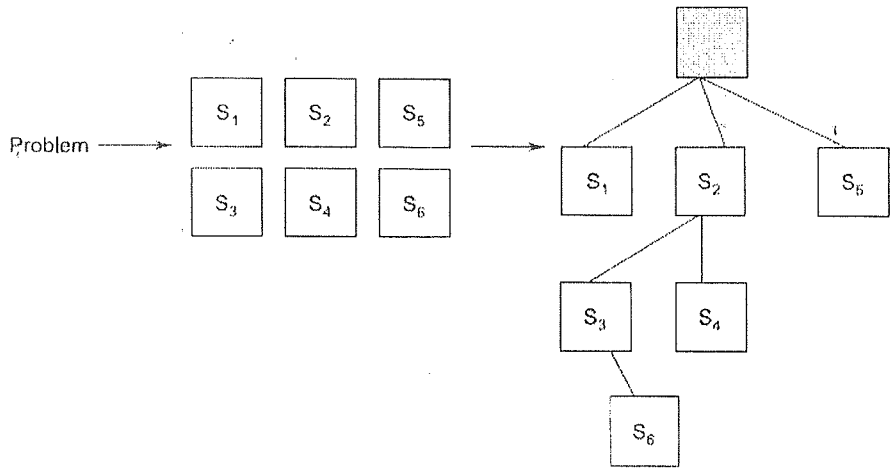



FIGURE 5.5 Problems, Solutions, and Architecture

The objective of architectural design is also to control the relationship between modules. One module may control another module or may be controlled by another module.

مبادئ الاختبار
## TESTING PRINCIPLES

There are many principles that guide software testing. Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. The following are the main principles for testing:

يتتبع خلل
**1. All tests should be traceable to customer requirements.** This is in order to uncover any defects that might cause the program or system to fail to meet the client's requirements.

**2. Tests should be planned long before testing begins.** Soon after the requirements model is completed, test planning can begin. Detailed test cases can begin as soon as the design model is designed.

**3. The Pareto principle applies to software testing.** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

**4. Testing should begin "in the small" and progress toward testing "in the large."** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

شامل
**5. Exhaustive testing is not possible.** The number of path permutations for even a moderately-sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

**6. To be most effective, testing should be conducted by an independent third party.** The software engineer who has created the system is not the best person to conduct all tests for the software.

اهداف الاختبار
## TESTING OBJECTIVES

The testing objective is to test the code, whereby there is a high probability of discovering all errors.

This objective also demonstrates that the software functions are working according to software requirements specification (SRS) with regard to functionality, features, facilities, and performance. It should be noted, however, that testing will detect errors in the written code, but it will not show an

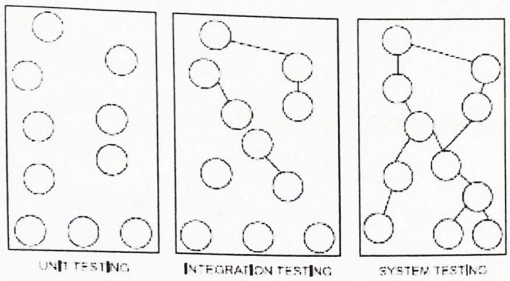
error if the code does not address a specific requirement stipulated in the SRS but not coded in the program

Testing objectives are:

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

## LEVELS OF TESTING

There are three levels of testing, i.e., three individual modules in the entire software system.




### 1. Unit Testing

In unit testing individual components are tested to ensure that they operate correctly. It focuses on verification effort. On the smallest unit of software design, each component is tested independently without other system components.

There are a number of reasons to do unit testing rather than testing the entire product:

- The size of a single module is small enough that we can locate an error fairly easily.
- The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
- Confusing interactions of multiple errors in widely different parts of the software are eliminated.

Common errors in computation are:

- Incorrect arithmetic precedence
- Mixed code operations
- Incorrect initialization
- Precision inaccuracy
- Incorrect symbolic representation of an expression

## 2. Integration Testing

The second level of testing is called integration testing. Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.

In this testing many unit-tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly.

### Objective of Integration Testing

The primary objective of integration testing is to test the module interfaces in order to ensure that there are no errors in the parameter passing, when one module invokes another module.

### Approaches to Integration Testing

The various approaches used for integration testing are:

1. Incremental Approach
3. Bottom-up Integration
5. Smoke Testing
2. Top-down Integration4. Regression Testing
6. Sandwich Integration

## 3. System Testing

Subsystems are integrated to make up the entire system. The testing process is concerned with finding errors that result from unanticipated interactions between subsystems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.

There are essentially three main kinds of system testing:

- Alpha testing
- Beta testing
- Acceptance testing

1. Alpha Testing. Alpha testing refers to the system testing carried out by the test team within the development organization.
The alpha test is conducted at the developer's site by the customer under the project team's guidance. In this test, users test the software on the development platform and point out errors for correction.

2. Beta Testing. Beta testing is the system testing performed by a selected group of friendly customers.
If the system is complex, the software is not taken for implementation directly. It is installed and all users are asked to use the software in testing mode; this is not live usage. This is called the beta test.

3. Acceptance Testing. Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.

When customer software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than the software engineers, an acceptance test can range from an informal 'test drive' to a planned and systematically executed series of tests.

## WHITE-BOX TESTING/STRUCTURAL TESTING

A complementary approach to functional or black-box testing is called structural or white-box testing. In this approach, test groups must have complete knowledge of the internal structure of the software.

The tester can analyze the code and use knowledge about the structure of a component to derive test data. The analysis of the code can be used to find out how many test cases are needed to guarantee that all of the statements in the program are executed at least once during the testing process.

### Reasons White-box Testing is performed

White-box testing is carried out to test whether:

- All paths in a process are correctly operational.
  - All logical decisions are executed with true and false conditions.
  - All loops are executed with their limit values tested.
  - To ascertain whether input data structure specifications are tested and then used for other processing.

### Advantages of Structural/White-box Testing

The various advantages of white-box testing include:

- Forces test developer to reason carefully about implementation.
- Approximates the partitioning done by execution equivalence.
- Reveals errors in hidden code.

## FUNCTIONAL/BLACK-BOX TESTING

Functional testing refers to testing that involves only observation of the output for certain input values, and there is no attempt to analyze the code, which produces the output.

The internal structure of the program is ignored. For this reason, functional testing is sometimes referred to as black-box testing (also called behavioural testing) in which the content of a black-box is not known and the function of black box is understood completely in terms of its inputs and outputs.

In black-box testing, the tester only knows the inputs that can be given to the system and what output the system should give. In other words, the basis for deciding test cases in functional testing is the requirements or specifications of the system or module. This form of testing is also called functional or behavioural testing.

Black-box testing identifies the following kinds of errors:

- Incorrect or missing functions.
- Interface missing or erroneous.
- Errors in data model.
- Errors in access to external data source.

### Advantages of Black-box Testing

The advantages of this type of testing include:

- The test is unbiased because the designer and the tester are independent of each other.
- The tester does not need knowledge of any specific programming languages.
- The test is done from the point-of-view of the user, not the designer.
- Test cases can be designed as soon as the specifications are complete.

# 1. MAİNTENANCE

Software maintenance is the activity associated with keeping an operational computer system continuously in tune with the requirements of users and data-processing operations. The software maintenance process is expensive and risky and is very challenging. There is a need for software maintenance due to the following reasons:

-Changes in user requirements with time
-Program/System problems
-Changing hardware/Software environment
-To improve system efficiency and throughout
-To modify the components
-To test the resulting product to verify the correctness of changes
-To eliminate any unwanted side effects resulting from modifications
-To augment or fine-tune the software
-To optimize the code to run faster
-To review standards and efficiency
-To make the code easier to understand and work with
-To eliminate any deviations from specifications

## 1.1 Maintenance To-Do List

-Correct errors
-Correct requirements and design flaws
-Improve the design
-Make enhancements
-Interface with other systems
-Convert for use with other hardware
-Migrate legacy systems
-Retire systems
-Major aspects
-Maintain control over the system's day-to-day functions
-Maintain control over system modification
-Perfect existing acceptable functions

-Prevent system performance from degrading to unacceptable levels

## 2. CATEGORIES OF MAINTENANCE

Maintenance may be classified into the four categories as follows:

-**Corrective** - reactive modifications to correct discovered problems.
-**Adaptive** - modifications to keep it usable in a changed or changing environment.
-**Perfective** - improve performance or maintainability.
-**Preventive** - modifications to detect and correct latent faults.

**(i) Corrective Maintenance.** Corrective maintenance means repairing processing or performance failures or making changes because of previously uncorrected problems.

**(ii) Adaptive Maintenance.** Adaptive maintenance means changing the program functions. This is done to adapt to external environment changes. For example, the current system was designed so that it calculates taxes on profits after deducting the dividend on equity shares. The government has issued orders now to include the dividend in the company profit for tax calculation. This function needs to be changed to adapt to the new system.

## 3. MAINTENANCE COSTS

In the 1970s, most of a software system's budget was spent on development. The ratio of development money to maintenance money was reversed in the 1980s, and various estimates place maintenance at 40 to 60% of the full life-cycle cost of a system (i.e., from development through maintenance to eventual retirement or replacement). However, this cost may vary widely from one application domain to another.

It is advisable to invest more effort in early phases of the software life-cycle to reduce maintenance costs. The defect repair ratio increases heavily from the analysis phase to the implementation phase as shown in Table 9.1.

Page 204

4.1 Factors Affecting Effort

There are many other factors that contribute to the effort needed to maintain a system. These factors include the following:

-Application type
-System novelty
-Turnover and maintenance staff availability
-System life-span
-Dependence on a changing environment
-Hardware characteristics
-Design quality
-Code quality
-Documentation quality
-Testing quality

(*iii*) **Perfective Maintenance.** Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. For example, earlier data was sent from stores to headquarters on magnetic media but after stores were electronically linked via leased lines, the software was enhanced to send data via leased lines.

(*iv*) **Preventive Maintenance.** Preventive maintenance is the process by which we prevent our system from being obsolete. Preventive maintenance involves the concept of re-engineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

## CASE

CASE stands for Computer-Aided Software Engineering. *CASE is a tool that helps a software engineer maintain and develop software.*

The workshop for software engineering is called an Integrated Project Support Environment (IPSE) and the tool-set that fills the workshop is called CASE.

CASE is an automated support tool for software engineers in any software engineering process.

Software engineering mainly includes the following processes:

- Translation of user needs into software requirements
- Transaction of software requirements into design specifications
- Implementation of design into code
- Testing of the code
- Documentation

CASE technology provides software-process support by automating some process activities and by providing information about the software being developed. Examples of activities, which can be automated using CASE, include:

- The development of graphical system models as part of the requirements specification or the software design.
- Understanding a design using a data dictionary, which holds information about the entities and relations in a design.
- Program debugging through the provision of information about an executing program.
- The automated translation of programs from an old version of a programming language, such as COBOL, to a more recent version.

## OBJECTIVES OF CASE

1. **Improve Productivity.** Most organizations use CASE to increase the speeds with which systems are designed and developed. Imagine the difficulties carpenters would face without hammers and saws. Tools increase the analysts' productivity by reducing the time needed to document, analyze, and construct an information system.

2. **Improve Information System Quality.** When tools improve processes, they usually improve the results as well. They:

- Ease and improve the testing process through the use of automated checking.
- Improve the integration of development activities via common methodologies.
- Improve the quality and completeness of documentation. Help standardize the development process.
- Improve the management of the project.
- Simplify program maintenance.
- Improve software portability across environments.
- Through reverse engineering and re-engineering, CASE products extend the files of existing systems.

3. **Improve Effectiveness.** Effectiveness means doing the right task (i.e., deciding the best task to perform to achieve the desired result). Tools can suggest procedures (the right way) to approach a task.

## CASE REPOSITORY

A CASE repository is a system-developer database. Synonyms include dictionary and encyclopedia. It is a place where developers can store system models, detailed descriptions and specifications, and other products of system development.

Analysts use CASE repositories for five important reasons:

- To manage the details in large systems.
- To communicate a common meaning for all system elements.
- To document the features of the system.
- To facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made.
- To locate errors and omissions in the system.

## CHARACTERISTICS OF CASE TOOLS

All CASE tools have the following characteristics:

- A graphic interface to draw diagrams, charts, models (uppercase, middle case, lowercase).
- An information repository, a data dictionary for efficient information management selection, usage, application, and storage.
- Common user interface for integration of multiple tools used in various phases.
- Automatic code generators.
- Automatic testing tools.

## LEVELS OF CASE

There are three different levels of CASE technology:

1. **Production Process Support Technology.** This includes support for process activities, such as specification, design, implementation, testing, and so on.

2. **Process Management and Technology.** This includes tools to support process modeling and process management. These tools are used for specific support activities.

3. **Meta-CASE Technology.** Meta-CASE tools are generators, which are used to create production process-management support tools.

## ARCHITECTURE OF CASE ENVIRONMENT

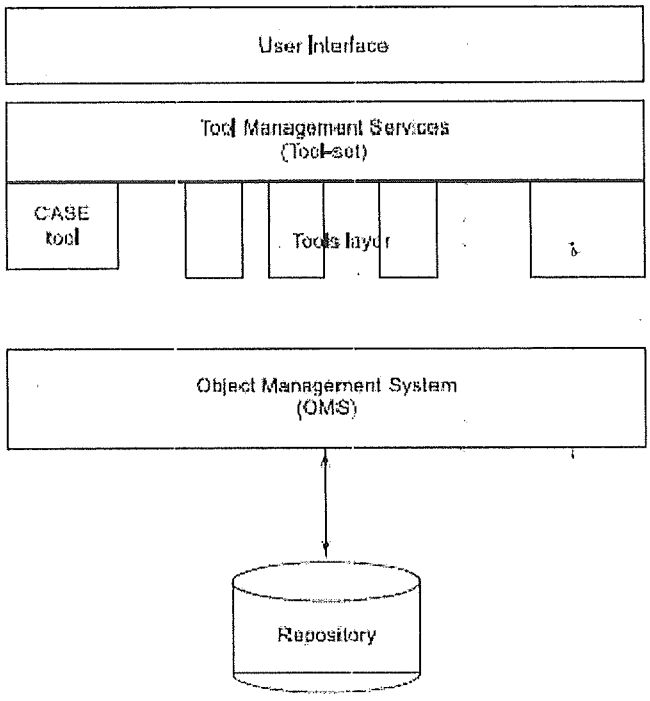



FIGURE 10.1 Architecture of CASE Environment

List of CASE Tools

| S.No. | Application | CASE Tool | Purpose of Tool |
|---|---|---|---|
| 1. | Planning | Excel spreadsheet, MS Project, PERT/CPM Network, Estimation tools | Functional Application: Planning, scheduling, control |
| 2. | Editing | Diagram editors, Text editors, Word processors | Speed and Efficiency |
| 3. | Testing | Test-data generators, File comparators | Speed and Efficiency |
| 4. | Prototyping | High-level modeling language, User-interface generators | Confirmation and certification of RDD and SRS |
| 5. | Documentation | Report generators, Publishing imaging, PPT presentation | Faster structural documentation with quality of presentation |
| 6. | Programming and Language-processing integration | Program generators, Code generators, Compilers, Interpreters interface, connectivity | Programming of high quality with no errors, system integration |

## CATEGORIES OF CASE TOOLS

CASE tools are divided into the following two categories:

1. Vertical CASE tools
2. Horizontal CASE tools

1. **Vertical CASE Tools.** Vertical CASE tools provide support for certain activities within a single phase of the software life-cycle.

2. **Horizontal CASE Tools.** These tools support the automated transfer of information between the phases of a life-cycle. These tools include project management, configuration-management tools, and integration services.

## ADVANTAGES OF CASE TOOLS

The major benefits of using CASE tools include the following:

- Improved productivity
- Better documentation
- Improved accuracy
- Improved quality
- Reduced lifetime maintenance
- Reduced cost of software
- Produce high-quality and consistent documents
- Reduce the drudgery in a software engineer's work
- Increase speed of processing
- Easy to program software
- Improved coordination among staff members who are working on a large software project
- An increase in project control through better planning, monitoring, and Communication

## DISADVANTAGES OF CASE TOOLS

1. **Purchasing of CASE Tools is Not an Easy Task.** The cost of CASE tools is very high. For this reason small software development firms do not invest in CASE tools.

2. **Learning Curve.** In general, programmer productivity may fall in the initial phase of implementation as users need time to learn this technology.

3. **Tool Mix.** It is important to make proper selection of CASE tools to get maximum benefits from the tools, as the wrong selection may lead to the wrong results.

## TEST CASE

A test case is a set of instructions designed to discover a particular type of error or defect in the software system by inducing a failure.

The goal of selected test cases is to ensure that there is no error in the program and if there is it then should be immediately depicted. Ideal test casement should contain all inputs to the program.

There are two criteria for the selection of test cases:

- Specifying a criterion for evaluating a set of test cases.
- Generating a set of test cases that satisfy a given criterion.

Each test case needs proper documentation, preferably in a fixed format. There are many formats; one format is suggested below:

| | |
|---|---|
| Test case name | Test Case ID |
| Purpose of test | Testing object (unit, application, module, etc.) |
| Test attribute | |
| Tests focus (function, feature, process, interface, validation, verification, etc.) | |
| Test type (alpha, beta, unit, integration, system) | |
| Test process | A set of instructions for conducting the test-initial stating condition-inputs-specifications-output expected |
| Test results | Expected and actual and comparison, error description, post-process state |
| Action | Correction, authorization, and feedback through retest |
| Action to initialize the pre-test status | |

# SOFTWARE-TESTINGSTRATEGIES

## STATIC-TESTING STRATEGIES

Static testing is the systematic examination of a program structure for the purpose of showing that certain properties are true regardless of the execution path the program may take.

Static testing strategies include:

- Formal technical reviews
- Walkthroughs
- Code inspections.

### 1. Formal Technical Reviews

A software review can be defined as a filter for the software-engineering process. The purpose of any review is to discover errors in the analysis, design, and coding, testing and implementation phases of the software-development cycle.

### Objectives for Reviews

Review objectives are used:

- To ensure that the software elements conform to their specifications.
- To ensure that the development of the software element is being done as per plans, standards, and guidelines applicable for the project.
- To ensure that the changes to the software elements are properly implemented and affect only those system areas identified by the change specification.

**Types of Reviews**

- Informal Technical Review: An informal meeting and informal desk checking.
- Formal Technical Review: A formal software quality assurance activity through various approaches, such as structured walkthroughs, inspections, etc.

**2. Code Walkthroughs**

A code walk-through is an informal analysis of code as a cooperative, organized activity by several participants.

**3. Code Inspections**

A code inspection, originally introduced by Fagan (1976) at IBM, is similar to a walk-through but is more formal.
In code inspection, the analysis is aimed explicitly at the discovery of commonly made errors. In such a case, it is useful to state beforehand the type of errors for which we are searching.

**DEBUGGING**

Debugging means identifying, locating, and correcting the bugs usually by running the program. It is an extensively used term in programming. These bugs are usually logical errors.

During the compilation phase the source files are accessed and if errors are found, then that file is edited and the corrections are posted in the file. After the errors have been detected and the corrections have been included in the source file, the file is recompiled. This detection of errors and removal of those errors is called debugging.

## Debugging Tactics/Categories

The various categories for debugging are:

- Brute-force debugging
- Backtracking
- Cause elimination
- Program slicing
- Fault-tree analysis

## Program Debugging

People think that program testing and debugging are the same thing. Though closely related, they are two distinct processes. Testing establishes the presence of errors in the program. Debugging is the locating of those errors and correcting them. Debugging depends on the output of testing which tells the programmer about the presence or absence of errors.

There are various debugging stages, as shown in Figure 8.2. The incorrect parts of the code are located and the program is modified to meet its requirements. After repairing, the program is tested again to ensure that the errors have been corrected. Debugging can be viewed as a problem-solving process.

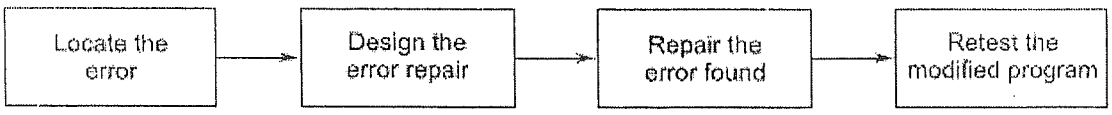



FIGURE 8.2 Debugging Stages

## ERROR, FAULT, AND FAILURE

### Errors

Error refers to the difference between the actual output of the software and the correct output.

### Types of Errors

Errors can be classified into two categories:

**1. Syntax Error.** A syntax error is a program statement that violates one or more rules of the language in which it is written.

**2. Logic Error.** A logic error deals with incorrect data fields, out-of-range terms, and invalid combinations.

### Faults

A fault is a condition that causes a system to fail in performing its required function.

A fault is the basic reason for software malfunction. It is also commonly called a bug. Even though correct input is given to the system, when it fails then we say the system has a fault or a bug, and needs repair.

### Failure

Failure is the inability of the software to perform a required function to its specification.

In other words, when software goes ahead in processing without showing error or fault even though certain input and process specification are violated, then it is called a software failure.

## Software Quality

Verification and validation (V and V) is the name given to the checking and analysis process that ensures that software conforms to its specifications and meets the needs of the customers who are paying for that software.

Variation and validation is a whole life-cycle process. It starts with requirements reviews and continues through design reviews and code inspection to product testing. There should be V and V activities at each stage of the software development process. These activities ensure that the results of process activities are as specified.

Verification and validation is not the same thing although they are easily confused. The difference between them is succinctly expressed by Boehm (1979):

- '**Validation:** Are we building the right product?'
- '**Verification:** Are we building the product right?'

### Verification

*Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase.*

### Validation

*Validation is the process of determining whether a fully developed system confirms to its requirement specifications.*

Thus, the goal of the verification and validation process is to establish confidence in the customer that the software system is 'fit for the customer.' It doesn't mean that the software system is free from errors.

## SOFTWARE QUALITY ASSURANCE

The aim of the Software Quality Assurance (SQA) process is to develop a high quality software product. *Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed and/or maintained.*

The purpose of a software quality assurance group is to provide assurance that the procedures, tools, and techniques used during product development and modification are adequate to provide the desired level of confidence in the work products.

The process of the SQA:

1. Defines the requirements for software controlled system fault/failure detection, isolation, and recovery;
2. Reviews the software-development processes and products for software error prevention and/or controlled change to reduced functionality states;
3. Defines the process for measuring and analyzing defects as well as reliability and maintainability factors.

### Software Quality Attributes

Software quality is comprised of six main attributes (called characteristics):

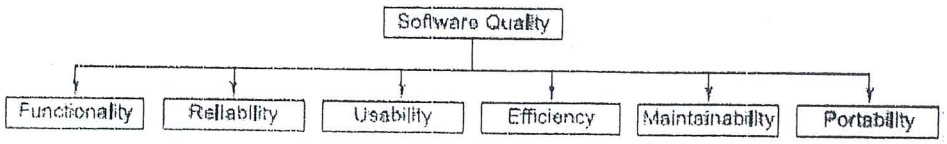



FIGURE 4.1 Software Quality Attributes

1. *Functionality:* The capability to provide functions which meet stated and implied needs when the software is used.

2. ***Reliability:*** The capability to maintain a specified level of performance.

3. *Usability:* The capability to be understood, learned, and used.

4. *Efficiency:* The capability to provide appropriate performance relative to the amount of resources used.

5. ***Maintainability:*** The capability to be modified for purposes of making corrections, improvements, or adaptation.

6. ***Portability:*** The capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product.

**Importance of Software Quality**

1. ***Increasing Criticality of Software.***
2. ***The Intangibility of Software.***
3. ***Accumulating Errors During Software Development.***

## INTERNATIONAL STANDARD ORGANIZATION (ISO)

The International Organization for Standardization (ISO) is a group of worldwide federations of national standards bodies from some 100 countries. The ISO is a non-governmental organization established in 1947.

The ISO-9000 standard specifies quality-assurance elements in generic terms; The ISO-9001 is a quality-assurance standard that is specific to software engineering. It specifies 20 standards with which an organization must comply for an effective implementation of the quality assurance system.

# SOFTWARE-PROJECT ESTIMATION

Software-project estimation is the process of estimating various resources required for the completion of a project. Effective software-project estimation is an important activity in any software-development project. Underestimating software projects and understaffing it often leads to low-quality deliverables, and the project misses the target deadline leading to customer dissatisfaction and loss of credibility to the company. On the other hand, overstaffing a project without proper control will increase the cost of the project and reduce the competitiveness of the company.

Software-project estimation mainly encompasses the following steps:

1. **Estimating the Size of the Project.** There are many procedures available for estimating the size of a project, which are based on quantitative approaches, such as estimating lines of code or estimating the functionality requirements of the project called function points.

2. **Estimating Efforts Based on Person-months or Person-hours.** Person-month is an estimate of the personal resources required for the project.

3. **Estimating Schedule** in Calendar Days/Month/Year Based on Total Person-months Required and Manpower Allocated to the Project. The duration in calendar month = Total person-months/Total manpower allocated.

**4. Estimating Total Cost of the Project Depending on the Above and Other Resources.** In a commercial and competitive environment, software-project estimation is crucial for managerial decision-making.

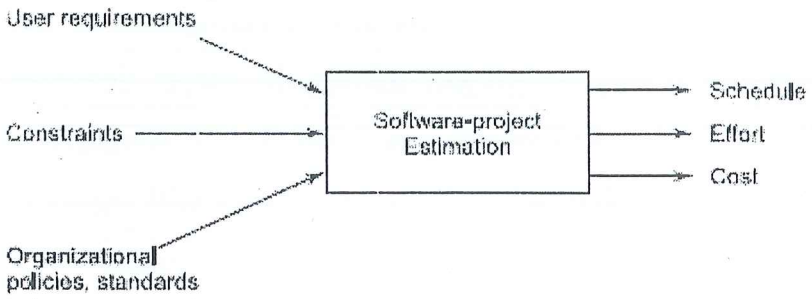



FIGURE 9.7 Software-project Estimation

## 1. Estimating Size

Estimating the size of the software to be developed is the very first step to make an effective estimation of the project. Customer requirements and system specifications form a baseline for estimating the size of software. At a later stage of the project, system design documents can provide additional details for estimating the overall size of the software.

1. The ways to estimate project size can be through past data from an earlier developed system. This is called estimation by analogy.
2. The other way of estimation is through product feature/functionality. The system is divided into several subsystems depending on functionality, and the size of each subsystem is calculated.

## 2. Estimating Effort

Once the size of software is estimated, the next step is to estimate the effort based on the size. The estimation of effort can be made from the organizational specifics of the software-development life-cycle. The development of any application software system is more than just the coding of the system. Depending on deliverable requirements, the estimation of effort for a project will vary.

Efforts are estimated in the number of person-months.

1. The best way to estimate effort is based on the organization's own historical data of development processes. Organizations follow a similar development life-cycle when developing various applications.
2. If the project is of a different nature, which requires the organization to adopt a different strategy for development, then different models based on algorithmic approaches can be devised to estimate the required effort.

## 3. Estimating Schedule

The next step in the estimation process is estimating the project schedule from the effort estimated. The schedule for a project will generally depend on human resources involved in a process. Efforts in person-months are translated to calendar months.

Schedule estimation in calendar months can be calculated using the following model [McConnell]:

Schedule in calendar months = $3.0*(\text{person-months})^{1/3}$

The parameter 3.0 is variable, used depending on the situation that works best for the organization.

## 4. Estimating Cost

Cost estimation is the next step for projects. The cost of a project is derived not only from the estimates of effort and size but from other parameters, such as hardware, travel expenses, telecommunication costs, training costs, etc. Figure 9.8 depicts the cost-estimation process.

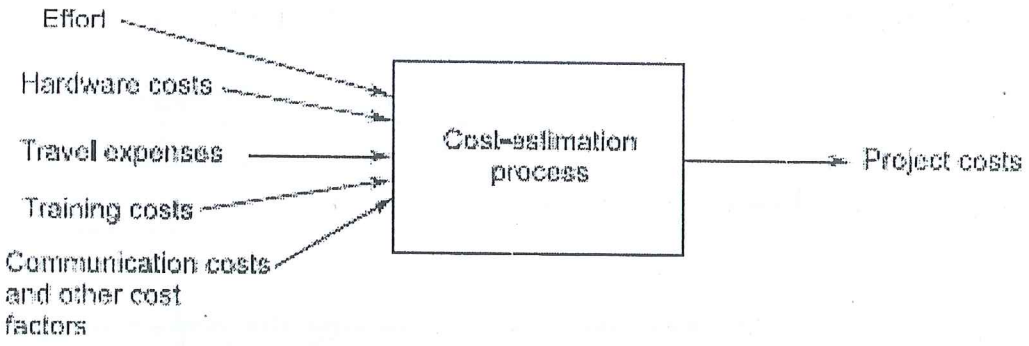



FIGURE 9.8 Cost-estimation Process

## CONSTRUCTIVE COST MODEL (COCOMO)

COCOMO stands for Constructive Cost Model. It was introduced by Barry Boehm in 1981. It is perhaps the best known and most thoroughly documented of all software-cost estimation models. It provides the following three levels of models:

1. **Basic COCOMO:** A single-value model that computes software-development costs as a function of an estimate of LOC.
2. **Intermediate COCOMO:** This model computes development costs and effort as a function of program size (LOC) and a set of cost drivers.
3. **Complete COCOMO:** This model computes development effort and costs which incorporates all characteristics of intermediate levels with assessment of cost implications in each step of development (analysis, design, testing, etc.).

This model may be applied to three classes of software projects as given below:

1. **Organic:** Small-size project. A simple software project where the development team has good knowledge of the application.
2. **Semi-detached:** An intermediate-size project and the project is based on rigid and semi-rigid requirements.
3. **Embedded:** The project is developed under hardware, software, and operational constraints. Examples are embedded software and flight control software.

## 1. Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \quad \text{PM}$$
$$T_{dev} = b_1 \times (\text{Effort})^{b_2} \quad \text{Months}$$

Where KLOC is the estimated size of the software product expressed in Kilo Lines and Code $a1$, $a2$, $b1$, $b2$ are constants of the software product.

Tdev is the estimated time to develop the software, expressed in months. Effort is the total effort required to develop the software product, expressed in person-months (PM).

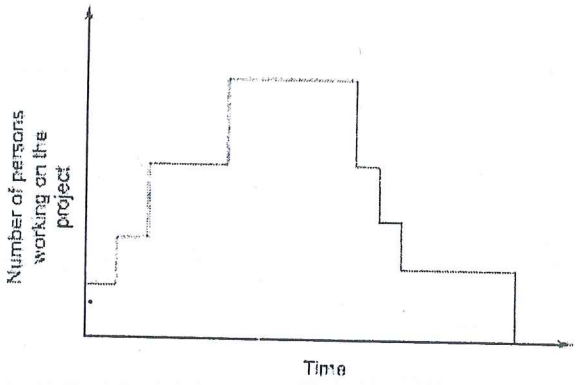



FIGURE 9.10 Person-month Curve

Estimation of Development Effort
Organic: Effort = 2.4 $(\text{KLOC})^{1.05}$ PM
Semi-detached: Effort = 3.0 $(\text{KLOC})^{1.12}$ PM
Embedded: Effort = 3.6 $(\text{KLOC})^{1.20}$ PM

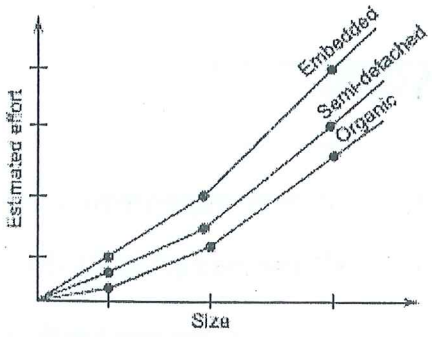



FIGURE 9.11 (a) Effort Versus Size

Figure 9.11 (*a*) shows a plot of the estimated effort versus the size for various product sizes.

Estimation of Development Time
Organic: Tdev = 2.5 (Effort)$^{0.38}$ Months
Semi-detached: Tdev = 2.5 (Effort)$^{0.35}$ Months
Embedded: Tdev = 2.5(Effort)$^{0.32}$ Months

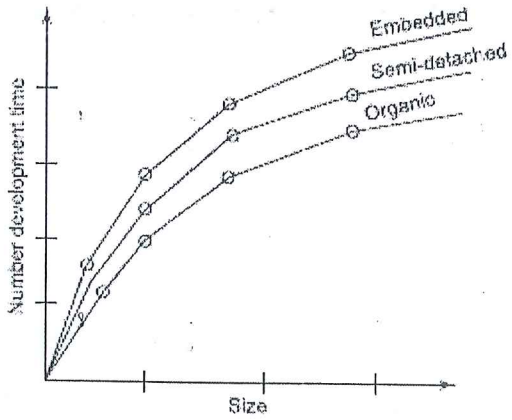



Figure 9.11 (*b*) shows a plot of development time versus product size.

## 2. Intermediate COCOMO Model

The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained through the basic COCOMO expression by using a set of 15 cost drivers (multipliers) based on various attributes of software development.

## 3. Complete COCOMO Model

A large amount of work has been done by Boehm to capture all significant aspects of software development. It offers a means for processing all the project characteristics to construct a software estimate.

## SOFTWARE RE-ENGINEERING

*Re-engineering means to re-implement systems to make them more maintainable.*

In re-engineering, the functionality and system architecture remains the same but it includes re-documenting, organizing and restricting, modifying and updating the system. It is a solution to the problems of system evolution. In other words,

*Re-engineering essentially means having a re-look at an entity, such as a process, task, designs, approach, or strategy using engineering principles to bring in radical and dramatic improvements.*

The re-engineering approach attacks five parameters, namely: management philosophy, pride, policy, procedures, and practices to bring in radical improvements impacting cost, quality, service, and speed.

When re-engineering principles are applied to business process then it is called Business Process Reengineering (BRP).

### Principles of Software Re-engineering

The principles of re-engineering when applied to software-development processes are called software re-engineering. It affects positively software cost, quality, service to the customer, and speed of delivery. Software, whether a product or system, deals with business processes making them faster, smarter, and automatic in response to delivery and execution. In software re-engineering,

We may resort to one or more of the following:

- Redefining software scope and goals.
- Redefining SRS by way of additions, deletions, and extensions of functions and features.

- Redesigning the application design and architecture using new technology, upgrades, and platforms, interfacing to new technologies to make the process faster, smarter, and automatic.
- Resorting to data restructuring, improving database design, code restructuring to make the size smaller and more efficient in operations.
- Rewriting the documentation to make it more users friendly.

## Re-engineering Process

Figure 10.8 illustrates a possible re-engineering process. The input to the process is a legacy program and the output is a structured, modularized version of the same program. At the same time as program re-engineering, the data for the system may also be re-engineered.

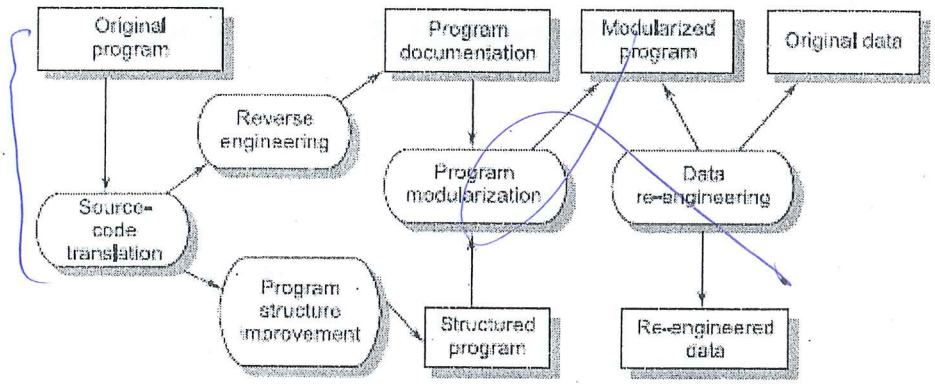



FIGURE 10.8 Re-engineering Process

The re-engineering process includes the following activities:

- ***Source-code translation:*** In source-code translation the programming language of an old program is converted into the modern version of the same language or to a new language.

- ***Reverse engineering:*** In reverse engineering the program is analyzed and important and useful information is extracted from it which helps to document its functionality.

- ***Program structure improvement:*** In program structure improvement the control structure of the program is analyzed and modified to make it easier to read and understand.

- ***Program modularization:*** In program modularization redundancy of any part is removed and related parts are grouped together.

- ***Data re-engineering:*** In data re-engineering the data processed by the program is changed to reflect program changes.

**Software Re-engineering Process Model**

Re-engineering is a rebuilding activity. To implement re-engineering principles we apply a software re-engineering process model. The re-engineering paradigm shown in Figure 10.9 is a cyclical model.

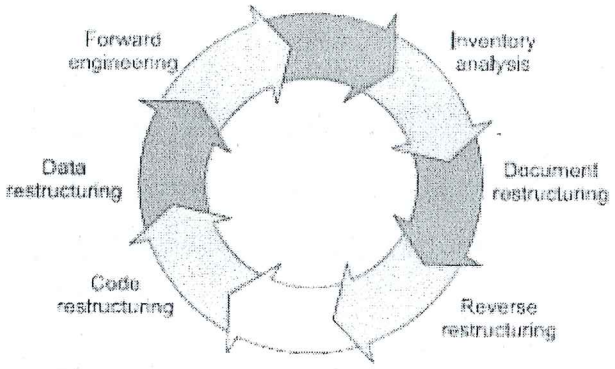



FIGURE 10.9 Software Re-engineering Process Model

There are six activities in the model:

1. Inventory analysis
2. Document restructuring
3. Reverse engineering
4. Code restructuring
5. Data restructuring
6. Forward engineering

## Factors Affecting Re-engineering Costs

Factors that affect re-engineering costs include:

1. *Quality of software to be re-engineered:* There is the inverse relationship between the quality and the cost of the software.
2. *Tools available for re-engineering:* It is not cost effective to re-engineer a software system unless you can use CASE tools to automate most of the program changes.
3. *Availability of expert staff:* The re-engineering staff is not the same as the maintaining staff, and this will increase costs.
4. *Extent of data conversion required:* There is a direct relationship between the volume of data to be converted and the cost of the software.

## Differences between Forward Engineering and Re-engineering

Forward engineering starts with a system specification and involves the design and implementation of a new system. Re-engineering starts with an existing system and the development process for the replacement is based on the understanding and transformation of the original system.

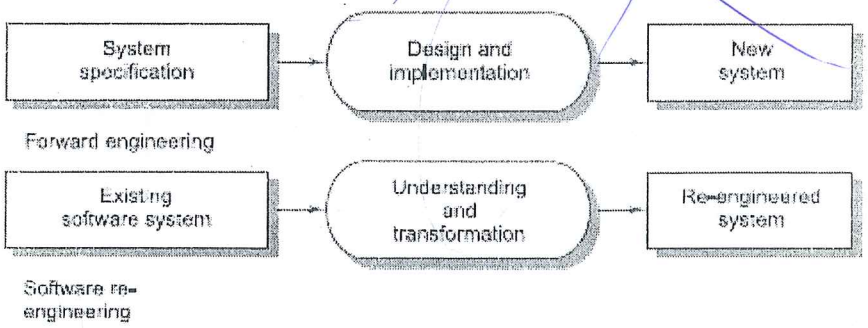



FIGURE 10.10 Forward Engineering and Re-Engineering

**Advantages and Disadvantages**

Re-engineering a software system has two key advantages over more radical approaches to system evolution:

1. ***Reduced risk:*** There is a high risk in redeveloping software that is essential for an organization. Errors may be made in the system specification; there may be development problems, etc.

2. ***Reduced costs:*** The costs of re-engineering are significantly less than the costs of developing new software. Ulrich (1990) quotes an example of a commercial system where the re-implementation costs were estimated at 550 million. The system was successfully re-engineered for $12 million. If these figures are typical, it is about four times cheaper to re-engineer than to rewrite.

The main disadvantages of software re-engineering are that there are practical limits to the extent that a system can be improved by re-engineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes of radical reorganizing of the system-data management cannot be carried out automatically, so involve high additional costs. Although re-engineering can improve maintainability, the re-engineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

# REVERSE SOFTWARE ENGINEERING

## Definition

*Reverse engineering is the process followed in order to find difficult, unknown, and hidden information about a software system.* It is becoming important, since several software products lack proper documentation, and are highly unstructured, or their structure has degraded through a series of maintenance efforts. Maintenance activities cannot be performed without a complete understanding of the software system.

## Purpose of Reverse Engineering

The main purpose of reverse engineering is to recover information from the existing code, or any other intermediate documents. Any activity that requires program understanding at any level may fall within the scope of reverse engineering.

## Reverse-Engineering Process

The reverse-engineering process is illustrated in Figure 10.5. The process starts with an analysis phase. During this phase, the system is analyzed using automated tools to discover its structure. In itself, this is not enough to re-create the system design. Engineers then work with the system source code and its structural model. They add information to this, which they have collected by understanding the system. This information is maintained as a directed graph that is linked to the program source code.

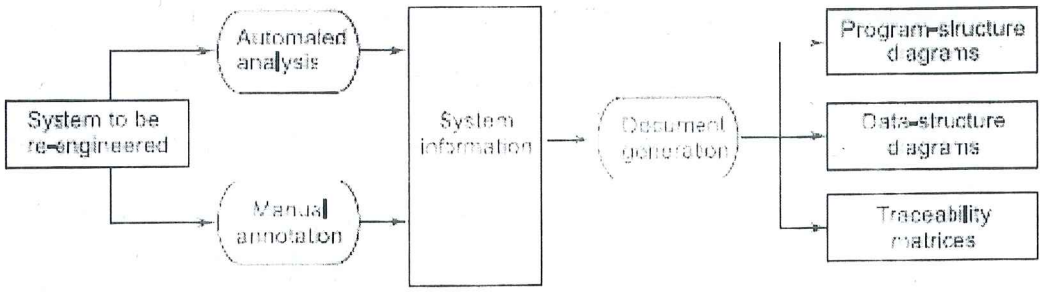



FIGURE 10.5 The Reverse-engineering Process

## Reverse-Engineering Tasks

Reverse engineering encompasses a wide array of tasks related to understanding and modifying software systems. This array of tasks can be broken into a number of classes. A few of these classes are briefly discussed below:

1. ***Mapping between application and program domains.*** The task of the reverse engineer is to reconstruct the mapping from the application domain to the program domain as shown in Figure 10.6.

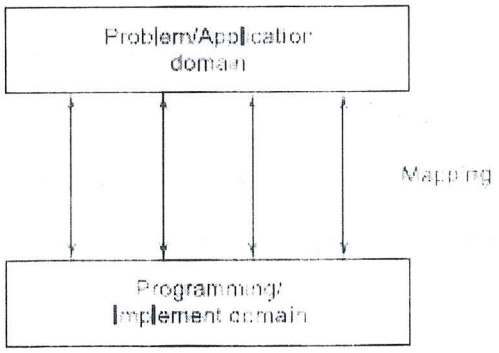



FIGURE 10.6 Mapping Between Application and Domain Programs

2. ***Mapping between concrete and abstract levels.*** The software-development process goes from high-level abstraction to more detailed design and concrete implementation. A reverse engineer has to move backward and create an abstract representation of the implementation from the mass of concrete details.

3. ***Rediscovering high-level structures.*** A program is the embodiment of a well defined purpose and coherent high-level structure.

4. ***Finding missing links between program syntax and semantics.*** In the formal world, the meaning of a syntactically correct program determines the output for a specific input. But systems that require reverse engineering generally would have lost their original semantics. The reverse-engineering process should determine the semantics of a given program from its syntax.

5. ***To extract reusable components.*** Based on the premise that the use of existing program components can lead to an increase in productivity and improvement in product quality, the concept of reuse has increasingly become popular among software engineers.

**Levels of Reverse Engineering**

Reverse engineers detect low-level implementation constructs and replace them with their high-level counterparts. The process eventually results in an incremental formation of an overall architecture of the program. It

should, nonetheless, be noted that the product of a reverse-engineering process does not necessarily have to be at a higher level of abstraction.

1. ***Re-documentation.*** Re-documentation is the recreation of a semantically equivalent representation within the same relative abstraction level. The goals of this process are threefold:

- Firstly, to create alternative views of the system as to enhance understanding; for example, the generation of hierarchical data flows or control-flow diagrams from source code.

- Secondly, to improve current documentation. Ideally, such documentation should have been produced during the development of the system and updated as the system changed. This, unfortunately, is not usually the case.

- Thirdly, to generate documentation for a newly modified program. This is aimed at facilitating future maintenance work on the system; preventive maintenance.

2. ***Design Recovery.*** Design recovery entails identifying and extracting meaningful higher-level abstractions beyond those obtained directly from examination of the source code. The recovered design, which is not necessarily the original design, can then be used for redeveloping the system.

## Characteristics of Reverse Engineering

The various characteristics of reverse software engineering are as follows:

1. ***Abstraction Level.*** The abstraction level of a reverse-engineering process and the tools used to affect it refers to the sophistication of the design information that can be extracted from the source code.

2. ***Completeness.*** The completeness of a reverse-engineering process refers to the level of detail that is provided at an abstraction level. In most cases, the completeness decreases as the abstraction level increases.

3. ***Interactivity.*** Interactivity refers to the degree to which the human is "integrated" with automated tools to create an effective reverse-engineering process.

4. ***Directionality.*** If the directionality of the reverse-engineering process is one-way, all information extracted from the source code is provided to the software engineer who can then use it during any maintenance activity. If directionality is two-way, the information is fed to a re-engineering tool that attempts to restructure or regenerate the old program.

5. ***Extract Abstractions.*** The core of reverse engineering is an activity called extract abstractions. The engineer must evaluate the old program and from the source code, develop a meaningful

specification of the processing that is performed, the user interface that is applied, and the program data structures or database that is used.

## Application Areas of Reverse Engineering

The different application areas of reverse software engineering include:

- Program comprehension
- Re-documentation
- Recovery of design approach and design details at any level of abstraction
- Identifying re-usable components
- Identifying components that need restructuring
- Recovery business rules

# Risk Analysis and Management

- A series of steps that help a software team to understand and manage uncertainty.
- A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it's a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.
- Everyone involved in the software process—managers, software engineers, and customers—participate in risk analysis and management.

**What are the steps?**

- Recognizing what can go wrong is the first step, called "risk identification."
- Risk is analyzed to determine the likelihood that it will occur and the damage that it will do if it does occur.
- Once this information is established, risks are ranked, by probability and impact.
- A contingency plan is developed to manage those risks with high probability and high impact.

**Reactive vs. Proactive Risk Strategies**

- Reactive strategies-very common, also known as firefighting mode, project team sets resources aside to deal with problems and does nothing until a risk becomes a problem.
- Proactive strategies-risk management begins long before technical work starts, risks are identified and prioritized/ranked by importance, then team builds a plan to avoid risks if they can or minimize them if the risks turn into problems(Not all risks can be avoided).

**Software Risks**

- There is general agreement that risk always involves two characteristics:
  - Uncertainty--the risk may or may not happen.
  - Loss--if the risk becomes a reality, unwanted consequences or losses will occur).
- Different types/categories of risks are considered:
  - **Project risks**--threaten the project plan.
  - **Technical risks**--threaten product quality and the timeliness of the schedule.
  - **Business risks**--threaten the viability of the software to be built (market risks, strategic risks, management risks, budget risks).
  - **Known risks**--predictable from careful evaluation of current project plan and those extrapolated from past project experience.
  - **Unknown risks**--some problems simply occur without warning.

## Risks Identification

- Is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.).
- One method for identifying risk is to create risk item checklist. The check list can be used for risk identification and focuses on known and predictable risks in the following generic subcategories:
  - **Product size**--risks associated with the overall size of the software to be built or modified.
  - **Business impact**--risks associated with constraints imposed by management or the marketplace.
  - **Customer characteristics**--risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
  - **Process definition**--risks associated with the degree to which the software process has been defined and is followed by the development organization.
  - **Development environment**--risks associated with the availability and quality of the tools to be used to build the product.
  - **Technology to be built**--risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
  - **Staff size and experience**--risks associated with the overall technical and project experience of the software engineers who will do the work.

## Risks Projection

- Risk projection, also called risk estimation, attempts to rate each risk in two ways:
  - The likelihood or probability that the risk is real.
  - The consequences of the problems associated with the risk, should it occur.
- There are four risk projection steps:
  - Establish a scale that reflects the perceived likelihood of a risk
  - Delineate the consequences of the risk
  - Estimate the impact of the risk on the project and the product,
  - Note the overall accuracy of the risk projection so that there will be no misunderstandings.

## Building Risk Table

A risk table provides a project manager with a simple technique for risk projection.

- List all risks in the first column of the table.
- Classify each risk and enter the category label in column two.
- Determine a probability for each risk and enter it into column three.

- Enter the severity of each risk (negligible, marginal, critical, and catastrophic) in column four.
- Sort the table by probability and impact value.

Determine the criteria for deciding where the sorted table will be divided into the first priority concerns and the second priority concerns.

**Risks Refinement**

- During early stages of project planning, arisk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk.
- Process of restating the risks as a set of more detailed risks that will be easier to mitigate, monitor, and manage.
- CTC (condition-transition-consequence) format may be a good representation for the detailed risks (e.g. given that <condition> then there is a concern that (possibly) <consequence>).
- The consequences associated with refined sub-conditions helps to isolate the underlying risks and might lead to easier analysis and response.

**Risk Mitigation, Monitoring, and Management**

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy considers three issues: avoidance, monitoring, management and contingency planning.

- Risk mitigation (proactive planning for risk avoidance).
- Risk monitoring (assessing whether predicted risks occur or not, ensuring risk aversion steps are being properly applied, collect information for future risk analysis, attempt to determine which risks caused which problems).
- Risk management and contingency planning (actions to be taken in the event that mitigation steps have failed and the risk has become a live problem).