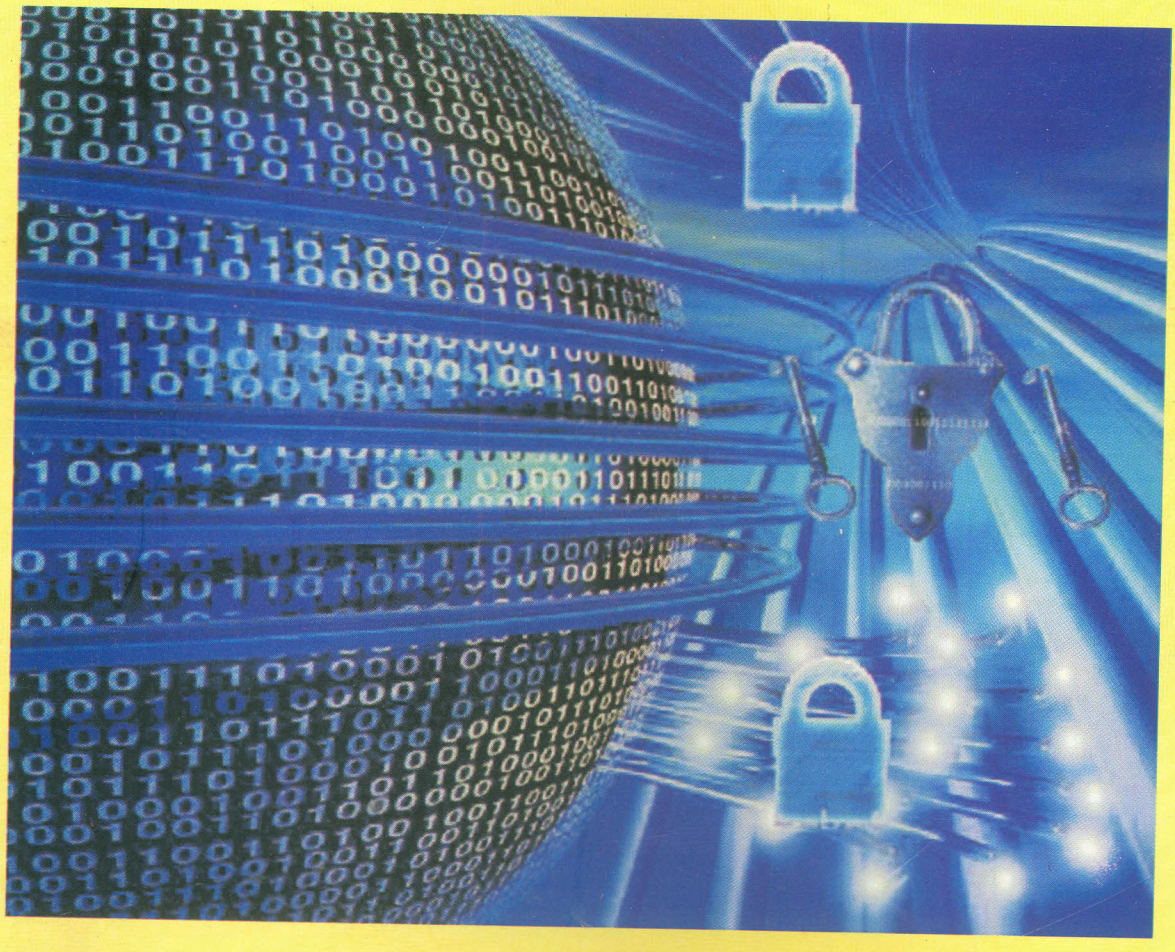
ignou
THE PEOPLE'S
UNIVERSITY

Indira Gandhi National Open University
School of Vocational Education and Training

MSEI-025
Application and Business Security Developments



Application Development Life Cycle

1

"शिक्षा मानव को बन्धनों से मुक्त करती है और आज के युग में तो यह लोकतंत्र की भावना का आधार भी है। जन्म तथा अन्य कारणों से उत्पन्न जाति एवं वर्गगत विषमताओं को दूर करते हुए मनुष्य को इन सबसे ऊपर उठाती है।"

– इन्दिरा गांधी

"Education is a liberating force, and in our age it is also a democratising force, cutting across the barriers of caste and class, smoothing out inequalities imposed by birth and other circumstances."

—Indira Gandhi

Indira Gandhi National Open University
School of Vocational Education and Training

MSEI-025
Application and Business Security Developments

Block

1

APPLICATION DEVELOPMENT LIFE CYCLE

Programme Expert/ Design Committee of Post Graduate Diploma in Information Security (PGDIS)

Prof. K.R. Srivathsan
Pro Vice-Chancellor, IGNOU

Mr. B.J. Srinath, Sr. Director & Scientist 'G',CERT-In, Department of Information Technology, Ministry of Communication and Information Technology Govt of India

Mr. A.S.A Krishnan, Director, Department of Information Technology, Cyber-Laws and E-Security Group, Ministry of Communication and Information Technology, Govt of India

Mr. S. Balasubramony, Dy. Superintendent of Police, CBI, Cyber Crime Investigation Cell, Delhi

Mr. B.V.C. Rao, Technical Director, National Informatics Centre, Ministry of Communication and Information Technology

Prof. M.N. Doja, Professor, Department of Computer Engineering, Jamia Milia Islamia
New Delhi

Dr. D.K. Lobiyal, Associate Professor, School of Computer and Systems Sciences, JNU
New Delhi

Mr. Omveer Singh, Scientist, CERT-In, Department of Information Technology, Cyber-Laws and E-Security Group, Ministry of Communication and Information Technology, Govt of India

Dr. Vivek Mudgil, Director, Eninov Systems
Noida

Mr. V.V.Subrahmanyam, Assistant Professor
School of Computer and Information Science
IGNOU

Mr. Anup Girdhar, CEO, Sedulity Solutions & Technologies, New Delhi

Prof. A.K. Saini, Professor, University School of Management Studies, Guru Gobind Singh Indraprastha University, Delhi

Mr. C.S. Rao, Technical Director in Cyber Security Division, National Informatics Centre, Ministry of Communication and Information Technology

Prof. C.G. Naidu, Director, School of Vocational Education & Training, IGNOU

Prof. Manohar Lal, Director, School of Computer and Information Science, IGNOU

Prof. K. Subramanian, Director, ACIIL, IGNOU
Former Deputy Director General, National Informatics Centre, Ministry of Communication and Information Technology, Govt. of India

Prof. K. Elumalai, Director, School of Law
IGNOU

Dr. A. Murali M Rao, Joint Director, Computer Division, IGNOU

Mr. P.V. Suresh, Sr. Assistant Professor, School of Computer and Information Science, IGNOU

Ms. Mansi Sharma, Assistant Professor, School of Law, IGNOU

Ms. Urshla Kant
Assistant Professor, School of Vocational Education & Training, IGNOU
Programme Coordinator

Block Preparation

Unit Writers

Mr. T. Lakshmana Kumar
M.Sc(IT)., PGDBM , Vijayawada.
(Unit 1&2)

Ms. Naina Kaushik
Assistant Professor, Computer Engineering Dept., Rajiv Gandhi Institute of technology Andheri(w), Mumbai (Unit 3)

Mr. Saoud Sarwar
Head , CSE Department, Al-Falah School of Engineering & Technology, Dhauj Faridabad (Unit 4)

Block Editor

Mr. P.V. Suresh
Sr. Assistant Professor
School of Computer and Information Science IGNOU

Ms. Urshla Kant
Assistant Professor
School of Vocational Education & Training
IGNOU

Proof Reading and Format Editing

Ms. Urshla Kant
Assistant Professor
School of Vocational Education & Training, IGNOU

PRODUCTION

Mr. B. Natrajan	Mr. Jitender Sethi	Mr. Hemant Parida
Dy. Registrar (Pub.)	Asstt. Registrar (Pub.)	Proof Reader
MPDD, IGNOU	MPDD, IGNOU	MPDD, IGNOU

Feb, 2012

ISBN: 978-81-266-5889-3

Further information on the Indira Gandhi National Open University courses may be obtained from the University's office at Maidan Garhi, New Delhi-110 068 or the website of IGNOU www.ignou.ac.in

Printed and Published on behalf of the Indira Gandhi National Open University, New Delhi, by the Registrar, MPDD.

Printed at: Berry Art Press A-9, Mayapuri Phase-I New Delhi-64

COURSE INTRODUCTION

This course talks about the application and business security developments. An application is a collection of programs that satisfies certain specific requirements and resolves certain problems. The solution could reside on any platform or combination of platforms, from a hardware or operating system point of view. This unit explains the development process for any application.

Application development is usually composed of the following phases, such as:

- Design phase
- Gather requirements.
    - User, hardware and software requirements
    - Perform analysis.
    - Develop the design in its various iterations:
        - High-level design
        - Detailed design
    - Hand over the design to application programmers.
- Code and test application.
- Perform user tests.
- User tests application for functionality and usability.
- Perform system tests.
    - Perform integration test (test application with other programs to verify that all programs continue to function as expected).
    - Perform performance (volume) test using production data.
- Go into production–hand off to operations.
- Ensure that all documentation is in place (user training, operation procedures).
- Maintenance phase--ongoing day-to-day changes and enhancements to application.

Application development involves the activities of planning, implementation, testing, documenting, deployment and maintenance. It is the process for creating something out of raw ideas and lead towards the problem solving application or software. It is a tedious process to develop software. Herein, iimplementation is adopted where software engineers actually program the code for the project. Then software testing is an integral and important phase of the software development process. This part of the process ensures that defects are recognized as soon as possible. Further, documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. This process deals with the authentication, data access, error handling, encryption, server configuration, security assessment and other important activities for the successful development of software.

This course includes the following blocks:

Block 1 – Application Development Life Cycle
Block 2 – Secure Application Development-I
Block 3 – Secure Application Development -II
Block 4 – Application Testing and Ethical Hacking

BLOCK INTRODUCTION

Many software development organizations, including many product and online services groups within Microsoft, use agile software development and management methods to build their applications. Historically, security has not been given the attention it needs when developing software with agile methods. Since agile methods focus on rapidly creating features that satisfy customers' direct needs, and security is a customer need, it's important that it not be overlooked. In today's highly interconnected world, where there are strong regulatory and privacy requirements to protect private data, security must be treated as a high priority. There is a perception today that agile methods do not create secure code, and, on further analysis, the perception is reality. There is very little "secure Agile" expertise available in the market today. This needs to change. But the only way the perception and reality can change is by actively taking steps to integrate security requirements into agile development methods. This block comprises of four units and is designed in the following way;

The **Unit One** helps you by explaining the importance of Analysis and Application Design. Design is a process of translating analysis model to design models that are further refined to produce detailed design models. The process of refinement is the process of elaboration to provide necessary details to the programmer. Data design deals with data structure's selection and design. Modularity of program increases maintainability and encourages parallel development. The aim of good modular design is to produce highly cohesive and loosely coupled modules. Independence among modules is central to modularity. Good user interface design helps software to interact effectively to external environment. Tips for good interface design helps designer to achieve effective user interface.

The **Unit Two** covers application coding. A coding standards document tells developers how they must write their code. We clarified why you need coding standards and also advantages of coding standards. We listed out the good methods for coding and also how to do effective source code control. This unit also gives you the rules for developing secured code. Here, we listed custom applications and their security threats, and also some General advice on securing custom applications.

Testing is a major component of software development, and is a major science in itself. **Unit three** focuses on application testing. Software testing is needed to verify and validate that the software that has been built has been built to meet these specifications. Testing ensures that what you get in the end is what you wanted to build. Testing enhances the integrity of a system by detecting deviations in design and errors in the system. Testing aims at detecting error-prone areas. This helps in the prevention of errors in a system. Testing also adds value to the product by conforming to the user requirements.

Unit four explains about the application production and maintenance. This unit will discuss what maintenance is, its role in the software development process, how it is carried out, and its role in iterative development, agile development, component-based development and open source development.

Hope you benefit from this block.

ACKNOWLEDGEMENT

The material we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

UNIT 1 ANALYSIS AND APPLICATION DESIGN

Structure

1.0 INTRODUCTION

Software development consists of the first phase called software requirements. To develop any software first its requirement is to be taken care off. Software requirement is to be analyzed and modeled where application design plays an important role.

Application design is the last software engineering action within the modeling activity. Application design compromises of the code generation and testing activities which are collectively known as construction. Therefore software

development consist of the analysis and modeling phase where in modeling the application design takes place.

The foundation of all elements in Design starts with the consideration of data. The next step is the derivation of the architecture and lastly all the design tasks are performed. Application designing is as simple as architect's plans for a house. To start with, all the things to be built and slowly progressing towards refining the things, progressing the construction, and taking care of each detail. Similarly, the design model that is created for software provides a variety of different views of the system.

Designing of software falls in the technical aspect of software engineering like Kernel. Application design is applied to every software process model as design is a core engineering activity. Design engineering compromises the set of principles, concepts and practices that constructs high quality system or product. The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight.

1.1 OBJECTIVES

After going through this unit, you should be able to explain:

- Analysis
- design engineering concepts;
- architectural design;
- data design;
- object-oriented design; and
- user interface design

1.2 ANALYSIS

Analysis is the process of breaking a complex topic or substance into smaller parts to gain a better understanding of it.. It is the examination and evaluation of the relevant information to select the best course of action from among various alternatives.

1.2.1 Requirements Analysis

Encompasses those tasks that go into determining the needs or conditions to m eet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a development project. Requirements must be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Requirements can be architectural, structural, behavioral, functional, and non-functional.

1.2.2 Different Ways of Performing Requirements Analysis

Brainstorm sessions bring together a set of design and task experts to inspire each other in the creative, idea generation phase of the problem solving process. They are used to generate new ideas by freeing the mind to accept any idea that is suggested, thus allowing freedom for creativity. The method has been widely used the early phases of design. The results of a brainstorming session are, it is hoped, a set of good ideas and a general feel for the solution area to meet user needs.

Card sorting is a technique for uncovering the hierarchical structure in a set of concepts by asking users to group items written on a set of cards. This is often used, for instance, to work out the organisation of a website. Users would be given cards with the names of the intended web pages on the site and asked to group the cards into related categories. After gathering the groupings from several users, designers can typically spot clear structures across many users. Statistical analysis can uncover the best groupings from the data where it is not clear by inspection. IBM (2002) is an example of an analysis programme.

Affinity diagramming is a related technique that can be used for organising the structure of a new system, and allows participants to work as a group. Designers or users write down items such as potential screens or functions on sticky notes and then organise the notes by grouping them, to uncover the structure and relationships in a domain. Affinity diagrams are often a good next step after a brainstorming session. See Beyer & Holtzblatt (1998) for more information.

Storyboards, also termed "Presentation Scenarios", are sequences of images that show the relationship between user actions or inputs and system outputs. A typical storyboard will contain a number of images depicting features such as menus, dialogue boxes and windows. Storyboard sequences provide a platform for exploring and refining user requirements options via a static representation of the future system by showing them to potential users and members of a design team (Andriole, 1989).

Prototyping is where designers create paper or software-based simulations of user interface elements (menus, buttons, icons, windows, dialogue sequences, etc.) in a static or dynamic way. When a ***paper prototype*** has been prepared, a member of the design team sits before a user and 'plays the computer' by moving the paper and card interface elements around in response to the user's actions. The difficulties encountered by the user and user comments, are recorded by an observer. ***Software prototypes*** provide a greater level of realism than is normally possible with simple paper mock- ups. Here, the aim is to create a rapid prototype that is used to establish an acceptable design for the user but is then thrown away prior to full implementation. Some design processes are based on a rapid application development (RAD) approach. Here a small group of designers and users work intensively on a prototype, making frequent changes in response to user comment. The prototype evolves into the full system. Hall (2001) discusses the merits and cost-benefits of varying fidelity levels of prototypes.

Allocation of function is an important element for many systems. As ISO 13407 (1999) states in clause 7.3.2, allocation of function is "the division of system tasks into those performed by humans and those performed by technology" to specify a clear system boundary. A range of options is established to identify the optimal division of labour, to provide job satisfaction and efficient operation of the whole work process. **User cost- benefit analysis** can then be carried out to determine how acceptable each user group will find the new arrangement. The use of task allocation charts and cost-benefit analysis is most useful for systems that affect whole work processes rather than single user, single task products. They also provide the opportunity to rethink the system design or user roles to provide a more acceptable solution for all groups. A process for performing a user cost- benefit analysis is described by Eason (1988).

Design guidelines and standards are referred to by designers and HCI specialists for guidance on ergonomic issues associated with the system being developed. The ISO 9241 standard (ISO, 1997) covers many aspects of hardware and software user-interface design, and contains a widely agreed body of software ergonomics advice. See Bevan (2001) for more information on ISO standards. Style guides embody good practice in interface design. Following a style guide will increase the consistency between screens and can reduce the development time. For a GUI (graphic user interface) an operating.

Stakeholder interviews

Stakeholder interviews are a common technique used in requirement analysis. Though they are generally idiosyncratic in nature and focused upon the perspectives and perceived needs of the stakeholder, very often without larger

enterprise or system context, this perspective deficiency has the general advantage of obtaining a much richer understanding of the stakeholder's unique business processes, decision-relevant business rules, and perceived needs. Consequently this technique can serve as a means of obtaining the highly focused knowledge that is often not elicited in Joint Requirements Development sessions, where the stakeholder's attention is compelled to assume a more cross-functional context. Moreover, the in-person nature of the interviews provides a more relaxed environment where lines of thought may be explored at length.

Joint Requirements Development (JRD) Sessions

Requirements often have cross-functional implications that are unknown to individual stakeholders and often missed or incompletely defined during stakeholder interviews. These cross-functional implications can be elicited by conducting JRD sessions in a controlled environment, facilitated by a trained facilitator, wherein stakeholders participate in discussions to elicit requirements, analyze their details and uncover cross-functional implications. A dedicated scribe and Business Analyst should be present to document the discussion. Utilizing the skills of a trained facilitator to guide the discussion frees the Business Analyst to focus on the requirements definition process.

JRD Sessions are analogous to Joint Application Design Sessions. In the former, the sessions elicit requirements that guide design, whereas the latter elicit the specific design features to be implemented in satisfaction of elicited requirements.

System style guide should be followed to implement good practice and to provide consistency. For websites, design guidelines are evolving but good web design principles are gradually being established (Nielsen, 2000). Nicolle and Abascal (2001) discuss issues and present guidelines to make systems accessible by people with disabilities.

Parallel design sessions involve a few small groups of designers working independently, to generate a range of diverse solutions. The aim is to develop and evaluate different system designs before choosing a solution (possibly drawing from several solutions) as a basis for the implemented system.

1.3 DESIGN PROCESS AND DESIGN QUALITY

Design Process

Design process is a step-by-step and repetitive procedure where software is simulated according to the requirements. The simulation depicts a holistic view of software and high level of abstraction. The design represented by this

level can be directly traced to the specific system objective and requirements such as data, functional and behavioral requirements. Design is an iterative process which leads to the refined design representation at lower level of abstraction.

As said earlier, design process is required for better quality software. This quality is achieved by a series of formal technical reviews. Three characteristics are important in constructing a good design:

- Design must consider and fulfill the entire customer's requirement that are implicit and must implement explicit requirements contained in the analysis model.
- The design should be so made that it can be used and understandable by the lay man, generator of code and testers.
- The design should provide a clear picture of the software, addressing the data, functional and behavioral domains from an implementation perspective.
- All the above characteristics help in achieving the goal of design process. The goals can be achieved by design quality.

(a) Design Quality

There are few guidelines which are the characteristics of a good design. They are followed to evaluate the quality of a design. Some criteria's such as technical criteria is established for good design:

1. A design should exhibit an architecture that has
   (a) Creation by recognizable architectural styles or patterns,
   (b) Components leading to good design characteristics, and
   (c) Evolutionary fashion which facilitates implementation and testing.
2. A design should be module based, that is, it contains partitions of elements or subsystems.
3. A design should have different representations of data, architecture, interfaces, and components.
4. A design should explore the data structures useful for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should consist of components that have independent functions characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be achieved from an iterative method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Design guidelines should be strictly followed to achieve the goal. Design engineering encourages good design through the application of fundamental design principles, systematic methodology and through review.

1.4 CHARACTERISTICS OF GOOD DESIGN PROCESS

Following are few important guidelines often referred as characteristics of good design:

- The notion of design should reflect its meaning.
- Design should be implemented based on the information obtained from the requirement analysis phase of the software development process.
- The design should clearly represent the interface applicable in the system. This causes ease while providing connections between various modules of the software as well as between the software and its vicinity of implementation
- The design should consist of all the independent modules (i.e. the modules capable of functioning independently).
- The design should exhibit clearly all the modules of the software.
- Various elements such as data, architecture, interfaces and components, should be distinguished clearly.
- Using this design, a software engineer should directly build data structure which is suitable for the implementation classes. Also these designs should be outcome of easily recognizable components.
- A design should describe architecture, should encompass several components and should reflect dynamic behavior etc.

1.5 DESIGN ENGINEERING CONCEPTS

(i) Abstraction (ii) Architecture

(iii) Patterns (iv) Modularity

(v) Information hiding (vi) Functional independence

(vii) Refinement (viii) Refactoring

(i) Abstraction

Abstraction is a means of describing a program function, at an appropriate level of detail. At the highest level of abstraction a solution is stated in the language of the problem environment 9requirements analysis). At the lowest level of abstraction, implementation-oriented terminology is used (programming). An abstraction can be compared to a model which incorporates detail only to the extent needed to fulfill it s purpose.

Abstraction is a very powerful concept which is used in all engineering disciplines. It is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component. Any component or system provides some services to its environment. An abstraction of a component describes the external behavior of that component without bothering with the internal details that caused the behavior.

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. To modify a system, the first step is understanding what the system does and how. The process of comprehending an existing system involves identifying the abstractions of subsystems and components from the details of their implementations. Using these abstractions, the behavior of the entire system can be analyzed. This also helps determine how modifying a component affects the system.

During requirements definition and design, phase abstraction permits separation of the conceptual aspects of a system from the (yet to be specified) implementation details.

For example: We can specify the FIFO property of a stack without concern for the representation scheme to be used in implementing the stack or queue.

Similarly, we can specify the functional characteristics of the routines that manipulate data structures (Example: NEW, PUSH, POP, TOP, EMPTY) without concern for the algorithmic details of the routines. Three widely sued abstraction mechanisms in application design are:

a) Functional Abstraction

b) Data Abstraction, and

c) Control Abstraction.

a) **Functional Abstraction:** in functional abstraction, a module is specified by the function it performs.

For example: A module to compute the log of a value can be abstractly represented by the function log.

Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the composition of the system is in terms of functional modules.

b) **Data Abstraction:** It involves specifying a data type or a data object by specifying legal operations on objects; representation and manipulation details are suppressed. Thus the type "stack" can be specified abstract as a LIFO mechanism in which the routines NEW, PUSH, POP, TOP and EMPTY interact. In data abstraction data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects from outside an object, the internal details of the object are hidden; only the operations on the object are visible. Data abstraction forms the basis for object-oriented design.

c) **Control Abstraction:** Control abstraction is the third commonly used abstraction mechanism in application design. Control abstraction is used to state a desired effect without stating the exact mechanism of control. IF statements and WHILE statement in modern programming languages are abstractions of machine code implementations that involve conditional jump instructions. A statement of the form "for all I in S sort files I" leaves unspecified sorting technique, the nature of S, the nature of the files, and how "for all I in S" is to be handled. Another example of control abstraction is the monitor construct, which is a control abstraction for concurrent programming; implementation details of the operator are hidden inside the construct. At the architectural design level, control abstraction permits specification of sequential subprograms, exception handlers, and courtliness and concurrent program units without concern for the exact details of implementation.

(ii) Architecture

The architecture of the procedural and data elements of a design represents a software solution for the real world problem defined by the requirements analysis. Software architecture alludes to two important characteristics of a computer program.

1. The hierarchical structure of procedural components.

2. The structure of data.

Software architecture is derived through a partitioning process that relates elements of a software solution to parts of a real-worlds problem implicitly, defined during requirements analysis. The evolution of software and data structure begins with a problem definition. The solution occurs when each part of the problem is solved by one or more software elements.

(iii) Patterns

Pattern forms an instance of the original design which includes certain specifications through which we can judge its implementation and various other details. Hence, by developing patterns a designer can determine whether the specifications included in these patterns are satisfying the requirement or whether the given patterns is reusable in other applications or whether it can form as a sample for developing other applications etc.,

(iv) Modularity

The main concept behind partitioning can be visualized, if a system is partitioned into modules so that the modules are solvable and modifiable separately, they are separately compiled (then changes in a module will not require recompiled of the whole system) etc., A system is considered modular if it consists of discrete components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

Modularity is a desirable property in a system. It helps in system debugging, isolating the system problem is easier changing a part of the system is easy and in system building a modular system can be easily built by "putting its modules together"

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface, through which it can interact with other modules. Modularity is where abstraction and partitioning come together. For easily understandable and maintainable system, modularity is clearly the basic objective; partitioning and abstraction can be viewed as concepts that help to achieve modularity.

(v) Information Hiding

Information hiding is a fundamental design concept for software. The principle of information hiding was formulated by Mr. Parnas. When a software system is designed using the information hiding approach, each module in the system hides the internal details of its processing activities and modules communicates only through well-defined interfaces.

According to Mr. Parnas, design should begin with a list of difficult design decisions and design decisions that are likely to change. Each module is designed to hide such a decision from the other modules. Because these design decisions transcend execution time, design modules may not correspond to processing steps in the implementation of the system. In addition to hiding of difficult and changeable design decisions, other candidates for information hiding include:

(a) A data structure, its internal linkage, and the implementation details of the procedures that manipulate it (this is the principle of data abstraction).

(b) The format of control blocks such as those for queues in an operating system (a "control block" module).

(c) Character codes, ordering of character sets, and other implementation details.

(d) Shifting, masking, and other machine dependent details.

Information hiding can be used as the principal design technique for architectural design of a system, or as a modularization criterion in conjunction with other design techniques.

(vi) Functional Independence

Functional independence usually refers to a feature of software, where the developers limit cohesion between various modules of software and make them to function independently. Hence, in this case independent functioning of each modules of software is of primary importance.

Functional independence is often advantageous since it causes ease during the development of software (as it easy to develop software modules performing specific task). Also, sing this feature, interfaces can be easily defined between these modules without much difficulty.

(vii) Refinement

Step-wise refinement is a top-down technique for decomposing a system from high-level specifications into more elementary levels. Step-wise refinement is also known as "Step-wise Program Development" and "Successive Refinement". Step-wise refinement involves the following activities:

(a) Decomposing design decisions to elementary levels.

(b) Isolating design aspects that is not truly interdependent.

(c) Postponing decisions concerning relative to representation details as long as possible.

(d) Carefully demonstrating that each successive step in the refinement process is a faithful expansion of previous steps.

The Step-wise refinement technique breaks the logic design problem into a series of steps, so that the development can be done gradually. The process starts by covering the specifications of the module into abstract description of an algorithm containing a few abstract statements. In each step, one or several statements in the algorithm (developed so far) are decomposed into more detailed instructions. The successive refinement terminates when all instructions are sufficiently precise and can easily be converted into programming language statements. During refinement, both data and instructions have to be refined. A guideline for refinement is that, in each step, the amount of decomposition should be such, that it (refinement process) can be easily handled and represent one or two design decisions.

(viii) Refactoring

Refactoring is a process of improving the existing software without altering its internal mechanisms. Hence, once given software is said to be refactored it means that, all the loopholes existing in it such as inappropriate data structures or inefficient algorithms or certain redundant data etc., can be exploited and what remains is the code with improved capabilities.

Check Your Progress 1

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) Architectural Design displays the _____ of computer-based system.

.....

.....

2) _____ is an step-by-step and repetitive procedure where software is simulated according to the requirements.

.....

.....

3) _____ is a means of describing a program function, at an appropriate level of detail.

.....

.....

1.6 CHARACTERISTICS OF WELL FORMED DESIGN

The four characteristics of well-formed design classes are discussed below:

Complete and Sufficient

When the word complete refers to the design class it means that the design class should include the entire entities (attributes and methods) essential to exist in the situation which is being described by a given design class.

Sufficiency is a relative term which measures exact (neither more nor less) nature of a given design. Hence, a well formed design class usually possess both the properties i.e. complete and sufficiency respectively.

High Degree of Cohesion

This feature reflects that, whenever a given design class is implemented to represent a specific purpose, then the attributes and methods, associated with that class should also reflect the same purpose. Till this condition is satisfied, we refer that these classes are maintaining high degree of cohesion property.

Low Degree of Coupling

It is often a true fact that, in a given model the design classes should have collaboration. But this collaboration should not be extended to extreme values. This is because, a model of design classes possessing high degree o collaboration is difficult to rest, maintain etc., also in this case, we need to consider even the "Law of Diameter" which suggests that, when there are multiple subsystems, then various methods belonging to single subsystem should communicate by transmitting messages at the same time restricting the methods of one class to communicate with the methods of another classes where the two classes are belonging to different subsystems.

Primitiveness

It suggests that any method belonging to a given design class should be made to handle only single purpose. Hence, once the method implements a given purpose, the same method should not be used for the implementation of any other purpose.

1.7 DESIGN MODEL

A design model here, depicts a two dimensional analogy i.e., it represents process dimension along horizontal x-axis and abstraction dimension along vertical y-axis.

The two dimensional representation of the design is show above. The process dimension along horizontal direction depicts the stages of development (i.e., architectural elements, interface element, component level element s and finally deployment level elements respectively) which is usually observed during software development process. Now consider the vertical view of this

diagram. It vertical view refers to abstraction dimension. Here, short details on the transformation of elements from analysis model to design model are also shown. Apart from this, the elements which undergo refinement with the design model are also shown. Usually the refinement is performed in a sequential fashion. The horizontal dashed line appearing at the mid of the diagram is used to separate the analysis model from design model. Usually in both of these diagrams we find the UML diagrams. The UML diagrams appearing in the design models are completely refined when compared to the diagrams in analysis model. This refinement can be in terms of architectural structure and style, higher implementation details a component that resides within the architect etc.

Finally, while dealing with the process dimension one has to remember that, the architectural design, interface design and component level designs are developed parallel and the deployment level diagrams is obtained at the end i.e. deployment level diagram is considered when we are completed with the entire designing process.

1.8 DESIGN ELEMENTS IN DESIGN MODEL

- **Data Design Elements**
- **Architectural Design Elements**
- **Interface Design Elements**
- **Component – level design Elements**
- **Deployment – level Design Elements**

(i) **Data Design Elements:** Data design often leads to a systematic representation of data in most abstract form. Later by applying refining tools the given data is progressed into a descent format which can be easily processed by any of the computer based systems. While dealing the data design, one has to remember that the format (rather design) of data (on which we are currently working exerts significant impact on the software which is going to be developed. This can be analyzed by considering the following states of development.

(a) **Component Level:** At this level we generally concentrate on the available forms of data like data structures and is corresponding algorithms. This turns out to be necessary, since quality of end product (software) depends primarily on this aspect of component level.

(b) **Application Level:** At this level, our attention shifts onto storage aspects i.e., converting the given data model into database. This

remains important to achieve the commercial objectives, which were estimated n the product.

(c) **Business or Commercial Level :** Once the data is stored in the database, now we concentrate largely on improving its storage aspects i.e., acquiring the required data with less effort applied. This can be achieved by data ware housing mechanisms.

(ii) **Architectural Design Elements:** Architectural design is nothing but a photocopy of the end software which is going to be developed. It is usually designed while considering following aspects:

- The knowledge on the application domain.
- By considering the analysis classes, their associated relationships and also the collaborations existing between them.
- By considering various architectural pattern sets etc.,

(iii) **Interface Design Elements:** In general sense interface are just like wiring provided in a given circuit i.e., it refers to paths or directions in which the given information proceed. Basically these are three types of interface design elements:

- The User Interface
- External Interface
- Internal Interface

The User Interface accommodates following elements:

- Technical
- Aesthetic, and
- Ergonomic respectively.

The external interface defines flow of information between the components of two independent systems. The information related to these interfaces is collected during requirement analysis phase. It is often recommended to check these interfaces before the initiation of interface design. As the information in this case usually flows into other system, hence, it should include security as well as certain error checking measures.

Internal Interface defines flow of information between various components of a single system.

Representation of interfaces is same as what used in UML. Hence, consider the following diagram.

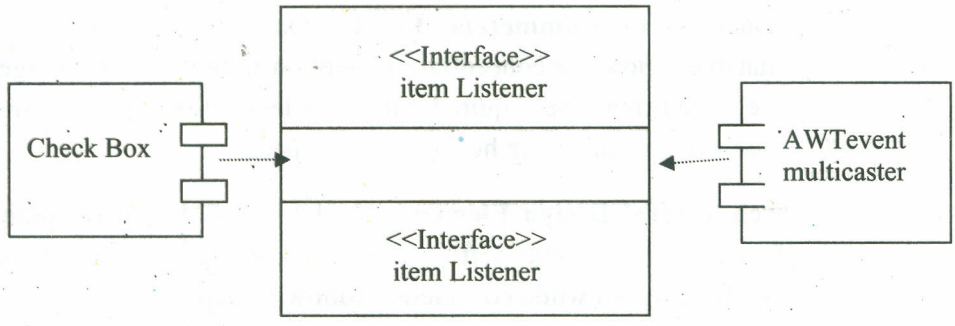


Fig. 1: Representation of Interface in terms of UML Notation

As shown in the above figure, an interface is usually represented in the form of a rectangle with a key word **"interface"** inscribed in the first partition of the icon. Apart from this, the two icons represented on the either sides of the interface or component. An interface can also be represented as a small circle.

(iv) **Component Level Design Element:** As the name suggests, component level design provides all details of a given software component. In order to achieve this, the component design describes the representation of data structure for orderly placement of objects, suitable algorithms to promote processing internal to a given component and finally defines interface.

The icon used to represent a component is given below:

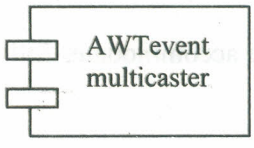


Fig. 2: Component Icon

Name of the component is inscribed in the icon component along with interface is shown below:

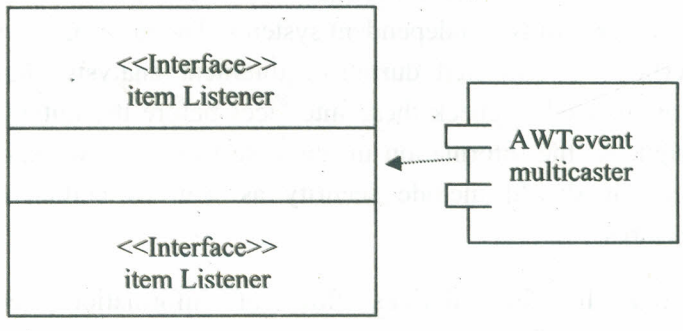


Fig. 3: Component along with an Interface

(v) **Deployment Level Design Elements:** This is usually the final design among all other designs narrated above. It usually describes the scenario of software functionality when all the subsystems are released to perform ascertained functionality in real environment. In order to design such instances, we usually resort to UML's deployment diagram. Hence, consider an example diagram in this regard.

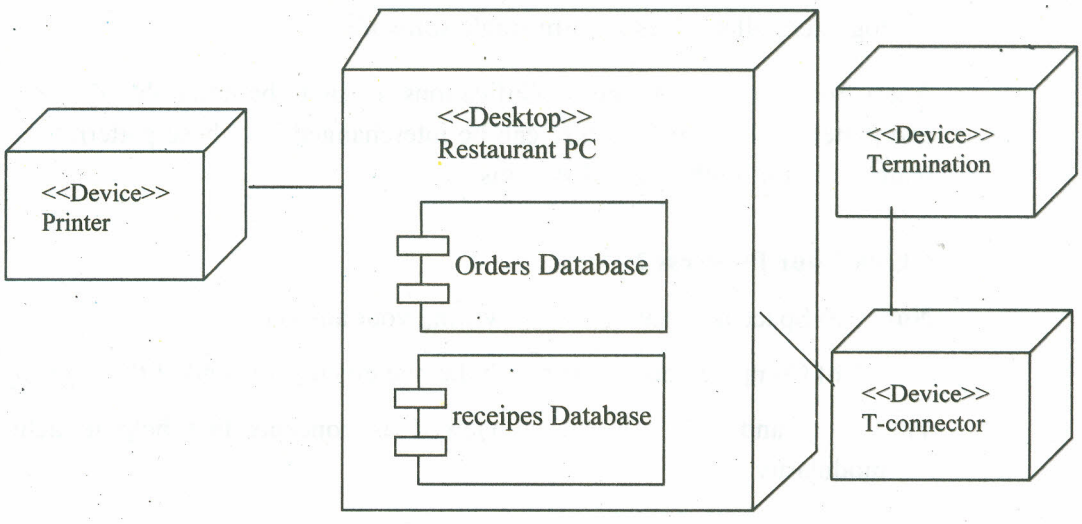


Fig. 4 : The above Deployment Diagram reflects only a Single Instance of Restaurant PC System

1.9 ARCHITECTURAL DESIGN

Software Architecture

Software architecture is a combination of certain number of components, along with their relationships as well as their visible features.

A software engineer uses software architecture in many ways i.e., a software architecture provides him specifications, through which he sets his goals, it provides him information related to modifications to be made to the design at the initial software development states etc., Hence, h can get rid of potential risks easily at the beginning stages of software development only.

As we are known to the definition of software architecture, now let us traverse through its importance.

Importance of Software Architecture

With above illustrations we can predict the role of architecture in the software development. Apart from this following summary provides importance of software architecture in the real world software development aspects.

- Software architecture describes a scenario, through with various specimens showing their interests in the software can easily communicate with each other.
- It provides an overview of each software development aspect which remains important for overall success of the software being developed.
- Finally it describes the structure and working of various entities which together collaborates to form single software.

Apart from above mentioned specifications, it has to be noted that the designs and patterns of the architecture can be interchanged i.e., these patterns can be matched to form other useful systems.

Check Your Progress 2

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) _____ and _____can be viewed as concepts that help to achieve modularity.

.....

.....

2) What are the activities involved in Step-wise refinement?

.....

.....

3) ____ degree of Cohesion and ____ degree of coupling is desirable.

.....

.....

4) _____ describes the structure and working of various entities which together collaborates to form single software.

.....

.....

1.10 DATA DESIGN AT THE ARCHITECTURAL LEVEL AND THE COMPONENT LEVEL

Before going into the discussion of data design at the architectural and component level, consider the definition of data design.

Data design refers to a criterion which is useful in transforming data objects into data structures, further into database architecture.

Now consider the following aspects.

(1) Data Design at the Architecture Level

In today's world whenever we look through any business enterprises either big or small, we find that these enterprises maintain large sets of databases. In such circumstances it often crucial to acquire useful information from such database especially when each of these databases is not correlated. Hence, to manage such circumstances, several researchers had put forth their efforts and came out with data mining techniques, like knowledge discovery in databases or in short KDD. But due to many reasons, these researchers turned their attentions towards other techniques due to runtime failure of KDD. This is because, due to the existence of large number of databases, their details of storage, their structures etc., differs significantly causing failure of KDD. Now-a-days each of the modern business enterprises immensely depend on new type of aspect referred as "Dataware House".

(2) Data Design at Component Level

Whenever we deal with data designing at component level, it reflects representation of data structures accessible by various components forming giving software. In order to do that following are certain important principles favoring it.

Principle – 1

"An application design and programming language should support the specification and realization of abstract data types"

Explanation: At time we come across the most sophisticated data structures for which its implementation goes to highest level of complexity. This usually happens since the programming language utilized for implementation of such data structures does not include mechanisms to support it.

Principle – 2

"A library of useful data structures and the operations that may be applied to there should be developed".

Principle -3

"Low-level data design decisions should be deferred until late in the design process".

Explanation: We can often resort to sequential or step-wise refinement terminology while

Restoring to designing of data. This can be done as follows:

Step – 1: Begin in process of data organization at requirement analysis phase.

Step – 2: Refine this organized data at data designing phase.

Step – 3: Include the details of refined data at final or component level phase.

Principle – 4

"The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure".

Explanation: The above principle refers to two important ideologies i.e., information hiding and coupling respectively. These two concepts hen implemented in current data designing, cause the resultant software to be of high quality.

Principle – 5

"All data structures and the operations to be performed on each should be identified".

Explanation: This phenomenon can be observed in class diagrams, where it defines it data items also various types of processing applied to these items.

Principle – 6

"The systematic analysis principles applied to function and behaviours should also be applied to data".

Explanation: This systematic analysis consists of following steps.

Step -1: It initially begins by developing the representations consisting of data flow and various contents. Later it is also reviewed.

Step- 2: Identify the data objects.

Step -3: It there exists any alternative data organizations, consider them.

Step-4: Finally effect of data modeling over the application design should be determined.

1.10.1 Architectural Styles

With an intention of providing suitable structure to various component of a given system, architectural style causes existing design of given system to be transformed. It may sometimes happens that the reengineered system is required to be applied with several architectural styles, in such occasions many fundamental changes are required to be adhered. The changes can even be adhering of appropriate functionality to the components resident of a given system. These architectural styles in terms of computer-based system are nothing but a system category consisting of:

- Certain number of connectors facilitating co-ordination, co-operation as well as communication among various components forming a system.
- Certain number of components capable of providing definite functionality.
- Semantic models.
- Constraints which refer to integration of a system etc.,

1.10.2 Architectural Pattern

Architectural pattern is bit analogous to architectural style. Following are certain illustrations through which we can distinguish architecture pattern with architecture style.

- Pattern usually considers only single aspect of the entire architecture.
- It specifies certain rules which are to be obeyed by a given architecture. Hence, this feature of pattern dictates the given software (on which it is laid) is going to function at given infrastructure level.
- It exposes only certain behavioral facts of the architecture.

Finally, one can consider architecture pattern along with architectural styles in order to provide suitable shape to the existing software.

1.10.3 Data Centered Architecture

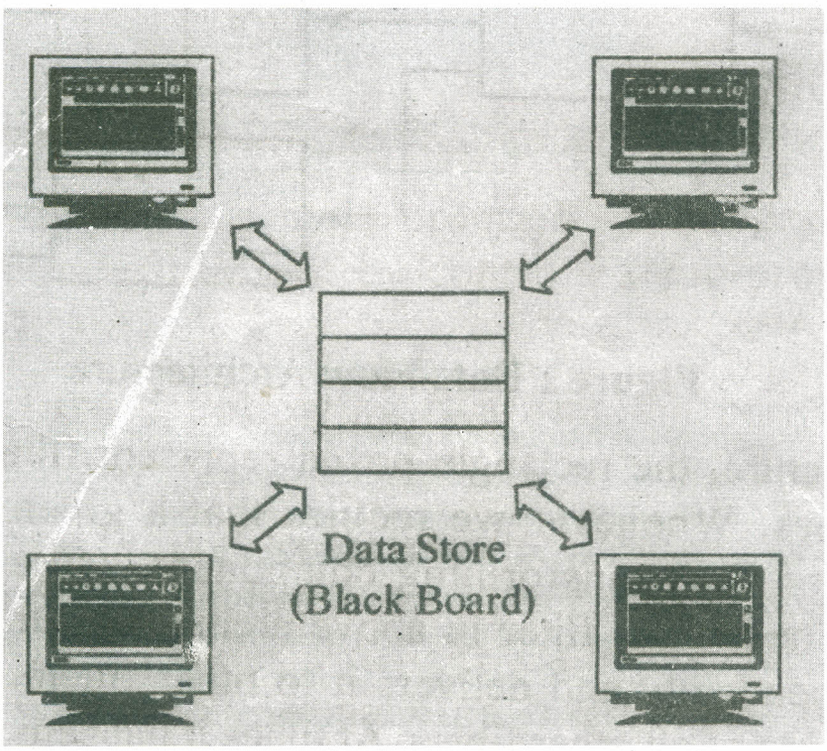


Fig. 5: Data Centered Architecture

Important points related to above architecture are illustrated below:

- In the above architecture, essential data reside at the centre of the architecture.
- All the client softwares are authorized to access this data.
- These client softwares can easily manipulate the centered data i.e., they c an delete, update, and etc.,
- As the data manipulation can be done independently. Hence, the centered data can be transformed into what called as "blackboard" due to which a message is supplied to client if he/she intends to access the data which is already been transformed by other clients.
- The above architecture is said to support integrability which is due to the fact that the architectures is promoting independent accessing and manipulation of data store.
- All the client software are authorized to access several of their processes independently, also as the data store can also act as a black board, these clients can send and receive messages among themselves.

1.10.4 Data Flow Architecture

Diagrammatic representation of data flow architecture is shown below:

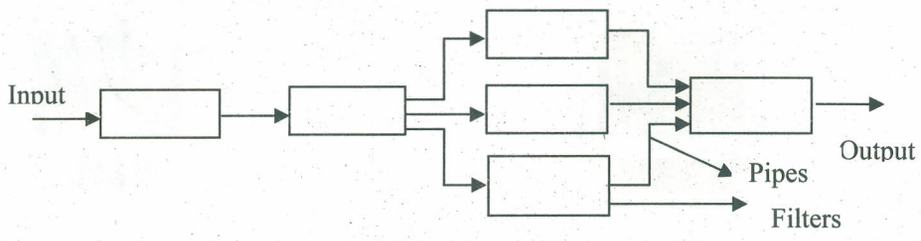


Fig. 6: Data Flow Architecture

In the above architecture, the rectangle boxed represent filters and the arrows connecting these filters represent pipes. Whenever we require that a given data to be transformed into certain output by means of few transforming components (filters), in those conditions data flow architecture is prescribed. Each filter in above architecture transforms the data received by it into certain output independently and delivers it t other filters. Also each of these filters are independent of the functioning of all other filters. At times it happens that a condition is met referred as batch sequential, in which data flowing gets degenerated into a single line transforms. In those conditions, the architecture accepts the data and transforms it into certain output using one or more, of these filters.

1.10.5 Call and Return Architecture and Layered Architecture

(a) The Call and Return Architecture

The call and return architecture is best known to frame, given software such that the resultant architecture can be modified or shrunk depending on the requirements. This architecture has got two sub-architectures referred as main program/subprogram architecture and remote procedure call architecture.

Main Program/Sub-Program Architecture: As the name suggests, this architecture causes decomposition of main program into a number of constituent program components. These program components can still be decomposed into further components. Hence, this mechanism can be named as transformation of function into control hierarchy structure which is diagrammatically represented below:

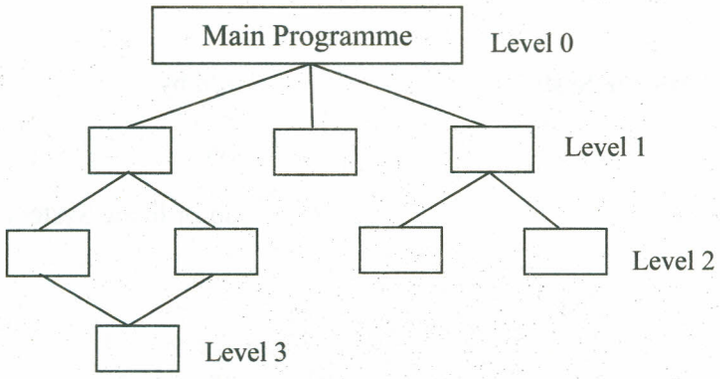


Fig. 7: Main Program/Sub-Program Architecture

The above diagram depicts the main program/Sub-Program architecture. The level-0 consists of main program followed by controlled sub-program at level-1, application sub-program at level -2,3…..

(b) Layered Architecture: Following is a diagram providing a preview of layered architecture.

The layered architecture is represented above. These are basically four layers used. The rectangular small boxes represented in each layer are nothing but the components at each layer. Each of these layers defines a definite set of operations. As we go deep inside the layers, we become nearer to the machine instruction. The outermost layer is user interface layer, where several components exhibit operations related to user inters. By moving further we encounter application layer, utility layer and core layer respectively.

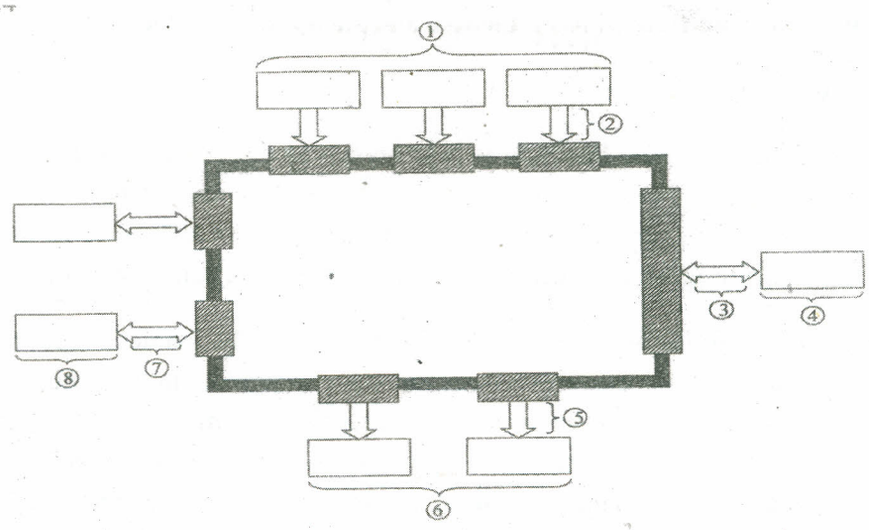


Fig. 8: Architectural Context Diagram

Components

1. Super-ordinate System
2. Used by
3. Uses
4. Peers
5. Depends on
6. Sub-ordinate System
7. Uses
8. Actors

Details on essential components of above system are given below:

Actors

These are nothing but specimens or any entities possessing a definite set of roles and interacting with the system. During this interaction an actor can either provide or accept information from the system.

Sub-ordinate Systems

These are the systems which function along with the target systems. Hence, supporting

The target system in successfully completing its processing.

Super-ordinate Systems

These are the systems which consider the target systems in order to complete few of its higher valued activities.

Peer Systems or Peers

These are the system which directly interacts with the target system (same as client-server interaction).

1.11 OBJECT-ORIENTED DESIGN

It deals with the designing aspects of a software project to be developed. Here, the objects quote the already analyzed requirement of current software project. In this case objects can be of two types i.e. solution object and problem objects. The problem objects refer to the expected problem of the system and solution objects intern signifies solution to these problem objects. But to generalize design, the designers while implementing new objects, strives to transform the problem objects to solution objects.

Following are essential steps required during the designing of a project using object oriented designing concepts.

Step – 1: Initially analyze the project completely. This includes defining its context and various modes of usage of the given system.

Step – 2: It deals with the designing of the architecture of the system to be developed.

Step – 3: Recognize the essential objects accommodating the system

Step – 4: Generate the equivalent design model.

Step – 5: End up the task by adhering suitable interfaces between objects.

While traversing through the above mentioned activities it can be analyzed that each activity is related to its next higher activity.

As and when the major objects are recognized, they are adhered with interface so as to make the architecture compatible. A further development of design model (in step 4) brings maturity to the architecture. Hence, step 4 can also be referred as "refinement of architecture". By looking into above mentioned specification, one would refer that the designing aspects may often easy. But, it is not true practically. Since a designer may get exposed to several critical measures for which he may return back and restart the designing process again or he may also postpone many of its activities if the implementation of above steps, does not advances proper results.

Example: Following is a layered architecture depicting the weather mapping system developed by following above mentioned criteria.

Before sorting to the designing aspects of this system it is often important to generalize the details of weather mapping system.

Weather Mapping System

Weather Mapping System is an effective system for generating weather reports.

Usually the data is essential in generating these reports which is collected from various sources such as

- Satellite
- Local Weather Reporting Stations
- Weather Observes etc.,

Further this information is passed to a special area computer which accepts them only after they had passed through validation procedures. Moreover, this entire data is integrated and backed. Later, using suitable digital map data base, various maps are developed through a special map printer.

Now, basing on this information the layered architecture for weather mapping system can be developed as follows:

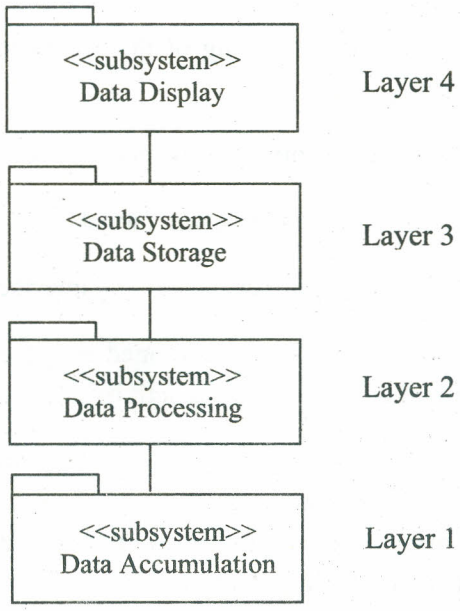


Fig. 9: Diagram Depicting the Layered Architecture (Weather Mapping System)

The icon << >> refers to a package in UML notation. It generally describes the behavioral aspects of a given system by including suitable name (narrating the situation) inside each package. Each layer in the above architecture has its own specification and importance.

Hence, the layers described above provide the information on every aspect of weather mapping system.

For example

Layer 1: Describes the process of accumulation of data from different sources.

Layer 2: It describes the way the accumulated data is processed

Layer 3: Resorts to storage of data.

Layer 4: Gives the information on displaying data i.e. creation of maps etc.

Now, consider the diagram depicting various subsystems (systems associated with each system described in above architecture).

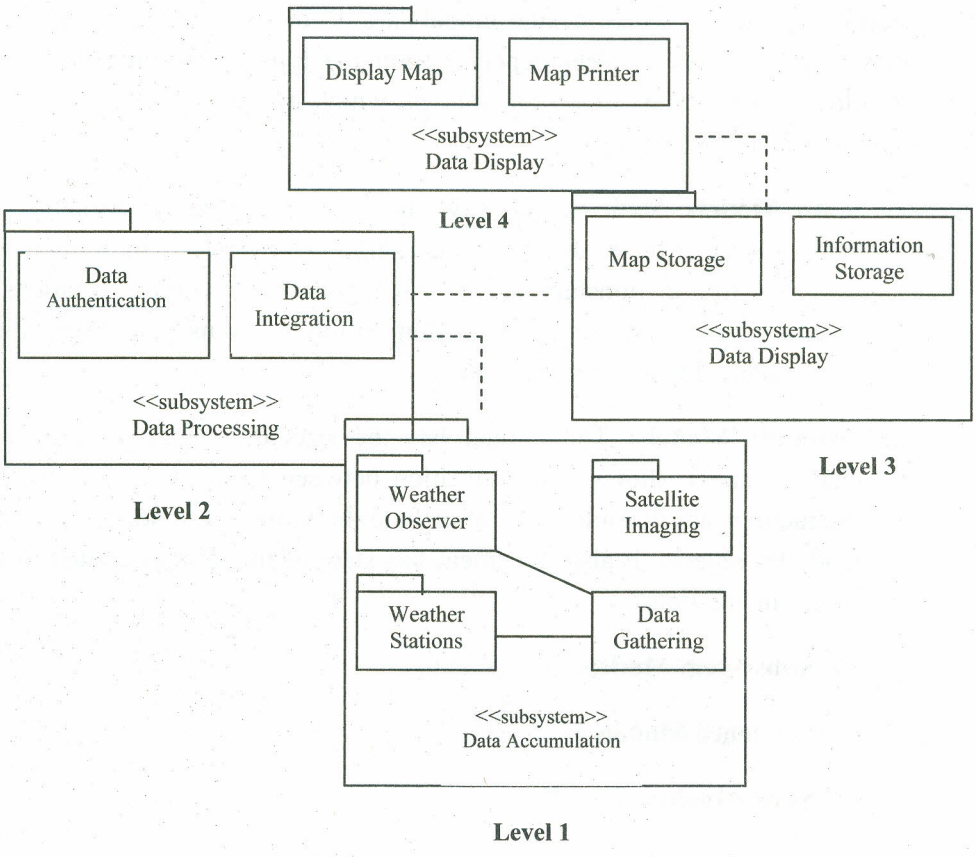


Fig. 10

The above figure describes all the essential subsystems of various systems corresponding to weather mapping system. The details of weather mapping system can be claimed form the box given above.

1.11.1 Design Models of Object-Oriented Design Process

Design models are those models which display the objects or object classes in a system. It also shows the relationship between the entities in a system. Design models are nothing but design and it forms a bridge between the requirements of the system and the system implementation.

Conflicting requirements takes place in this model and to avoid that abstract is produce which explore the relationships between them and the system requirements. Along with the abstract, details also help the programmers to make implementation decisions.

This conflict is originated by developing models at different levels of detail. There are three close roles which has link between them i.e., the requirements engineers, designers and programmers. As the system is implemented specific design decisions may be made.

If the role models are not close or the link between them is indirect them more detailed models may be required. The design process has the most important step i.e., to decide which design models are needed and the level of detail of these models. This majorly depends upon the type of system that is being developed. Two types of design models which are produced to describe an object-oriented design are:

1. **Static Models:** Static models have the structure of the system which does not change and these are described using object classes. Static model also describes the relationships at this stage such as generations relationships, uses/used by relationships and composition relationships. These all are important and hence documented.

2. **Dynamic Models:** This model has the structure of the system which changes and it shows communication between the system objects. These interactions are documented and include the sequence of sequence requests made by objects. It also document has state of the system related to these object interactions.

   (a) Subsystem Models

   (b) Sequence Models

   (c) State Machine Model

a) Subsystem Models: These models consists of association of various objects into an appropriate subsystem entity, where each of these subsystem entities are rendered using a package icon i.e. " "(It is an UML notation). The subsystem models come under UML supported static model.

For example: Consider the following figure.

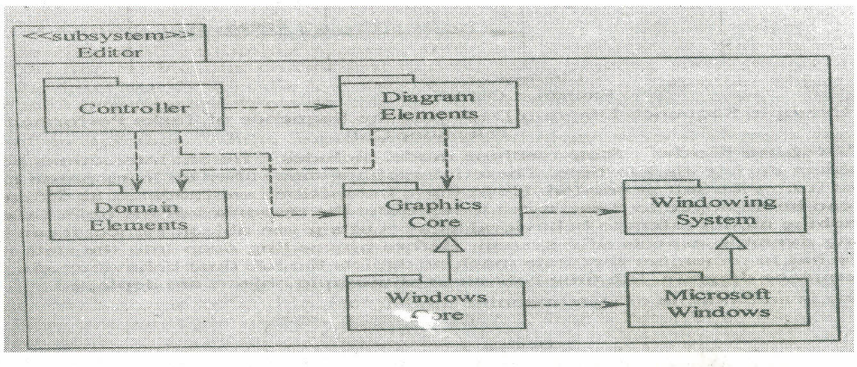


Fig. 11: Example figure Depicting the Subsystem Modeling

b) Sequence Models: Sequence models lay stress on time ordering of messages. These models come under the category of dynamic models supported by UML. The current discussion of sequence models utilizes the services of sequence and collaboration diagrams of UML. Sequence models are usually, initiated by maintaining all the participating objects (objects which are part of the diagram) at the top of the diagram in an horizontal fashion as usual, the objects which begin or initiates the interaction are rendered first in the diagram followed by the other participating objects. This sequence of placements helps in maintaining the clarity of the model. Each object is differentiated from other objects using a suitable horizontal line which represents the life time of the specific object. Finally, as it is a form of interaction diagram, hence, facilities are provided to represent transmission of messages between these objects, which are rendered on different kinds of relationships. Following is a sequence diagram depicting the sequence of interactions during a telephone call.

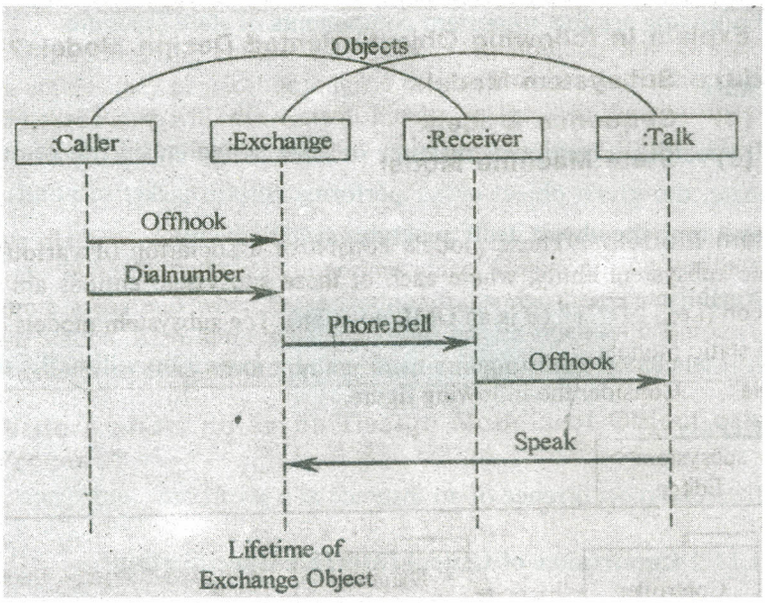


Fig. 12: Example Sequence Diagram Depicting the Sequence of Tasks Performed during a Telephone Call

c) State Machine Model: State machine model includes different interactions performed by a given object during its life time, these interactions can either be in response to the external events or it may be self generated. Here, these interactions are rendered as different states and arrows can be deployed to these states to represent the responses of the object at each state. As state machine model reefers to behavioral aspects of a given object. Hence, it can be claimed in modeling dynamic aspects of a system. Before proceeding deep into the state machine concepts one has to remember that state machine depicts the life time behavior of single object, but in sequence diagram, life time behavior of multiple objects are deployed.,

Following is an example of state machine model.

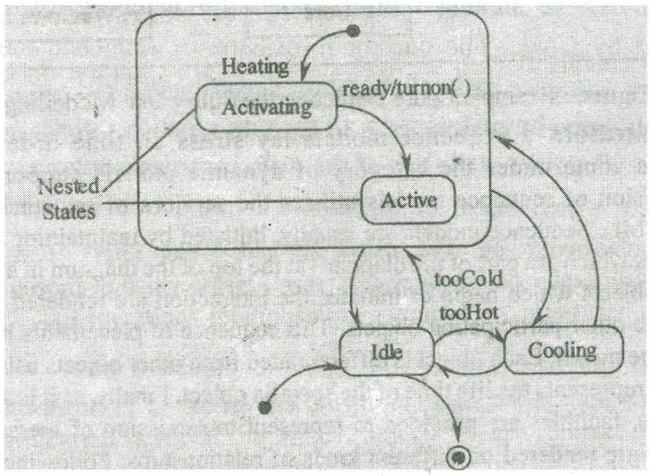


Fig. 13: Example diagram Depicting the Modeling of State Machine

Following are few important components of state machine:

State

It specifies a particular instance or state in the entire life of an object. IN this state, the given object either performs certain tasks, responds to any external events or the object itself generates certain events.

Activity

It refers to certain process which is under execution within the diagram.

Events

It refers to occurrence of an instance at a given time and space.

1.11.2 Importance of Object Interface Specification

While dealing with designing process, interface specification plays a major role. It is important in developing various constituent entities (say, object, subsystems) simultaneously. Hence, once the interface is specified, it can be implemented in the other object development. While dealing with inter face specification, once has to remember that the interface should not include all the details, rather it should be blocked from the objects which are deployed to obtain the specification of an interface. While, implementing this strategy, several changes can be applied to a given interface, without affecting the objects dealing the various specifications of this interface. Also it results in enhancement of maintainability of the design obtained. On the other hand, reverse of this strategy should be applied while dealing with static design model (i.e.,) in static design model displaying the essential characteristics fetches the optimum way to featuring the objects.

Whenever we discuss an object and interfaces, it is not true that each object should have a single interface or vice versa, rather, we may have design consisting of single object implementing multiple interfaces. Moreover, if we resort to object oriented program terminology (say Java) with respect to object and interfaces, it refers to the declaration of interface as a separate entity, with single or multiple objects implementing them. In this regard we can also say that, if there are multiple objects implementing a single interface, then using this single interface we can easily access all these objects. Here, the details of interface are with single or multiple objects. In designing of an interface object system, UML, uses the same diagrammatic notation to an interface as that of class (i.e.,). Rectangle with two partitions, with first partition including the interface name along with "<<interface>>" so, as to differentiate it with other classes. Also in next partition, operations are specified. While dealing with representation of interfaces, one has to remember that, it does not include any attributes section, as that of class.

For example: Diagrammatic representation "item Listener" interface is given below.

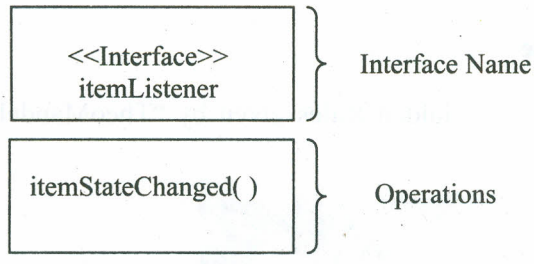


Fig. 14: Icon Used in UML to Represent Interface

The above specification of interface was using UML; now consider the specification of interface using programming language say Java. Following is a simple code representations interface in Java Programming Language.

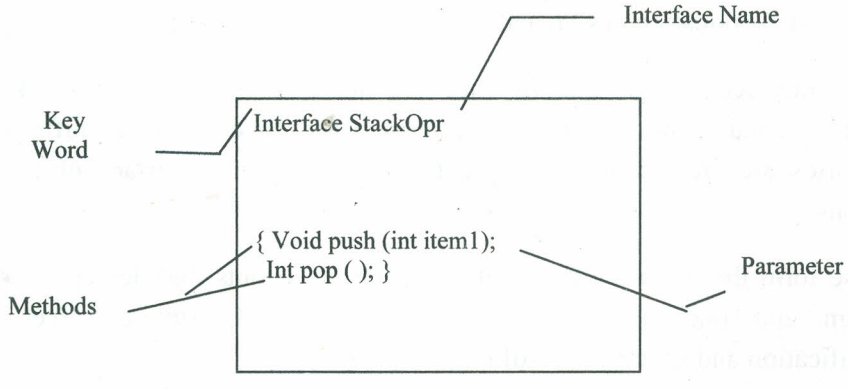


Fig. 15: Code to Represent in Interface

It is often fruitful to program an interface since, it is the compiler which brings up all the errors before hand and these can be easily rectified. No, matter even though the complexity of interface increases.

1.12 USER INTERFACE DESIGN

As the name itself suggests user interface design provides an interface between the user and the computer. Interface design revolves around three areas of concern i.e.,

1. The design of the interfaces between software components.
2. The design of the interface between the software and, producers and consumers of information which are non-human, and
3. The design of the interface between the user and the system.

Earlier the user use to face many interface problems but thanks, to the new technology like GUI, Windows, Icons, etc., which have eliminate extreme interface problems. But even after the new developments it is difficult for a user to have interface with the computer as they are difficult to use, hard to learn, confusing and so on. There as many such questions among which few are answered as part of user interface design.

The Golden Rule

There are three main Golden Rules given by "TheoMandel" which are to be followed.

They are:

1. Place the user in control,
2. Reduce the user's memory load, and
3. Make the interface consistent.

These rules are nothing but the principles for a set of user interface design and they guide application design action.

User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions.

These form the basis for the creation of screen layouts that depicts graphical design and placement of icon, definition of descriptive screen text, specification and specification of major menu items.

"Place the User in Control"

The above statement refers to one of three golden rules supplied by The Mandel in order to achieve good interface design. A deep illustration on above mentioned statement is given below.

People who deal with the first phase of software development (i.e.,) information gathering phase to prepare the customer requirement specification, often get astonished with certain peculiar requirement of the user.

For example: If windows based graphical interface is a resultant system to be developed and if say certain customer specifies that the requires the system (to be developed) to be well formed such that, the software implementing the system should be capable enough to judge the user requirement even before he attempts for it.

Though this statement looks quite contradicting but by analyzing it deeply we may come out with many conclusions like:

— Every user intends comfort while he dealing with any of the graphical interface.

— Moreover, the extent of comfort should be of high value that it should completely satisfy the user as well as the environment, the system provides should be understandable by normal human mentality (i.e.,) the system should not resort any complex interfaces creating an environment hard to be understood.

Hence, with above mentioned specifications, it can be realized that various routines, procedures or certain limitations implemented by designer of a system are meant to favour the user. But these specifications some times are implemented by the designers to make the implementation easy but at the same time they turnout to be complex for the normal users operating the system. Hence, by considering these consequences few principle have been deduced favoring the users in order to maintain control.

These are as follows:

Principle – 1

"Allow user interaction to be interruptible and undoable"

Explanation: To analyze above mentioned principle, consider a situation where a user is working with multiple tasks. Hence, the system should be so compatible to allow this user to switch between these tasks with the cost of not losing the data. Also, the user should be capable enough to terminate or undo the executing tasks etc., switching between different tasks is referred as the process of interrupting the current task and moving ahead with nest task.

Principle – 2

"Hide technical internals from the casual user".

Explanation: In this principle, stress is laid to keep away the user from the technical details of the system. The user should view only the outputs of the

program, he should not be made aware of the implementation details of the program (or) the operations of various entities of the system, which are running collaborate to display the output.

Principle – 3

"Design for direct interaction with objects that appear on the screen".

Explanation: The interface should allow meaningful display of objects, such that they should resemble the physical real world entities. Hence, the user, just by viewing once should be able to identify the purpose of a given object.

For example: we often an icon displayed by the operating system, to refer to an object which is responsible to control volume of the system.

Principle -4

"Provide for flexible interaction".

Explanation: The above principle specifies, that the user interface should agree with the priority of interaction on the feasibility of the user.

For example: If a person is blind, then the system should supply feasibility by allowing him to supply commands by means of using keyboards or voice commands etc.

At the same time, it should be remembered that the provision of priority should not be an alternative to the mode of interaction which suits best for given application.

For example: It often turnout to be critical to use a keyboard to accomplish sophisticated diagrams using "Auto Cad" Software.

Principle -5

"Define Interaction modes in a way that does not force the user into unnecessary of undesired action".

Explanation: In an interface design, the designing of interaction mode is of prime focus. The above principle lays stress on this aspect, since the user gets frustrated if he/she remains too long or too short in it, hence, the user should remain in this mode for a moderate period of time.

For example: Assume that the user is using the paint software and he needs to draw a square on the monitor. This he can do it easily by selecting the square from the list of shapes provided. The list of shapes should get deactivated as soon as he selects and pastes the appropriate item on the monitor so that he can switch on to another mode easily, without worrying much of shape mode.

Principle – 6

"Streamline interaction as skills levels advance and allow the interaction to be customized".

Explanation: The given principle favours to improve the design of the interface, where the user intends to use a given series of action multiple times. In this regard the concept of say, "Macros" can be used effectively.

"Reduce the User's Memory Load"

It is estimated that the best user interface design is that which makes the user to apply very little of his intelligence. Hence, the above mentioned principle is referred as one of the golden rules suggested by Mr. TheoMandel in favour of designing the user interface design. According to him the interface should be so interactive that, it should be capable of storing certain valuable information which is necessary and sufficient enough to make the user to recall the procedures of interaction, thus allowing the user to apply very little of his mental ability. Following are the certain principle suggested by Mr. Mandel in this aspect.

Principle - 1

"Disclose information in a progressive fashion".

Explanation: This principle favours disclosing of information, belonging to a given instance of an interface in a step by step manner. This means that, large volume of information related to any instance of an interface should not be directly provided to the user, rather, it should be initiated with a less volume of information in first step, followed by increasing it next higher step.

Principle – 2

"Define Shortcuts that are intuitive".

Explanation: While designing a user interface short cuts should be preferred to wider extent. This makes interaction very easy and in less time, since the user need not worry of entire procedure, rather, just by remembering certain shortcut values, he can make the work to be done.

For example: just by remembering shortcuts, "ALT+F+S", he can store a given file easily of halting the given session by clicking the escape button, using the mouse, clicking the file drop down list and then selecting the "store" value which is cumbersome and time consuming process.

Principle – 3

"Establish meaningful defaults".

Explanation: This is a direct principle, which specifies a default value to be included in every application of user interface. This is an important aspect, since a given interface can be used by large number of users (normal to professional). But means should be provided to select from a set of values, which are alternative to defaults and an "reset" option should also be included which can switch to a default value after remaining on the other alternative values.

Principle – 4

"Reduce demand on short-term memory".

Explanation: The demand for short term memory rises only when a given user is dealing with certain complex computations. If the interface includes certain mechanisms to remember the previous interactions, this itself will take away certain amount of memory. Hence, to check that the memory remains available, then, instead of storing the previous interactions, this task can be accomplished just by displaying certain "Visual Cues" which itself can be a remainder to the user which he himself can recall his past interactions.

Principle – 5

"The visual layout of the interface should be based on a real world metaphor".

Explanation: The above principle lays stress on the layout structure of the interface. It says that the layout of an interface should resemble the real world system terminology, referring which the interface has been built.

For example: If the interface represents the restaurant cash system, then it should include the layout structure, as which is frequently observed at normal restaurant's cash counter (i.e.,) the systems should include fields like total amount, amount paid, amount returned etc.,

"Make the Interface Consistent":

This is one of the golden rules suggested by Mr. Mandel to achieve best user interface design.

Following are the principles suggested by Mr. Mandel in favour of making the interface design consistent.

Principle – 1

"Maintain consistency across family applications".

Explanation: Whenever we deal with a specific kind of task, the given set of applications (associated with that task), should be implemented by following

certain prescribed rules. These rules should be equally adhered to all these applications. This is important so as to maintain consistency.

Principle – 2

"Allow the user to put the current task into a meaningful context".

Explanation: In general, a given interface remains the outcome of combination of hundreds of interactive modules. Here each module or layer provides interactive entities (thorough which interaction) such as buttons, checkboxes, images etc., It is prescribed that, selection of interactive entity to be made depending on the context. When switching from one module to other, a path should be provide, so that it remains an indicator to determine the location of the user (as users can get deep into several modules or layers, hence, indicators remain effective in getting deep into or coming out of these layers).

Principle – 3

"If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so".

Explanation: In this principle, producing the interface in favour of user's interest is of prime focus. In general, if a given user gets habituated using several key's for a definite task such as ALT+F+S to save current file, it is prescribed that, such common usage should not be altered to some other keys. Sometimes these alterations can make users frustrating.

1.13 INTERFACE DESIGN STEPS

End of Interface analysis marks the beginning of interface design issues. Interface design is a step by step process in which each step may get repeated multiple times and the new step generated contains upgraded qualities than previous steps.

Following are few important steps effective in developing the interface design:

Step-1: Initially consider the useful information obtained from the interface analysis phase. Extract interface objects from it and define their (objects) operations.

Step-2: Model the events that may cause the change of state of the user interface.

Step-3: Define the interface states.

Step-4: Consider the information obtained from the interface. Using this information mark the way the user interprets the state of information.

Irrespective of above mentioned steps, a given interface designer may initiate the process of designing by neatly sketching the required interface states and later defining the essential objects and its related information (i.e., attributes, and operations).

It is always prescribed to the designer to traverse through the golden rules, produce models depicting the implementation of the interfaces and lastly define the platform that will be taken into consideration.

Implementation of Interface Design Steps

Following are the steps essential while implementing the interface design:

- Initially identify the object and its associated operations. This is usually done by traversing through the use case written. Later these are sufficiently elaborated and refined.
- Now, categorize these objects, under the following types i.e., source, application and target objects respectively.
- Transfer the source objects onto the target objects, hence, in this way a hard copy of the application is created
- When all the possible objects are identified and their relative operations are defined, then start creating the screen layout.
- To make the layout attractive, several text, graphic images, icon etc., we placed at suitable locations. To make the layout to look livelier, any real world entity which is satisfying the situation (while making the layout) can be added.

Development User Interface Design Patterns

In general the design pattern refers to an abstract representation, which can be treated as a solution to a well formed problem. As there are many highly developed graphical user interfaces available today and hence, any of them can be utilized effectively to obtain a wide variety of design pattern.

Various Design issues associated with the User Interface Design

Generally, there are four major design issues which should be addressed at the beginning of the design process so as to lower the constraints (such as delays in the project etc.,) which may dissatisfy the customer. Details of these issues are illustrated below:

Error Handling

While operating computer, we often encounter certain error messages displaying an unknown information i.e., the messages of the form "My Computer not responding due to error at 1736 and requires to terminate immediately. Such messages often frustrate the users since, it neither carries any valuable information nor it describes any measures to escape such errors". Hence, while ;designing user interface, following are few effective guidelines to be followed, for making error handling mechanism easy:

1. The messages should be understandable i.e., it should be displayed in the language preferred by the user.
2. The messages should carry valuable information to escape errors.
3. Actual cause of the occurrence of error should also be included in the error messages.
4. While displaying an error message, certain graphics should be associated with message i.e., a strong beep sound or flashing or color changing periodically et.,
5. Finally, the information embedded on the message should not comment or blame the operators for their erotic operation.

Inclusion of help Topics or Options

Almost every software includes help topics which may guide the users for the proper usage of the software. Hence, while including help topics following are few essential facts to be considered effectively.

1. In order to provide query, sufficient space such as text field or certain options should be provided.
2. While responding to a given query a separate window should be displayed narrating the suggestions to the operator's query or should specify the exact path to obtain the solution or should narrate the ways through which the operation can gain valuable information (often diagrammatic explanation is useful).
3. Once the operator is satisfied, there should be sufficient way through which the user may resume to normal operations. These way can include a return button, a function key etc.,
4. While displaying a window, the data presented on it should follow a flat layout. The text should include certain important key or can contain a hierarchical tree structure or can include certain hyperlinks which may take the user from one document to other.

5. The help a topic or options should be included to only few functions. But help data should contain information to all the application specific to a given function.

Inclusion of Menu and Command Labeling

The inclusion of menus and command labeling should be such that, it must satisfy the following questions:

1. Is there availability of a command with almost every menu?
2. What will be the form of representation of command?
3. Do all the menus represent the purpose of their inclusion?
4. Do the submenus (belonging to a concrete menu) satisfy the condition for which it is referred by an item of its concrete menu?
5. Are there any facilities with which a given operator can easily abbreviate or create short forms for the commands?
6. Is there any provision for the operator if he/she forgets the command? Is the command complex to be remembered?

The System Response Time

It is nothing but the time taken by the system to respond to a user activity. It is usually measured from time of generation of an event by the user till the response made by the system. The system response time has two important characteristics i.e., variability and length.

Length refers to the time between the event generated and the response made by the system. Variability is the deviation in the available average response time. As the variability gets decreased, it enriches the user during system. As its negative aspect, if the variability increases it irritates the user and he blames himself, thinking that some error might have crept during his operation.

Check Your Progress 3

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) What are the interface Design elements?

.....

.....

.....

.....

2) _____provides an interface between the user and the computer.

.....

.....

3) What are the steps involved in developing an interface Design?

.....

.....

.....

.....

4) _____have the structure of the system which does not change and these are described using object classes.

.....

.....

5) _____ has the structure of the system which changes and it shows communication between the system objects.

.....

.....

1.14 LET US SUM UP

Design is a process of translating analysis model to design models that are further refined to produce detailed design models. The process of refinement is the process of elaboration to provide necessary details to the programmer. Data design deals with data structure's selection and design. Modularity of program increases maintainability and encourages parallel development. The aim of good modular design is to produce highly cohesive and loosely coupled modules. Independence among modules is central to modularity. Good user interface design helps software to interact effectively to external environment. Tips for good interface design helps designer to achieve effective user interface.

Quality is built into software during the design of software. The final word is: Design process should be given due weightage before rushing for coding.

1.15 CHECK YOUR PROGRESS: THE KEY

Check Your Progress 1

1) Framework

2) Design process

3) Abstraction

Check Your Progress 2

1) Partitioning , abstraction

2) Step-wise refinement involves the following activities:

   a) Decomposing design decisions to elementary levels.

   b) Isolating design aspects that is not truly interdependent.

   c) Postponing decisions concerning relative to representation details as long as possible.

   d) Carefully demonstrating that each successive step in the refinement process is a faithful expansion of previous steps.

3) High, Low

4) Software architecture

Check Your Progress 3

1) Basically these are three types of interface design elements:

   a) The User Interface b) External Interface c) Internal Interface

2) Interface design

3) Following are few important steps effective in developing the interface design:

   Step – 1 Initially consider the useful information obtained from the interface analysis phase. Extract interface objects from it and define their (objects) operations.

   Step – 2 Model the events that may cause the change of state of the user interface.

   Step – 3 Define the interface states.

   Step -4 Consider the information obtained from the interface. Using this information mark the way the user interprets the state of information.

4) Static models

5) Dynamic models

1.16 SUGGESTED READINGS

- http://sdg.csail.mit.edu
- http://www.ieee.org
- http://www.rspa.com
- Jindal, Gaurav and Bakshi, Arun (2010). *Object Oriented Software Engineering,* Haranand Publications, Delhi..
- Braude, Eric J. Software Design : From *Programming to Architecture,* Wiley
- Pressman, Roger S. *Software Engineering-A Practitioner's Approach,* McGraw-Hill International Edition
- Sommerville, Ian. *Software Engineering*, Pearson Education

UNIT 2 APPLICATION CODING

Structure

2.0 INTRODUCTION

The best applications are coded properly. A coding standards document tells developers how they must write their code. Everybody wants to create maintainable code. If the standards are followed, then the source code will be more comprehensive and will become easy-to-maintain. Code standards implements traceability. Repeated performance pitfalls could be avoided. Choosing the appropriate architecture for your application is key. Most initial attacks against your systems are likely to focus upon the systems infrastructure and commercial applications. Four attack categories are most prevalent. **Buffer overflow attacks**, **Race conditions, Exploitation of application component privileges**, **Client-side manipulation**. In the case of web applications, the security threats to compiled applications are very similar and often share the same principles of data integrity and resilience. Some new methods are to be implemented for securing applications to minimize the developer's time for correcting potential flaws that may be discovered during a security assessment

and also to maximize security and data integrity during the process of building a new application.

2.1 OBJECTIVES

After going through this unit you will be able to:

- describe why you need coding standards;
- list the advantages of coding standards;
- list the good methods for coding;
- describe the methods for effective source code control;
- list out the rules for developing secured code;
- describe security principles;
- list out the custom applications and their security threats, and
- explain general advice on sending custom applications.

2.2 WHY WE NEED CODING STANDARDS?

The best applications are coded properly. This sounds like an obvious statement, but by 'properly', I mean that the code not only does its job well, but is also easy to add to, maintain and debug.

This "maintainable code" is a popular talking point among those involved with PHP and probably with other languages as well. There's nothing worse than inheriting an application or needing to make changes to code that requires a lot of energy to decipher – you end up trawling through lines and lines of code that doesn't make its purpose or intentions clear. Looking through unfamiliar code is much easier if it is laid out well and everything is neatly commented with details that explain any complicated constructs and the reasoning behind them.

2.2.1 The Problem

When we learn a new language, we usually begin to code in a specific style. In most cases, we'll write in a style that we want, not one that has been suggested to us. But once we start to code using a particular style, like a spoken language dialect, it will become second nature -- we'll use that style in everything we create. Such a style might include the conventions we use to name variables and functions ($userName, $username or $user_name for example), and how we comment our work. Any style should ensure that we can read our code easily.

However, what happens when we start to code bigger projects and introduce additional people to help create a large application? Conflicts in the way you write your code will most definitely appear.

2.2.2 The Solution: A Coding Standards Document

A coding standards document tells developers how they must write their code. Instead of each developer coding in their own preferred style, they will write all code to the standards outlined in the document. This makes sure that a large project is coded in a consistent style -- parts are not written differently by different programmers. Not only does this solution make the code easier to understand, it also ensures that any developer who looks at the code will know what to expect throughout the entire application.

Coding standards are great -- but how do you decide which standards you want to apply, and how they will be defined? When you formulate your ideal coding style, you should think about these points:

1. Can you actually read the code? Is it spaced out clearly?
    - Do you separate blocks of code into 'paragraphs' so that different sections are easily defined?
    - Are you using indentation to show where control structures (if, else, while and other loops) begin and end, and where the code within them is?
    - Are your variable naming conventions consistent throughout the code and do they briefly describe that data that they'll contain?
    - Are functions named in accordance with what they do?
2. If you come back to the code in a few weeks or months, will you be able to work out what's happening without needing to look at every line?
3. How are you commenting the work?
4. Have you used complex language functions/constructs that are quicker to write but affect readability?

Once you've considered those points, you can begin to draft your coding standards. Consult with your team members (if any) and compare how they code to your own style -- you shouldn't force total change upon everyone. Compromise and incorporate elements of everyone's style. If someone has been coding in a specific way for a long time, it will take a while for that developer to change to the new method. Developers will likely adopt the style gradually, just as an accent develops over time.

2.2.3 Create Maintainable Code

As we said at the beginning of this discussion, you want to create maintainable code. If you stop developing a project, return to it several months later, or hand

development over to someone else, you (and other developers) want to be able to understand what's going on in the code. We keep mentioning 'readability', but what actually constitutes "readable code"? The answer to this question will obviously differ for each programmer, but I believe there are some common fundamentals, which we'll discuss now. I've used PHP to outline various styles here, but similar ideas will apply to other languages:

```
// Method 1
if (condition)
function1($a, $b, $c);
for ($i < 7 && $j > 8 || $k == 4)
a($i);

// Method 2
if (condition)
{get_user($id, $username, $key); }

for (($i < 7) && (($j < 8) || ($k == 4)))
{display_graph($value); }
```

Here are two ways we might write the same code -- you can see the difference between easily readable code and complex, but more quickly written code. Even if you don't know much PHP, you probably noted that the second snippet is much tidier and easier to understand. Here, the functions are get (), display () used variable names that describe the data they contain, and used brackets to help show what the for condition is.

Your own idea of readable code might differ slightly from this, but every PHP coder could easily understand what's going on in the second example. The first example, however, takes extra time to comprehend. Sacrificing extra lines and whitespace will make a noticeable difference to the layout of the code. As you develop coding standards, try to ensure that they allow anyone to work out the code in future!

2.2.4 Different Standards

PHP example code and Pear modules (etc.) generally use the Pear Coding Standards, which are slightly different from the other methods. Here is an example for one type of coding standards.

```
if (condition)
{get_user($id, $username, $key); }
```

But in the Pear standards, the first brace is on the same line as the if (condition):

```
if (condition) {get_user($id, $username, $key); }
```

The standards you choose are all down to personal preference and you may find easiest to code and read.

2.3 ADVANTAGES OF CODING STANDARDS

Having coding standards in a software development organization has the following advantages.

For the Developers

1. The source code will be more comprehensive and will become easy-to-maintain. As the programmers became more and more familiar with the coding style as they implements the coding standards on project after project.

2. The uniform approach for solving problems will be handy because the code standards documents reveal the recommended methods that were tried and tested on the earlier projects.

3. Less communication between developers and managers will be needed because the programmers will not asked anymore on the details of the specification document because the defaults are all stated in coding standards.

4. Is common to the less experience programmer to re-invent the wheel. When there are coding standards, there is a big chance that particular problem is not really a new problem, but in fact, a solution may be documented before.

For the Quality Assurance Team

5. Well documented coding standards will aid the creation of "Test Scripts". Having reviewed the source code and tested an application based on compliance to coding standards, it added strong direction to ensure quality of the software product.

6. Because code standards implements traceability, the item ids can be used to describe a violation in the "Test Results" document that both developers and testers are familiar with.

For the Project Managers

7. It is important for the project managers to maintain and secure source code quality on their projects. Implementing coding standards could jumpstart this goal halfway to its realization.

8. Repeated performance pitfalls could be avoided. It is a common case that a released software product could be less impressive when it comes to performance when the real data has been loaded in the new developed database application.

9. Lesser man-hour consumption as the sum of all efforts implementing coding standards.
10. It is also beneficial for the organization who are applying for ISO 9001 license because coding standards is a complement from organization's execution plan requirements.

Check Your Progress 1

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) _____ need not reinvent the wheel if conding standards are available.

.....

.....

2) For _____ , Well documented coding standards will aid the creation of "Test Scripts"

.....

.....

3) What are the advantages of coding standards for an organization?

.....

.....

.....

.....

2.4 STRUCTURED CODING TECHNIQUES

Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Structured programming was first suggested by Corrado Bohm and Guiseppe Jacopini. The two mathematicians demonstrated that any that any program

logic, no matter how complex, can be expressed in terms of just three kinds of logical operations: sequence, selection and iteration. A program then is made up of a series of blocks of code to perform these operations. Let's define these three kinds of operations.

Sequence—a series of program statements are executed one after the other, in the order in which they appear in the source code. Obviously, this rules out statements like GOTO and IF, restricting us just to statements that perform some specific action. (A CALL or GOSUB to another procedure would qualify as an action in this sense.) Hence, a block of statements that are executed sequentially is called an "action block."

Syntax of Selection

Statement1;

Statement1;

Statement1;

Statement1;

...

```
# include<stdio.h>
# include<conio.h>
        void main()
{
        int a,b, sum;
        printf("Please, enter first integer value:");
        scanf("%d",&a);
        printf("Please, enter Second integer value:");
        scanf("%d",&b);
        sum=a+b;
        printf("The sum of two integer values is : %d",sum);
}
```

Selection—one set of statements, from a choice of two or more, is selected for sequential execution, based on some criterion. One way to accomplish this is to use an IF/THEN/ELSE construct. Some of the languages will also permit a SELECT/CASE/OTHERWISE/END - type structure, perhaps with different

but analogous keywords. The set of statements that gets executed in each case is itself an action block. Sometimes selection constructs are called "branch blocks."

Syntax of If else in C programming language

```
if (expression)
{
    Statement (s)
}
 else
{
   Statement (s)
}
```

Example: Program in C programming Language to check a number is even or odd

```
# include<stdio.h>
# include<conio.h>
void main()
{
    int num;
    printf("Please, enter an integer value:");
    scanf("%d",&num);
    if (num%2= = 0)
    {
        printf("The number is an even number :");
    }
    else
    {
       printf("The number is an odd number :");
    }
```

```
}//End of main
```

The general syntax of switch structure is C programming Language

```
switch (expression)
{
   case constant1: statement(s);
                   break;
   case constant2: statement(s);
                   break;
   case constant3: statement(s);
                   break;
   case constant4: statement(s);
                   break;
   case constant5: statement(s);
                   break;
   default:        statement(s);
                   break;
}
```

Example: Program in C programming language to check an alphabet is a vowel or not.

```
# include<stdio.h>
void main()
{
  char ch;
  printf("Please, enter a character →");
  scanf("%c",&ch)
switch (ch)
{
   case 'a': printf("Vowel");
```

```
            break;
    case 'A': printf("Vowel");
            break;
    case 'e': printf("Vowel");
            break;
    case 'E': printf("Vowel");
            break;
    case 'i': printf("Vowel");
            break;
    case 'I': printf("Vowel");
            break;
    case 'o': printf("Vowel");
            break;
    case 'O': printf("Vowel");
            break;
    case 'u': printf("Vowel");
            break;
    case 'U': printf("Vowel");
            break;
    default:
            printf("Consonant");
            break;
}
```

Iteration—a series of statements is executed repeatedly until some termination condition is met. These are also called "loop blocks." Virtually all languages contain simple FOR or WHILE or DO WHILE -type loops. More modern languages include variations such as DO UNTIL/END and DO WHILE/END.

These three kinds of "control blocks" have some features in common. First, the code in each is executed from top to bottom, which is the same way that it

appears in the source file. This makes the program much easier to read and understand than does the convoluted branching you find in so many BASIC programs. Of course, in a selection block, not every statement is executed, and in a loop block they may be executed more than once, but they still are always executed from top to bottom.

In structured programming the use of the "Go To" statement is discouraged.

In addition, each control block has just one logical entry point: the first statement. And if they're well structured, they have just one logical exit point: the last statement. A complete program is written by assembling and nesting blocks of these three kinds to perform the required processing.

Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages. Unstructured programming must rely upon the discipline of the developer to avoid structural problems, and as a consequence may result in poorly organized programs. Most modern procedural languages include features that encourage structured programming. Object-oriented programming (OOP) can be thought of as a type of structured programming, uses structured programming techniques for program flow, and adds more structure for data to the model.

SYNTAX OF FOR LOOP IN C PROGRAMMING LANGUAGE

```
for (initialization, condition, updation/reinitialization)
{
  Statement(s);
}
```

Example: Program in C programming language to print first 10 natural numbers.

```
int i;
for(i=1; i<10;i++)
{
   printf("%d",i);
}
```

The output of the program is

1 2 3 4 5 6 7 8 9

SYNTAX OF WHILE LOOP IN C PROGRAMMING LANGUAGE

```
while(condition)
{
  Statement(s);
}
```

Example: Program in C programming language to print first 10 natural numbers.

```
int i;
i=1;
while(i<10)
{
  printf("%d",i);
  i++;
}
```

The output of the program is

1 2 3 4 5 6 7 8 9

SYNTAX OF DO WHILE LOOP IN C PROGRAMMING LANGUAGE

```
do
{
 statement(s).
} while(condition);
```

Example: Program in C programming language to print first 10 natural numbers.

```
int i;
i=1;
do
{
  printf("%d",i);
  i++;
```

```
} while(i<10);
```

The output of the program is

```
1 2 3 4 5 6 7 8 9
```

2.5 GOOD METHODS FOR CODING

Since the first programmable computers, software developers and managers have struggled with fundamental problems in executing a project from inception to deployment. After nearly 65 years, most of the same breakdowns still remain. In fact, the Standish group reports that over 80% of projects are unsuccessful either because they are over budget, late, missing function, or a combination. Moreover, 30% of software projects are so poorly executed that they are cancelled before completion. Software projects using modern technologies such as Java, J2EE, XML, Visual Studio and Web Services are no exception to this rule.

Development Process

It is important to choose the appropriate development lifecycle for a given project because all other activities are derived from this process. A couple examples of this are the Rational Unified Process (RUP) and the extreme Programming (XP) methods. Having a well defined process is usually better than having none at all, and in many cases it is less important what process is used than how well it is executed. The methodologies above are very common and a quick Web search will turn up all kinds of information regarding how to implement them.

Requirements

Gathering and agreeing on requirements is fundamental to a successful project. This does not necessarily imply that all requirements need to be fixed before any architecture, design, and coding are done, but it is important for the development team to understand what needs to be built. Quality requirements are broken up into two kinds: functional and non-functional. A good way to document functional requirements is using Use Cases. Note that Use Cases are used for non-OO projects. Non-functional requirements describe the performance and system characteristics of the application. It is important to gather them because they have a major impact on the application architecture, design, and performance.

Architecture

Choosing the appropriate architecture for your application is key. You have to know what you are building on before you can start a project. Check the architecture of the target. Read as much as you can about the ins and outs of the

platform and note any pitfalls before you start your code. It will go a long way to heading off any bugs that might be 'show stoppers' later on.

Design

Even with a good architecture, it is still possible to have a bad design. Many applications are either over-designed or under-designed. The two basic principles here are "Keep it Simple" and information hiding. For many projects, it is important to perform Object-Oriented Analysis and Design using UML. Code reuse is but one form of reuse and there are other kinds of reuse that can provide better productivity gains.

Code Building

Building the code is really just a small part of the total project effort even though it's what most people equate with the whole process since it's the most visible. Other pieces equally or even more important include what we have already gone over the above namely requirements, architecture, analysis, design, and testing. A best practice for building code involves daily builds and testing.

Peer Reviews

It is important to review other people's work. Experience has shown that problems are eliminated earlier this way and reviews are as effective or even more effective than testing. Any artifact from the development process is reviewed, including plans, requirements, architecture, design, code, and test cases. Peer reviews are helpful in trying to produce software quality at top speed.

Software Testing

Testing is an integral part of software development that needs to be planned. It is also important that testing is done proactively; meaning that test cases are planned before coding starts and test cases are developed while the application is being designed and coded.

Performance Testing

Testing is usually the last resort to catch application defects. It is labor intensive and usually only catches coding defects. Architecture and design defects may be missed. One method to catch some architectural defects is to simulate load testing on the application before it is deployed and to deal with performance issues before they become problems.

Configuration Management

Configuration management involves knowing the state of all artifacts that make up your system or project, managing the state of those artifacts, and releasing

distinct versions of a system. There is more to configuration management than just source control systems, such as Rational Clearcase.

Quality and Defects Management

It is important to establish quality priorities and release criteria for the project so that a plan is constructed to help the team achieve quality software. As the project is coded and tested, the defect arrival and fix rate can help measure the maturity of the code. It is important that a defect tracking system is used that is linked to the source control management system. By using defect tracking, it is possible to gauge when a project is ready to release.

Deployment

Deployment is the final stage of releasing an application for users.

System Operations and Support

Without the operations department, you cannot deploy and support a new application. The support area is a vital factor to respond and resolve user problems. To ease the flow of problems, the support problem database is hooked into the application defect tracking system.

Data Migration

Most applications are not brand new, but are enhancements or rewrites of existing applications. Data migration from the existing data sources is usually a major project by itself. It is as important as the new application. Usually the new application has better business rules and expects higher quality data. Improving the quality of data is a complex subject.

Project Management

Project management is key to a successful project. Given the number of other checklists and tip sheets for project management, it is surprising how many project managers are not aware of them and do not apply lessons learned from previous projects, such as: "if you fail to plan, you plan to fail." One way to manage a difficult project is through timeboxing.

Measuring Success

You can measure your development process against an industry standard known as the Capability Maturity Model (CMM) from the Software Engineering Institute at Carnegie Mellon University. Most projects are at level 1 (initial). If you implement the best practices described above and the guidelines, then you could be well on the way to achieving a higher maturity level and a successful project.

2.6 METHODS FOR EFFECTIVE SOURCE CODE CONTROL

Most developers have at least tried using a source code control system at one time or another. If you're working on a development project as part of a team, such a system is practically indispensable as a way to prevent team members from tripping on each other's changes. Even if you're working all by yourself, source code control can provide a valuable safety net, saving a separate copy of your code in a secure repository and providing you a way to undo major mistakes.

But just how effective is your source code control? Like other parts of the development world, there are ways to make source code control work harder on your behalf. Here are some tips on getting the most out of your source code control system.

Method #1: Choose the Right System

There are many source code control systems on the market. Rather than just picking the one you know about, or the one that comes for free with your IDE, it's worth taking the time to seriously evaluate the alternatives. You'll probably find some that fit in better with your development style than others. Here are a few factors to consider:

- **Price** - Some systems are free. Others will cost you hundreds of rupees per user (or even more).
- **Concurrent development style** - Some systems require you to actively check out code before working on it. Others let you change anything you like, and then merge the changes when you're ready to commit them.
- **Repository** - Does the system store data in a database, or in the file system? Are you comfortable with the safety and security of the repository?
- **Internet friendliness** - If your team is geographically distributed, make sure that the system you choose works well over TCP/IP, without using up excessive bandwidth.
- **IDE integration** - Some people really care whether the can perform source code control operations from within their IDE. Others don't.
- **Cross-platform support** - If you need to do development on more than one platform (say, Windows and Linux), you'll want a system that works on both platforms.

Method #2: Put the Right Things in the System

Smart teams use the source code control repository for more than just source code. The key is to store any artifact that's not easily rebuilt from other artifacts. "Artifact" is a general term for anything having to do with your software product. Here are a few things you might consider storing in your source code control system:

- Source code
- Windows Installer databases
- Database build scripts
- Icons, bitmaps, and other graphics
- License files
- Readme files
- Informal development notes
- Documentation
- Help files
- Build scripts
- Test scripts

Having everything in one place makes life much simpler when you're faced with the inevitable support request for an older version of your product. This is much easier than trying to collect source code from one place, database scripts from another, and bitmaps from a third - if anyone even bothered to save old database scripts.

Remember, too, that once you store something in the system, you must manage it through the system. Don't let developers (or anyone else) copy things off to a private sandbox with the intent of checking them in later. Once a file escapes from source code control, you'll lose the ability to easily re-create the state of your project at a point in time.

Method #3: Don't Hog the Files

The best developers make frequent small changes to the repository, rather than a few huge changes. In a check-in/edit/check-out system (like Visual SourceSafe) this means only checking out a few files at a time. In an edit/merge/continue system (like CVS) this means committing changes after each task, rather than waiting until the end of a day or even longer.

Minimizing the size of changes has several good effects. First, if you are working with a system that locks files, you can avoid locking other people out for longer than necessary. Second, by keeping your commits small, you vastly lower the chance of needing to merge two incompatible versions of code. Finally, by working in small chunks, you can keep the comments in the source code control system targeted and informative.

Method #4: Use Labels and Branches Wisely

Labels (sometimes called tags) give you a way to mark a versioned set of files with some friendly name. This is useful because humans are much better at remembering things like "Beta 2" than "Version 3019.4". You should apply a label to your source code control repository at any significant point in time. This includes:

- Public releases of the software
- The start and finish of major code changes
- The incorporation of a new third-party component
- Builds that pass basic QA testing

Effective use of labels makes it easy to jump back to a particular point in time, whether to undo a terrible mistake or to see what state the system was in at that time.

By contrast, branches give you a way to develop two sets of source code for the same project in parallel. You should create a branch whenever different developers in the same project are following different rules. For example, if one team is finishing up version 1, while another is starting on version 2, that's a good time for a branch. Presumably the first group of developers is checking in small, careful changes, while the other is roughing out the broad outlines of new features. Because the rules for these activities are different, they can't easily be performed in a single set of source code files.

Branching is useful for exploration as well. If your main line of code is using a known, stable technique, but you want to explore a faster and potentially unstable way of accomplishing the same objectives, you should create a branch to explore the consequences. If the branch works out well, merge its changes back to the main line. If not, just stop working in the branch.

Finally, don't create a branch just because you know you'll need it in the future. Wait to make the version 2.0 branch until someone is actually going to work on it, to avoid extra merging before it's really needed.

Source code control is one of those basic practices that every developer should use on every project. With the availability of several excellent free systems,

there's no excuse for working without the source code control safety net. After you get used to storing files in a source code control repository, take the time to learn what else your system can do for you. You might just be surprised at the difference that it makes to your development efforts.

2.7 RULES FOR DEVELOPING SECURED CODE

Rule #1: Take Responsibility

Rule #2: Never Trust Data

Rule #3: Model Threats against Your Code

Rule #4: Stay One Step Ahead

Rule #5: Fuzz!

Rule #6: Don't Write Insecure Code

Rule #7: Recognize the Strategic Asymmetry

Rule #8: Use the Best Tools You Can

Rule #1: Take Responsibility

Getting security right in your product is totally up to you. No one else—and certainly no magic tool or programming language—solves all security ills. I like source code analysis tools, but they will not magically fix all your security vulnerabilities. Only you can do that.

Secure products are built by developers who create secure designs and write secure code. Ultimately, writing code is an individual endeavor. You are an individual, and you cannot be replaced by a tool. Therefore, the security of your product is your responsibility! The Blaster and CodeRed worms exploited code that was written by individuals (see **Fig..1**).

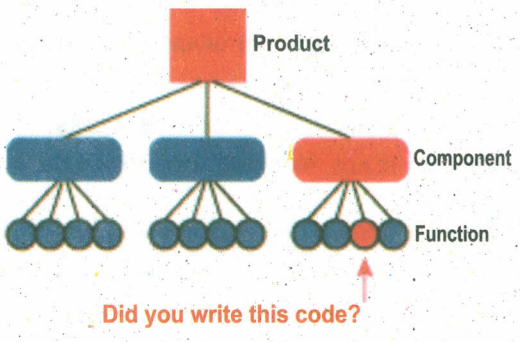


Fig.1: Vulnerable Code is written by Individuals

Remember that all code will be scrutinized and possibly attacked. That's totally okay. Being attacked is fine. The big question is, will *your* code be compromised? Only you can determine that outcome. So take pride in your code. You must be happy with the quality of your code and be able to sleep at night knowing that if it's attacked, you've done everything possible to prevent the code from being exhausted.

If at all possible, have your code peer-reviewed by a security expert. Don't have it examined by someone who knows nothing about security and expect that person to find security bugs and vulnerabilities. Go out of your way to have someone who really knows what he is doing look at the code.

And don't be so bigheaded that you can't ask for help when you need it. You shouldn't solely rely on tools, but you should definitely take advantage of any that are readily available to you. Do run all available source code analysis tools on your code, and run them often. Take advantage of every possible defensive language construct and library trick available to you. For example in C#, wrap network-facing code that performs array access, where the array index is derived from a network request, in checked operators to detect possible integer arithmetic errors.

Rule #2: Never Trust Data

All input is evil until proven otherwise. If you look at the most heinous security vulnerabilities, they all share the common trait that the developer trusted the incoming data. The problem is if your code assumes the data is well formed, what happens if your assumption is incorrect? On a good day, your application will probably crash. On a bad day the attacker could inject malicious code into your process and wreak havoc.

The somewhat whimsical definition of a secure system is one that performs the tasks it is supposed to, and no more. But when there are input trust issues, you can usually get the system to perform other tasks. The most well-known issues are buffer overruns, integer arithmetic bugs, cross-site scripting, and SQL injection bugs. We are starting to see new variations on this theme, such as XPath injection and Lightweight Directory Access Protocol (LDAP) injection vulnerabilities.

You can remedy input trust issues by following a few simple rules. First, don't look only for things you know are bad; this assumes you know all the bad things and you can predict all the bad things into the future. Looking for bad things is OK so long as it's not your only defense. A better strategy is to constrain the input to what you know is good. For high-level languages such as C# and Perl, I like to use regular expressions to achieve this.

Next, reject what you know to be bad. For example, if someone remotely requests a file through your code and the file name includes a dodgy character

(such as : or \), reject the request. And don't tell the attacker why; just say "file not found."

Finally, and this does not work for all scenarios, sanitize the data. For example, in the case of a Web server, you should HTML-encode output that came from potentially untrusted input.

Rule #3: Model Threats against Your Code

You do have threat models, right? Threat models allow you to understand the potential risks to your software and to make sure you have the appropriate mitigations in place. But the benefits of threat modeling extend beyond secure design. Threat models can help with your code quality, too. Threat models tell you where the data came from. Is the data remote or local? Is the data from anonymous users or is it from more trusted (authenticated) users, perhaps administrators?

With this information at hand, you can determine whether your defenses are appropriate. For example, code accessible to anonymous and remote users had better be very secure code. I'm not saying that code accessible only to local admins should not be secure, but I am saying that remotely accessible code, especially code running by default, has to be bulletproof and that means more defenses, more review, and more attention to detail. Moreover, the threat model can tell you the nature of the data being protected. High value business data and personally identifiable information, for example, must be protected very well. Are your defenses appropriate?

Make sure your threat models are accurate and up-to-date, then identify all the entry points into your code and rank them by accessibility—remote versus local and high-privilege versus low-privilege (or no-privilege) users. The most accessible code should be reviewed the deepest and the earliest. Finally, review all code along anonymous data paths; in other words, start at each anonymously accessibly entry point and trace the data along that path, checking for code correctness.

Rule #4: Stay One Step Ahead

The security landscape evolves constantly. It seems that every week there are new variations of security issues. This means you must evolve and learn about new threats and defenses or you'll suffer the consequences.

Some simple strategies to stay ahead of the curve include reading a few good books on the subject of software security every now and then. Also, learn from your past mistakes and, better still, the mistakes of others.

Rule #5: Fuzz!

Fuzzing is a testing technique that was invented to find reliability bugs. It turns out that a percentage of reliability bugs are security vulnerabilities waiting for the right exploit! Sure, a buffer overrun might crash an application, but given a well-crafted malicious payload, the crash might not happen, and the attacker could run code to do his bidding instead. Our motto around here is "today's denial of service is tomorrow's code execution."

Just about every file-parsing bug/vulnerability was found by dumb luck or fuzzing. Microsoft has found security vulnerabilities parsing a number of file formats including the XLS, PPT, DOC, and BMP files. Most vendors have had similar vulnerabilities because parsing complex data structures is a complex task, complex code will have bugs, and some of those bugs will reveal security vulnerabilities.

You must fuzz all code that parses files and network traffic. The Security Development Lifecycle (SDL) at Microsoft is very specific about what this means for file formats. You must fuzz all parsers with 100,000 iterations of malformed files using a file fuzzer. One last note about fuzzing. If you get a crash, don't think it is only a crash. It is likely that a good percentage of these so-called crashes are begging for someone to write an exploit. So don't simply punt a crash as "just a crash."

Rule #6: Don't Write Insecure Code

At Microsoft, they use the concept of quality gates to help reduce the chance a developer will check vulnerable code into the product. The gates run a battery of source code analysis tools on the code prior to check-in to flag any issues. And any identified issues must be fixed before the check-in can be completed. You can also enforce strict code rules such as excluding the use of banned functionality, like no calls to strcpy or strncat and no lousy crypto. For example, with regard to cryptography, don't allow DES (the key length is too small), MD4, or MD5 (they are both broken now) in new code, unless an industry standard dictates their use.

Don't reinvent functionality. If you have code that parses a specific file format, you don't need two or three sets of parsing code; stick with the one set, make it robust, and wrap it up in a form that can be used across multiple projects.

Finally, remember that tools are no replacement for knowing how to write secure code. That's why security and privacy education are so important. You need a solid understanding of the concepts to make the judgment calls and insights your tools aren't capable of making.

Rule #7: Recognize the Strategic Asymmetry

This is one of my favorites. Remember that as a software developer, the security odds are stacked against you. I like to call this the "'Attacker's Advantage, and the Defender's Dilemma." You need to get the code and the designs 100 percent correct 100 percent of the time, and that is impossible. To make matters worse, you must reach this insurmountable goal on a fixed budget, on time, while having to consider requirements of supportability, compatibility, accessibility and other "-ilities." An attacker can spend as long as he wants to find one bug, and then announce to the world that your application is insecure.

In Rule #6, It is mentioned that you should stop writing new insecure code. For Rule #7, you should focus on all code because attackers attack all code regardless of age. Spend time reviewing old code for security vulnerabilities, and seriously consider deprecating old, insecure functionality. If you use agile development methods, you should think about dedicating one or more sprints to fixing old code to bring it up to the quality of newer code.

Rule #8: Use the Best Tools You Can

Finally, use the best tools you possibly can. I love source code analysis tools and I love any technology that helps me write more secure code. As I mentioned, tools are no panacea but they help. A lot! Tools also help scale the problem of source code analysis. Tools can scan huge quantities of code rapidly, much faster than a human could. And this helps give you a feeling for how "bad" some code might be.

A good trick is to compile code using the highest possible warning levels. If you see a large number of warnings in the code, perhaps the code has other bugs that were not found by the compiler or other tools. Such code should be subject to a greater degree of scrutiny before it ships (see Rule #3).

Check Your Progress 2

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) Gathering and agreeing on _____is fundamental to a successful project.

.....

.....

2) What are the two basic principles for Good Design?

.....

.....

.....

3) What factors do you consider in selecting a Right system?

.....

.....

.....

.....

2.8 SECURITY PRINCIPLES

Minimize Attack Surface Area

Every feature that is added to an application adds a certain amount of risk to the overall application. The aim for secure development is to reduce the overall risk by reducing the attack surface area.

For example, a web application implements online help with a search function. The search function may be vulnerable to SQL injection attacks. If the help feature was limited to authorized users, the attack likelihood is reduced. If the help feature's search function was gated through centralized data validation routines, the ability to perform SQL injection is dramatically reduced. However, if the help feature was re-written to eliminate the search function (through better user interface, for example), this almost eliminates the attack surface area, even if the help feature was available to the Internet at large.

Establish Secure Defaults

There are many ways to deliver an "out of the box" experience for users. However, by default, the experience should be secure, and it should be up to the user to reduce their security – if they are allowed.

For example, by default, password aging and complexity should be enabled. Users might be allowed to turn these two features off to simplify their use of the application and increase their risk.

Principle of Least Privilege

The principle of least privilege recommends that accounts have the least amount of privilege required to perform their business processes. This encompasses user rights, resource permissions such as CPU limits, memory, network and file system permissions.

For example, if a middleware server only requires access to the network, read access to a database table, and the ability to write to a log, this describes all the permissions that should be granted. Under no circumstances should the middleware be granted administrative privileges.

Principle of Defense in Depth

The principle of defense in depth suggests that where one control would be reasonable, more controls that approach risks in different fashions are better. Controls, when used in depth, can make severe vulnerabilities extraordinarily difficult to exploit and thus unlikely to occur.

With secure coding, this may take the form of tier-based validation, centralized auditing controls and requiring users to be logged on all pages.

For example, a flawed administrative interface is unlikely to be vulnerable to anonymous attack if it correctly gates access to production management networks, checks for administrative user authorization and logs all access.

Fail Securely

Applications regularly fail to process transactions for many reasons. How they fail can determine if an application is secure or not.

For example:

```
isAdmin = true;

try {codeWhichMayFail();

isAdmin = isUserInRole("Administrator"); }

Catch (Exception ex) {log,write(ex.to string());}
```

If either codeWhichMayFail() or isUserInRole fails or throws and exception, the user is an admin by default. This is obviously a security risk.

Don't Trust Services

Many organizations utilize the processing capabilities of third party partners, who more than likely have differing security policies and posture than you. It is unlikely that you can influence or control any external third party, whether they are home users or major suppliers or partners.

Therefore, implicit trust of externally run systems is not warranted. All external systems should be treated in a similar fashion.

For example, a loyalty program provider provides data that is used by Internet Banking, providing the number of reward points and a small list of potential redemption items. However, the data should be checked to ensure that it is safe to display to end users, and that the reward points are a positive number, and not improbably large.

Separation of Duties

A key fraud control is separation of duties. For example, someone who requests a computer cannot also sign for it, nor should they directly receive the computer. This prevents the user from requesting many computers and claiming they never arrived.

Certain roles have different levels of trust than normal users. In particular, administrators are different to normal users. In general, administrators should not be users of the application.

For example, an administrator should be able to turn the system on or off, set password policy but shouldn't be able to log on to the storefront as a super privileged user, such as being able to "buy" goods on behalf of other users.

Avoid Security by Obscurity

Security through obscurity is a weak security control, and nearly always fails when it is the only control. This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not be reliant upon keeping details hidden.

For example, the security of an application should not rely upon knowledge of the source code being kept secret. The security should rely upon many other factors, including reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls.

A practical example is Linux. Linux's source code is widely available, and yet when properly secured, Linux is a hardy, secure and robust operating system.

Keep Security Simple

Attack surface area and simplicity go hand in hand. Certain software engineering fads prefer overly complex approaches to what would otherwise be relatively straightforward and simple code.

Developers should avoid the use of double negatives and complex architectures when a simpler approach would be faster and simpler.

For example, although it might be fashionable to have a slew of singleton entity beans running on a separate middleware server, it is more secure and faster to simply use global variables with an appropriate mutex mechanism to protect against race conditions.

Fix Security Issues Correctly

Once a security issue has been identified, it is important to develop a test for it, and to understand the root cause of the issue. When design patterns are used, it

is likely that the security issue is widespread amongst all code bases, so developing the right fix without introducing regressions is essential.

For example, a user has found that they can see another user's balance by adjusting their cookie. The fix seems to be relatively straightforward, but as the cookie handling code is shared among all applications, a change to just one application will trickle through to all other applications. The fix must therefore be tested on all affected applications

2.9 CUSTOM APPLICATIONS AND THEIR SECURITY THREATS

For many organisations, their internal security professionals are adept at finding and responding to information about the latest vulnerabilities and threats to the software employed within business critical systems under their supervision. There are a great many security resources available, online and printed, ready to help explain and address potential vulnerabilities with the most common commercial software products. However, there exist two problems for those responsible for the security and integrity of your systems. Firstly, the "hit or miss" disclosure of vulnerabilities in commercial software, and secondly, how do you identify or address potential vulnerabilities in the custom (in-house developed) application that connects and runs atop the commercial software?

There is very little most organisations can do about the "hit or miss" disclosure of vulnerabilities inherent to the commercial software deployed throughout the business. You must rely upon the software provider to have initiated appropriate quality controls and security testing, and provide bug-fixes or patches as necessary. Due to various country specific laws or software agreements, it may actually be illegal for your organisation to reverse-engineer or otherwise identify security flaws with the software you have purchased.

In the case of custom (in-house developed) applications, it is normally possible to conduct a more thorough security review of the system. There are two paths available to an organisation wishing to review the security of their application; a code review and an application assessment. A code review tends towards more of an audit of coding practices utilised throughout the pre-compiled application, while an assessment reviews the functionality and resilience of the compiled application components to real-life threats. While a code review often initially appears to be more valuable, most organisations soon find to their amazement that such a review is extremely costly in both time and effort, and often fails to identify post compilation and deployment issues. An application assessment focuses on the compiled and installed elements of the entire system. Attention is made to how the application components are deployed, communicate or otherwise interact with both the user and server environments.

From past experience, the author has found that application assessments often represent the best value-for-money when identifying security threats and resolutions to distributed client-server applications.

Unfortunately, many organisations will find that they do not have sufficiently skilled resources available internally to cope with an application security assessment, and will have to seek security advice from a security specialist. Not all security organisations or consultancies can provide application security assessments, but those that do will normally classify the application or system into one of two classes; Web and Compiled.

This whitepaper aims to provide an organisation with enough information to understand the potential threats to your application beyond classical "ethical hacking" methodologies, and provide helpful advice on steps that can be taken to limit such security threats.

2.9.1 How Hackers Execute Attacks against Custom Applications

Most initial attacks against your systems are likely to focus upon the systems infrastructure and commercial applications, as that is where most of the exploit material and methodologies are likely to work. Depending upon the skills or resources, if your systems are well secured and up to date with the latest bug-fixes or patches, the attacker may either resort to social engineering or focus upon the manipulation of the vulnerabilities inherent in the application, or sub-components. Although custom applications can be vulnerable to a number of attack methodologies. Four attack categories are most prevalent:

- **Buffer Overflow Attacks:** Buffer overflow attacks are aimed at application components that take data as an input and pass it to memory buffers for later use and manipulation. Failure to adequately check the size of data before passing it into too small a buffer is commonplace. Attackers may be able to include their own embedded commands within the oversized data package and thus have their commands replace existing application code and execute on the system. If successfully executed, these commands will enable attackers to acquire privileges that exceed authorised permissions up to, and including, those of the system administrator — with corresponding control of the host system. In some circumstances, application components subject to a buffer overflow may suspend or crash, thus denying users further access to the service. In other cases such an attack may bring down the host server and deny access to any services provided by the system.

- **Race Conditions:** Under certain circumstances, when an application requires access to specific files, variables or data, its programmers may not have correctly implemented multiple simultaneous accesses and installed the appropriate checks. This can often lead to an attacker enjoying

unintended access to files or data through trusted and untrusted server application components.

- **Exploitation of Application Component Privileges:** Server based application components run with specific group or user permissions, not necessarily with that of the user running them (such as an anonymous Web user). These application components, if they suffer additionally from buffer overflows or race conditions, can be used to increase access and escalate the potential damage to the system.

- **Client-Side Manipulation:** To speed up connectivity and reduce performance loads at the server end, client-side validation of input and manipulation of data is often required. It is often a relatively trivial exercise for attackers to bypass this checking and supply incorrect data or data formats to the server in an attempt to initiate any of the other three common attack formats or to reveal both confidential information and server application functionality. This method is also a popular means of conducting fraudulent transactions on commerce-enabled sites by changing the values of available products.

2.9.2 Web-based Applications

As Internet businesses have matured, a new category of application has emerged that enables companies to interact with both potential and existing clients in this medium. These Web-based applications — while they rely on conventional corporate technologies such as database, mail or Web servers — are often subject to design compromises due to the limits of the medium. Such design compromises, unless carefully reviewed and analyzed, can expose the integrity of the application components and corporate data to avoidable security risks. In many cases, in fact, application risks far outweigh the threats to the supporting environment from traditional hacking techniques.

To make the best use of client-side Internet bandwidth and to deal with high volumes of simultaneous connections and data requests, Web-based applications tend to be distributed over multiple servers using a tiered architecture model. Moreover, they frequently rely on client-side code to present and manage data. These design considerations, coupled with the use of scripting-type development languages and constant changes in authentication and certification procedures, frequently lead to security flaws in an applications implementation. Increasingly, Internet-based attacks exploit these security flaws to compromise sites and gain access to critical systems.

Direct attacks against custom Web applications through manipulation of their inherent vulnerabilities have become more popular due to their relative ease and the scope they offer for maintaining anonymity. Although companies may install various security mechanisms to strengthen security — firewalls, intrusion detection systems, operating system hardening procedures, etc. —

they seldom expend much effort in securing and verifying the integrity of applications and coded pages against external attacks. In these circumstances, simple manipulation of client code or data, such as the price of goods in an online shopping basket application or sending corrupt and incorrect data to the server, can lead to fraudulent transactions or theft of confidential information.

It seems hardly a week goes by without the appearance of fresh newspaper stories describing how faulty application processes and input manipulation have led to the loss of confidential data such as banking details and credit card numbers. In almost all such cases, an understanding of manipulation techniques combined with rigorous client-side security testing would have identified the potential failure points and resulted in a more robust application, thus averting embarrassing, often dangerous compromises of system and data integrity.

2.9.3 The Web-based Application Security Assessment Process

The process of assessing the security of a web-based application, although not technically complex, often relies upon a multi-facetted approach utilizing a variety of technologies and techniques. Unfortunately, there is currently no quick shrink-wrapped solution available to automatically and comprehensively assess an application's security. Various vendors can supply testing products that will search for the most basic faults in non-complex applications/environments and provide advice on better coding practices. Based upon experience in assessing critical Web-enabled applications, automated tools should only be used for first-round security testing and preliminary identification of potential flaws.

Depending on your specific requirements and the type of web-based application, an application security assessment should typically consist of the following phases:

- Examination of external/client-side visible code for information that could be used for social engineering purposes or for information on how an application functions that might be used for a more focused attack.
- Discovery of information on the type of environment that exists at the server side (eg, embedded SQL queries specific to a single database version).
- Inspection of application validation and bounds checking — both for accidental *and* mischievous input. The purpose of this exercise is to ascertain the limits of correct server responses when handling unexpected data formats or sizes. This phase involves buffer overflow attempts to establish system resilience and performance continuity.
- Manipulation of client-side code and locally stored information such as cookies and session information. Client-side code is altered to subvert

authentication checking and used to establish the bounds of server reliance on client data fields. URL request information and GET/PUT requests are altered to achieve unexpected system responses and access confidential information.

- Examination of application-to-application interaction between system components such as the Web service and back-end data sources. Attempts are made to reference system components by impersonating other system functions or sources. Redirection methods and messaging functions are closely examined.

- Discovery of techniques that could be employed by attackers to escalate their permissions by referencing application components with higher server-side permissions, or exploitation of race conditions to identify lax permission or authentication checking.

- Attempts to subvert in-transit data between the client and server system. Examination of data delivery methods and the likelihood of their subversion or use in a replay-type attack, or other session orientated attacks, including an analysis of system responses to such data.

- Authentication methods in use are examined for their robustness and resilience to various subversion techniques. Attempts are made to bypass authentication processes and/or impersonate valid logged-in users. Detailed studies of user segregation methods are undertaken and an analysis of server-side responses to failed attempts is made.

- Overall examination of the application's deployment and security configuration from perceived threat models. Advice is given on secure deployment methodologies for the application type, based upon market considerations, new vulnerability developments and attack methodologies.

2.9.4 Compiled Applications

Although not as visible or exposed as most web applications, the security threats to compiled applications are very similar and often share the same principles of data integrity and resilience. However, the methodologies and tools necessary for testing the compiled application and sub-components are quite different. While the emphasis of a web application assessment is on manipulation of external client data; for a compiled application, the emphasis is on the communication methods and data streams between all internal system components.

The great diversity of compiled applications and their interaction with other commercial products (e.g. operating systems, databases, messaging systems and hardware) often means that assessment methodologies must be tuned to system at hand. For this reason, it is vital that both your organisation and the

assessment supplier fully understand the scope and reasons for undertaking the security assessment.

2.10 GENERAL ADVICE ON SECURING CUSTOM APPLICATIONS

The following are the best practices for securing applications to minimize the developer's time for correcting potential flaws that may be discovered during a security assessment and also to maximize security and data integrity during the process of building a new application:

1. Assume that someone out there will intentionally try to break your application and may have access to greater resources than you expended in developing it.
2. Assume that you will receive erroneous data from authorized, authenticated users, and develop a way to deal with it.
3. If you choose to use client-side input validation, ensure that the input is also validated at the server end.
4. Avoid complex code and use minimal coding structures for handling performance related issues.
5. Be careful with application component privileges. Always apply the minimum possible permissions needed to carry out any particular task.
6. Utilize methods of load limiting to prevent overloading application components or servers.
7. Check every return code from system components and functions to prevent race conditions or malicious input.
8. Never allow passwords or user specific details to be passed in plain-text to the client browser or between application components.
9. Ensure that confidential system information is not encoded into documents that could potentially be accessed by a remote client, either directly or through escalated application component permissions and calls.
10. Commonly available libraries or sample code may not necessarily be secure. Review carefully any third-party code.
11. Always use full pathnames in system and application procedure calls to prevent redirection and unexpected use.
12. Take extreme care with file permissions and access rights.

13. Remove all unnecessary material from the hosting servers and only install services that are required for the application/system to function.
14. Remove all comments and unnecessary information from client-side code.
15. Distribute application functions between servers when possible to limit data access from a compromised host.
16. Use reasonable system timeouts for network reads and writes to help prevent race conditions being reached and denial of service attacks.
17. Ensure you are capable of logging and tracking all inbound requests for post attack analysis.
18. Where possible, only use shared resources you have direct control over. If this is not possible, ensure appropriate checks for data integrity are made.
19. Test the application thoroughly!
20. Get a third party to perform an independent assessment both of the application's security and the system's host servers before going live.
21. Assume that someone out there will intentionally try to break your application and may have access to greater resources than you expended in developing it.
22. Assume that you will receive erroneous data from authorised, authenticated users, and develop a way to deal with it.
23. If you choose to use client-side input validation, ensure that the input is also validated at the server end.
24. Avoid complex code and use minimal coding structures for handling performance related issues.
25. Be careful with application component privileges. Always apply the minimum possible permissions needed to carry out any particular task.
26. Utilise methods of load limiting to prevent overloading application components or servers.
27. Check every return code from system components and functions to prevent race conditions or malicious input.
28. Never allow passwords or user specific details to be passed in plain-text to the client browser or between application components.
29. Ensure that confidential system information is not encoded into documents that could potentially be accessed by a remote client, either directly or through escalated application component permissions and calls.

30. Commonly available libraries or sample code may not necessarily be secure. Review carefully any third-party code.

31. Always use full pathnames in system and application procedure calls to prevent redirection and unexpected use.

32. Take extreme care with file permissions and access rights.

33. Remove all unnecessary material from the hosting servers and only install services that are required for the application/system to function.

34. Remove all comments and unnecessary information from client-side code.

35. Distribute application functions between servers when possible to limit data access from a compromised host.

36. Use reasonable system timeouts for network reads and writes to help prevent race conditions being reached and denial of service attacks.

37. Ensure you are capable of logging and tracking all inbound requests for post attack analysis.

38. Where possible, only use shared resources you have direct control over. If this is not possible, ensure appropriate checks for data integrity are made.

39. Test the application thoroughly!

40. Get a third party to perform an independent assessment both of the application's security and the system's host servers before going live.

Check Your Progress 3

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) How can you achieve the better productivity gains?

.....

.....

.....

.....

2) What is the use of defect tracking?

.....

.....

.....

.....

3) What is the aim of Buffer overflow attack?

.....

.....

.....

.....

4) What are the various security mechanisms can be installed by the company to strengthen security?

.....

.....

.....

.....

2.11 LET US SUM UP

In this unit, we stressed the point on "maintainable code". A coding standards document tells developers how they must write their code. We clarified why you need coding standards and also Advantages of coding standards. We listed out the good methods for coding and also how to do effective source code control. This unit also gives you the rules for developing secured code. Here, we listed Custom applications and their security threats, and also some General advice on securing custom applications.

2.12 CHECK YOUR PROGRESS: THE KEY

Check Your Progress 1

1) Developer

2) Quality assurance team

3) It is also beneficial for the organization who are applying for ISO 9001 license because coding standards is a complement from organization's execution plan requirements.

Check Your Progress 2

1) Requirements

2) The two basic principles for good design are "Keep it Simple" and information hiding.

3) The points to be considered in selecting a right system are:

a) Price

b) Concurrent development style

c) Repository

d) Internet friendliness

e) IDE integration

f) Cross-platform support.

Check Your Progress 3

1) Code reuse is but one form of reuse and there are other kinds of reuse that can provide better productivity gains such as open source software, better programming tools, common libraries, easier access to information.

2) By using defect tracking, it is possible to gauge when a project is ready to release.

3) Buffer overflow attacks are aimed at application components that take data as an input and pass it to memory buffers for later use and manipulation.

4) Companies may install various security mechanisms to strengthen security — firewalls, intrusion detection systems, operating system hardening procedures etc.

2.13 SUGGESTED READINGS

- Applications = Code + Markup: A Guide to the Microsoft® Windows® Presentation Foundation.
- Bakshi, Arun; Jindal, Gaurav; Aggrawal, Ritu and Chauhan, Sumit. *Programming in C using LINUX*, ISBN 978-81-241-1651-7, Haranand Publications,Delhi
- Cracking the Code: peer-to-peer Application Development by Dream tech software team *racking The Code.*
- Hohmann, Luke. Beyond. *Software Architecture: Creating and Sustaining Winning Solutions.*
- Martin Robert C. Clean Code: *A Handbook of Agile Software Craftsmanship.*

UNIT 3 APPLICATION TESTING

Structure

3.0 INTRODUCTION

What is software testing???

Application testing deals with tests for the entire application. This is driven by the scenarios from the analysis team. The application must successfully execute all scenarios before it is ready for general customer availability. After all, the scenarios are a part of the requirement document and measure success. Application testing represents the bulk of the testing done by industry.

Unlike the internal and unit testing, which are programmed, these test are usually driven by scripts that run the system with a collection of parameters and collect results. In the past, these scripts may have been written by hand but in many modern systems this process can be automated.

Most current applications have graphical user interfaces (GUI). Testing a GUI to assure quality becomes a bit of a problem. Most, if not all, GUI systems have event loops. The GUI event loop contains signals for mouse, keyboard, window, and other related events. Associated with each event are the coordinates on the screen of the event. The screen coordinates can be related

back to the GUI object and then the event can be serviced. Unfortunately, if some GUI object is positioned at a different location on the screen, then the coordinates change in the event loop. Logically the events at the new coordinates should be associated with the same GUI object. This logical association can be accomplished by giving unique names to all of the GUI objects and providing the unique names as additional information in the events in the event loop. The GUI application reads the next event off of the event loop, locates the GUI object and services the event.

The events on the event loop are usually generated by human actions such as typing characters, clicking mouse buttons, and moving the cursor. A simple modification to the event loop can journal the events into a file. At a later time, this file could be used to regenerate the events, as if the human was present, and place them on the event loop. The GUI application will respond accordingly. A tester, using the GUI, now executes a scenario. A journal of the GUI event loop from the scenario is captured. At a later time the scenario can be repeated again and again in an automated fashion. The ability to repeat a test is key to automation and stress testing.

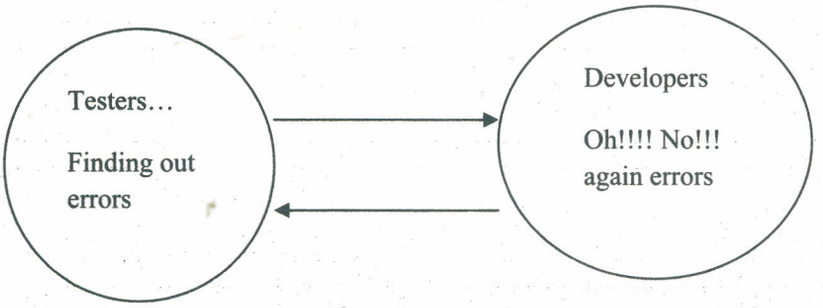
3.1 OBJECTIVES

After studying this unit, you should be able to:

- describe fundamental of Testing;
- describe the role of Testing in Application Life Cycle; and
- explain Management of Test.

3.2 FUNDAMENTAL OF TESTING

Testing is very important phase in application life cycle model. After doing coding part in software development, testers do testing. After finding out errors they again send the software to the development team to correct the errors.



Software testing also identifies important defects, flaws, or errors in the application code that must be fixed.

Application Development Life Cycle

Have we built the software right???

(i.e., does it match the specification).

Have we built the right software????

i.e., is this what the customer wants.

Who does the testing????

Software testing has three main purposes: **verification, validation, and defect finding.**

♦ **The verification process** confirms that the software meets its technical specifications. A "specification" is a description of a function in terms of a measurable output value given a specific input value under specific preconditions. A simple specification may be along the line of "a SQL query retrieving data for a single account against the multi-month account-summary table must return these eight fields <list> ordered by month within 3 seconds of submission."

♦ **The validation process** confirms that the software meets the business requirements. A simple example of a business requirement is "After choosing a branch office name, information about the branch's customer account managers will appear in a new window. The window will present manager identification and summary information about each manager's customer base: <list of data elements>." Other requirements provide details on how the data will be summarized, formatted and displayed.

♦ **A defect** is a variance between the expected and actual result. The defect's ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

Why Do Software Testing?

"A clever person solves a problem. A wise person avoids it."

Albert Einstein

3.3 ROLE OF TESTING

Software testing is not a one person job. It takes a team, but the team may be larger or smaller depending on the size and complexity of the application being tested.

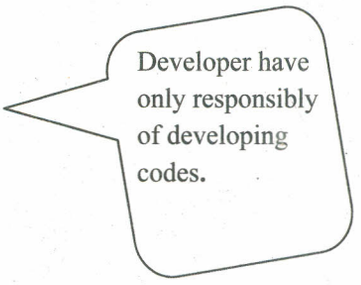
The programmer(s) who wrote the application should have a reduced role in the testing if possible. The concern here is that they're already so intimately involved with the product and "know" that it works that they may not be able to take an unbiased look at the results of their labors.

Testers must be cautious, curious, critical but non-judgmental, and good communicators. One part of their job is to ask questions that the developers might find not be able to ask themselves or are awkward, irritating, insulting or even threatening to the developers.

- How well does it work?

- What does it mean to you that "it works"?
- How do you know it works? What evidence do you have?
- In what ways could it seem to work but still have something wrong?
- In what ways could it seem to not work but really be working?
- What might cause it to not to work well?

A good developer does not necessarily make a good tester and vice versa, but testers and developers do share at least one major trait—they itch to get their hands on the keyboard. As laudable as this may be, being in a hurry to start can cause important design work to be glossed over and so special, subtle situations might be missed that would otherwise be identified in planning. Like code reviews, test design reviews are a good sanity check and well worth the time and effort. Testers are the only IT people who will use the system as heavily an expert user on the business side. User testing almost invariably recruits too many novice business users because they're available and the application must be usable by them. The problem is that novices don't have the business experience that the expert users have and might not recognize that something is wrong. Testers from IT must find the defects that only the expert users will find because the experts may not report problems if they've learned that it's not worth their time or trouble.



Key Players and Their Roles

Business sponsor(s) and partners

♦ Provides funding

♦ Specifies requirements and deliverables

♦ Approves changes and some test results

Project manager Plans and manages the project

Software developer(s)

♦ Designs, codes, and builds the application

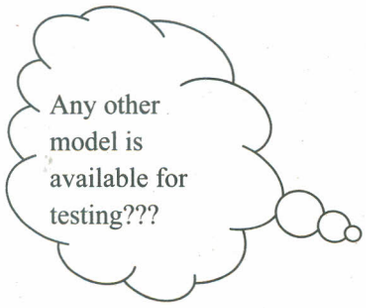
♦ Participates in code reviews and testing

♦ Fixes bugs, defects, and shortcomings

Testing Coordinator(s)

Creates test plans and test specifications based on the requirements and functional, and technical documents Tester(s) Executes the tests and documents results

Application Development Life Cycle



The V-Model of Software Testing

Software testing is too important to leave to the end of the project, and the V-Model of testing incorporates testing into the entire software development life cycle. In a diagram of the V-Model, the V proceeds down and then up, from left to right depicting the basic sequence of development and testing activities. The model highlights the existence of different levels of testing and depicts the way each relates to a different development phase. Like any model, the V-Model has detractors and arguably has deficiencies and alternatives but it clearly illustrates that testing can and should start at the very beginning of the project. (See Goldsmith for a summary of the pros and cons and an alternative. Marrik's articles provide criticism and an alternative.)

In the requirements gathering stage the business requirements can verify and validate the business case used to justify the project. The business requirements are also used to guide the user acceptance testing. The model illustrates how each subsequent phase should verify and validate work done in the previous phase, and how work done during development is used to guide the individual testing phases. This interconnectedness lets us identify important errors, omissions, and other problems before they can do serious harm. Application testing begins with Unit Testing and in the section titled "Testing Methods", we will discuss each of these test phases in more detail.

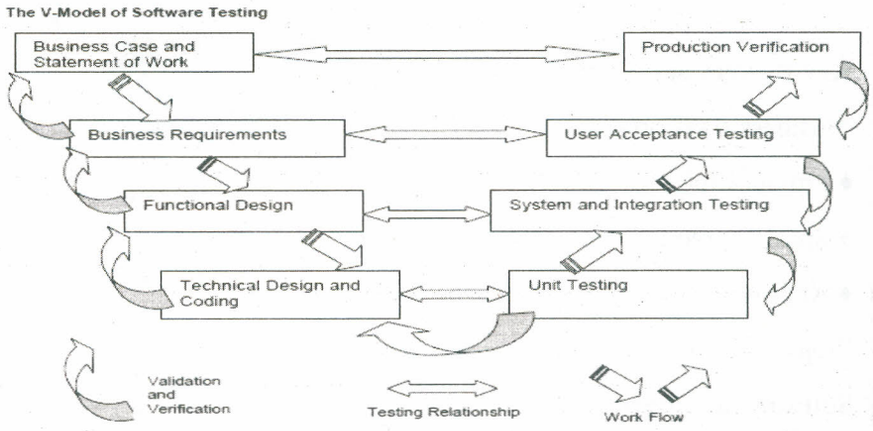


Fig.1

Types of Software Tests

The V-Model of testing identifies five software testing phases, each with a certain type of test associated with it.

Phase	Guiding Document	Test Type
Development Phase	Technical Design	Unit Testing
System and integration Phase	Fundamental Design	System Testing Integration Testing
User Acceptance Phase	Business Requirements	User Acceptance Testing
Implementation Phase	Business Case	Product Verification Testing
Regression Testing applies to all Phases		

The tester generally makes test cases. Now what is test plans and test cases?

What is a test case?

"A test case has components that describes an input, action or event and an expected response, to determine if a feature of an application is working correctly."

We will study about test case after testing methods.

3.4 TESTING METHODS-I

3.4.1 Type of Testing

There are many techniques available for doing testing. We just summaries it to understand what exactly we do in testing.

UNIT TESTING

Only one module

Unit Testing: A series of stand-alone tests are conducted during Unit Testing. Each test examines an individual component that is new or has been modified. A unit test is also called a module test because it tests the individual units of code that comprise the application.

For example, if you have two units and decide it would be more cost effective to glue them together and initially test them as an integrated unit, an error could occur in a variety of places:

Is the error due to a defect in unit 1?

Is the error due to a defect in unit 2?

Is the error due to defects in both units?

Is the error due to a defect in the interface between the units?

Is the error due to a defect in the test?

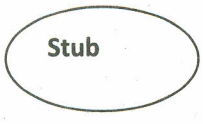
Module Testing: Module Testing is also known as unit testing. The large program cannot be tested all at once. so it is better practice that divide the whole program into small module. After that check each program with different test cases.

Benefits of Module Testing

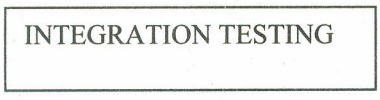
- Manage complexity of testing
- Facilitates debugging
- Encourages parallel testing

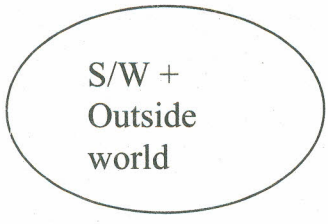
Testing Modules alone is not a easy task In module testing we have driver and stub programs.

Driver call a module and passes parameter to it.

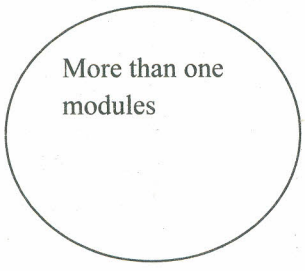


Stub represents an as-yet missing module. It not simply a place-holder but it must receive data from calling module. It must return valid values to calling module





Integration Testing: Integration testing examines all the components and modules that are new, changed, affected by a change, or needed to form a complete system. Where system testing tries to minimize outside factors, integration testing requires involvement of other systems and interfaces with other applications, including those owned by an outside vendor, external partners, or the customer. For example, integration testing for a new web interface that collects user input for addition to a database must include the database's ETL application even if the database is hosted by a vendor—the complete system must be tested end-to-end. In this example, integration testing doesn't stop with the database load; test reads must verify that it was correctly loaded.



System Testing: System Testing tests all components and modules that are new, changed, affected by a change, or needed to form the complete application. The system test may require involvement of other systems but this should be minimized as much as possible to reduce the risk of externally-induced problems. Testing the interaction with other parts of the complete system comes in Integration Testing. The emphasis in system testing is validating and verifying the functional design specification and seeing how all the modules work together. For example, the system test for a new web interface that collects user input for addition to a database doesn't need to include the database's ETL application—processing can stop when the data is moved to the data staging area if there is one. The first system test is often a **smoke test.** This is an informal quick-and-dirty run through of the application's major functions without bothering with details.

Acceptance Testing is done when the completed system is handed over from the developers to the customers or users. The purpose of acceptance testing is rather to give confidence that the system is working than to find errors.

3.4.2 Static Testing

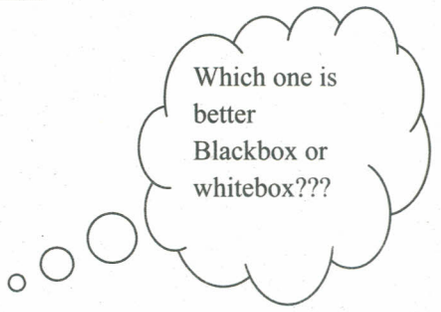
Testing of a component or system at specification or implementation level without execution of that software eg. reviews or static code analysis.

During static testing, software work products are examined manually or with a set of tools, but not executed.

Types of defect that can be easily found during static testing are: deviations from standards, missing requirements, design defects, non-maintenable code and inconsistent interface specifications.

3.5 TESTING METHODS-II

3.5.1 Dynamic Testing



Dynamic testing can be done in two ways

- Black Box Testing
- White Box Testing

3.5.1.1 Black Box Testing

Black box testing treats the system as a **"black-box"**, so it doesn't explicitly use Knowledge of the internal structure or code. Or in other words the Test engineer need not know the internal working of the "Black box" or application. **Main focus in black box testing is on functionality of the system as a whole.** The term **'behavioral testing'** is also used for black box testing and white box testing is also sometimes called **'structural testing'**. Behavioral test design is slightly different from black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged.

Each testing method has its own advantages and disadvantages. There are some bugs that cannot be found using only black box or only white box. Majority of the application are tested by black box testing method. We need to cover majority of test cases so that most of the bugs will get discovered by blackbox testing.

Black box testing occurs throughout the software development and Testing life cycle i.e in Unit, Integration, System, Acceptance and regression testing stages.

What is Black box testing?

Black box testing, also called functional testing and behavioral testing, focuses on determining whether or not a program does what it is supposed to do based on its functional requirements. Black box testing attempts to find errors in the external behavior of the code in the following categories

1. incorrect or missing functionality;
2. interface errors;
3. errors in data structures used by interfaces;
4. behavior or performance errors; and
5. initialization and termination errors.

Through this testing, we can determine if the functions appear to work according to specifications. However, it is important to note that no amount of testing can unequivocally demonstrate the absence of errors and defects in your code.

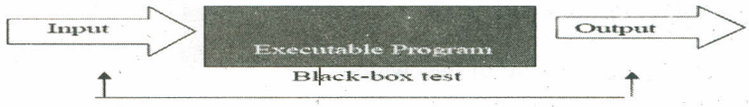


Fig. 2

Advantages of Black Box Testing

- Tester can be non-technical.
- Used to verify contradictions in actual system and the specifications.
- Test cases can be designed as soon as the functional specifications are complete

Disadvantages of Black Box Testing

- The test inputs needs to be from large sample space.
- It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult
- Chances of having unidentified paths during this testing

There are different techniques for doing black box testing. Many methods are available for this. Some popular techniques are mentioned below.

Techniques for Black Box Testing

1. Equivalence class partition

2. Boundary value analysis
3. Cause and effect graphics

Equivalence Class Partition

Bank A/c can be 500 to 1000 or 0 to 499 0r 2000(the field type is integer). What are equilance classes????

We define "Equivalence Class Partitioning" as a method that can help you derive test cases. You identify classes of input or output conditions. The rule is that each member in the class causes the same kind of behavior of the system. In other words, the "Equivalence Class Partitioning" method creates sets of inputs or outputs that are handled in the same way by the application.

There are 2 major steps we need to do in order to use equivalence class partitioning:

- Identify the equivalence classes of input or output. Take each input's or output's condition that is described in the specification and derive at least 2 classes for it:
    - One class that satisfies the condition – the **valid class**.
    - Second class that doesn't satisfy the condition – the **invalid class**.
    - Design test cases based on the equivalence classes.

Example 1

In a computer store, the computer item can have a quantity between -500 to +500. What are the equivalence classes?

Answer: Valid class: -500 <= QTY <= +500

Invalid class: QTY > +500

Invalid class: QTY < -500

Example 2

In a computer store, the computer item type can be P2, P3, P4 and P5 (each type influences the price). What are the equivalence classes?

Answer: Valid class: type is P2

Valid class: type is P3

Valid class: type is P4

Valid class: type is P5

Invalid class: type isn't P2, P3, P4 or P5

Boundary Value Analysis

More application **errors occur at the boundaries** of input domain. 'Boundary value analysis' testing technique is used to identify errors at boundaries rather than finding those exist in center of input domain.

Boundary value analysis is a next part of Equivalence partitioning for designing test cases where test cases are selected at the edges of the equivalence classes.

Test cases for input box accepting numbers between 1 and 1000 using Boundary value analysis:

1) Test cases with test data exactly as the input boundaries of input domain i.e. values 1 and 1000 in our case.
2) Test data with values just below the extreme edges of input domains i.e. values 0 and 999.
3) Test data with values just above the extreme edges of input domain i.e. values 2 and 1001.

Note: There is no hard-and-fast rule to test only one value from each equivalence class you created for input domains. You can select multiple valid and invalid values from each equivalence class according to your needs and previous judgments.

E.g. if you divided 1 to 1000 input values in valid data equivalence class, then you can select test case values like: 1, 11, 100, 950 etc. Same case for other test cases having invalid data classes.

Equivalence Class vs. Boundary Testing

Let us discuss about the difference between Equivalence class and boundary testing. For the discussion we will use the practice question:

Bank account can be integer in the following ranges: 500 to 1000 or 0 to 499 or 2000. What are the equivalence classes?

Answer:

- valid class: 0 <= account <= 499
- valid class: 500 <= account <= 1000
- valid class: 2000 <= account <= 2000
- invalid class: account < 0
- invalid class: 1000 < account < 2000

- invalid class: account > 2000

In equivalence class, you need to take one value from each class and test whether the value causes the system to act as the class' definition. It means that in this example, you need to create at least 6 test cases – one for each valid class and one for each invalid class.

How many test cases will be, if you use boundary testing?

The following table shows how much test cases will be using "Boundary Testing" method:

Test Case #	Value	Result
1	-1	Invalid
2	0	Valid
3	1	Valid
4	498	Valid
5	499	Valid
6	500	Valid
7	501	Valid
8	999	Valid
9	1000	Valid
10	1001	Invalid
11	1999	Invalid
12	2000	Valid
13	2001	Invalid

In boundary testing, you need to test each value in the boundary and you know the value, you don't need to choose it from any set. In this example you have 13 test cases.

Cause and Effect Graph

It considers only the desired external behaviour of a system. This is a testing technique that aids in selecting test cases that logically relate Causes (inputs) to

Effects (outputs) to produce test cases. A "Cause" represents a distinct input condition that brings about an internal change in the system. An "Effect" represents an output condition, a system transformation or a state resulting from a combination of causes.

According to Myer Cause & Effect Graphing is done through the following steps:

Step 1: For a module, identify the input conditions (causes) and actions (effect).

Step 2: Develop a cause-effect graph.

Step 3: Transform cause-effect graph into a decision table.

Step 4: Convert decision table rules to test cases. Each column of the decision table represents a test case.

Basic symbols used in Cause-effect graphs are as under:

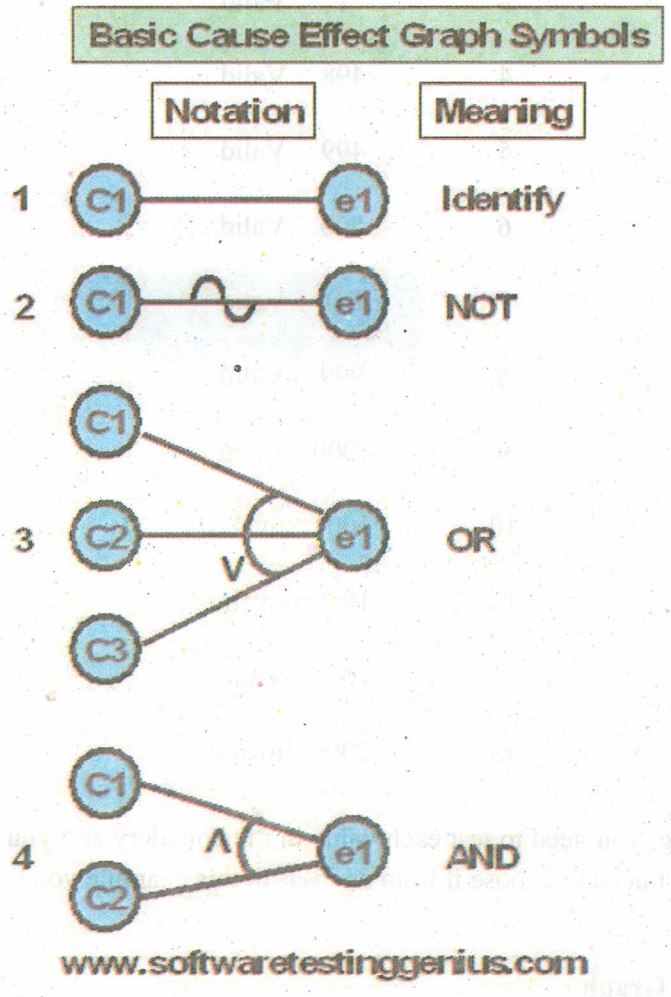


Fig.3

Going by the Myer methodology, test cases can be designed for the triangle problem in the following way:

Let design a cause and effect graph for a triangle to find out is triangle is scalene triangle, Isosceles triangle, Equilateral triangle or Impossible to make.

Step – 1: First of all we need to identify the causes and its effects.

The causes designated by letter "C" are as under

C1: Side "x" is less than sum of "y" and "z"

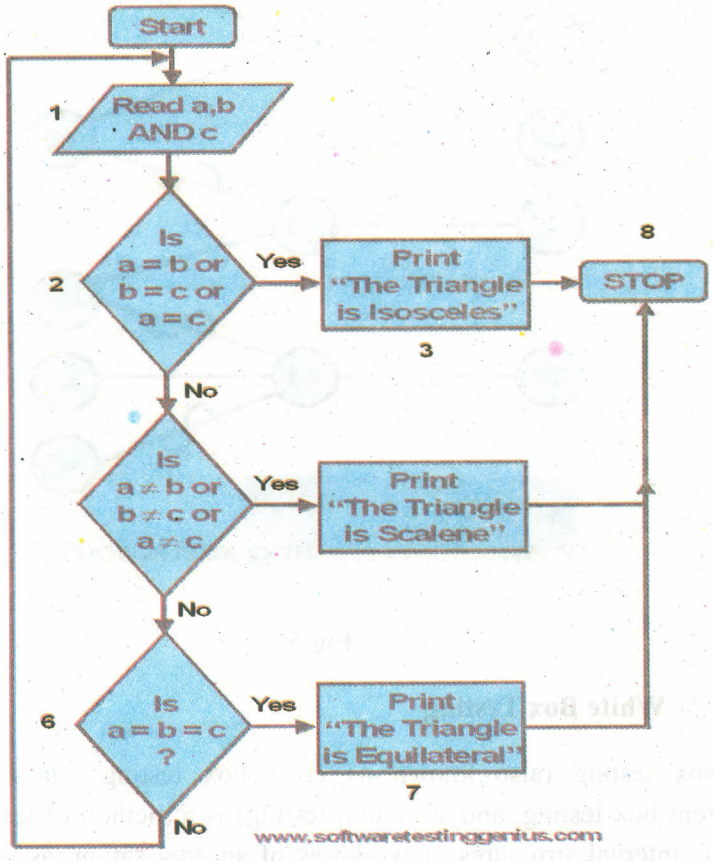


Fig.4

C2: Side "y" is less than sum of "x" and "z"

C3: Side "z" is less then sum of "x" and "y"

C4: Side "x" is equal to side "y"

C5: Side "x" is equal to side "z"

C6: Side "y" is equal to side "z"

The effects designated by letter "e" are as under

e1: Not a triangle

e2: Scalene triangle

e3: Isosceles triangle.

e4: Equilateral triangle

e5: Impossible

Step – 2: Draw the following cause-effect graph

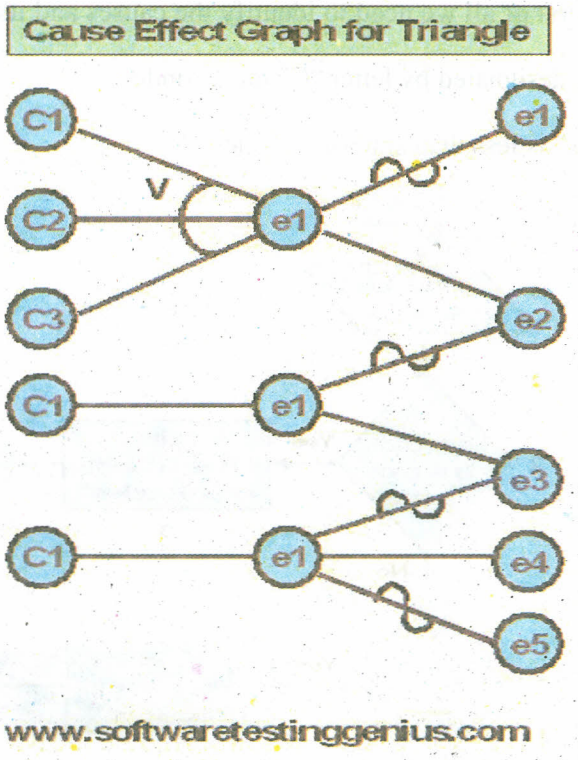


Fig.5

White-box testing is verification technique software engineers can use to examine if their code works as expected.

3.5.1.2 White Box Testing

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are required and used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

White-box test design techniques include:

- Control flow testing
- Data flow testing
- Branch testing

Path testing

Depends on above criteria white box has basic path testing technique which describe as follows

Basis path testing is a white-box testing technique first proposed by Tom McCabe. 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. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

Flow Graph Notation

The flow graph depicts logical control flow which is used to depict the program control structure.

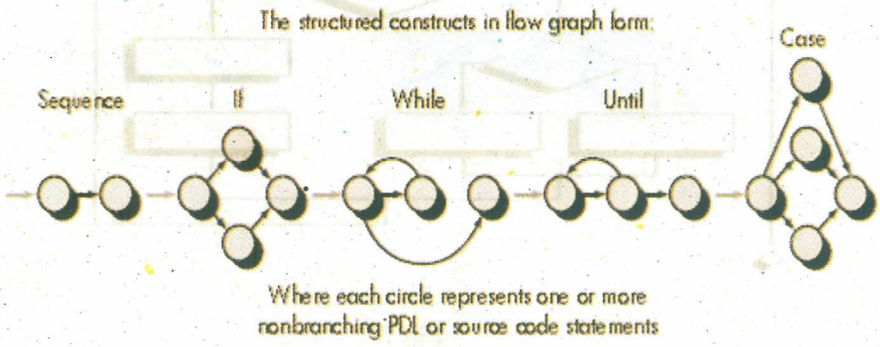


Fig.6

Referring to the figure, 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, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region. Each node that contains a

condition is called a predicate node and is characterized by two or more edges emanating from it.

Cyclomatic Complexity / Independent Program Path

Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

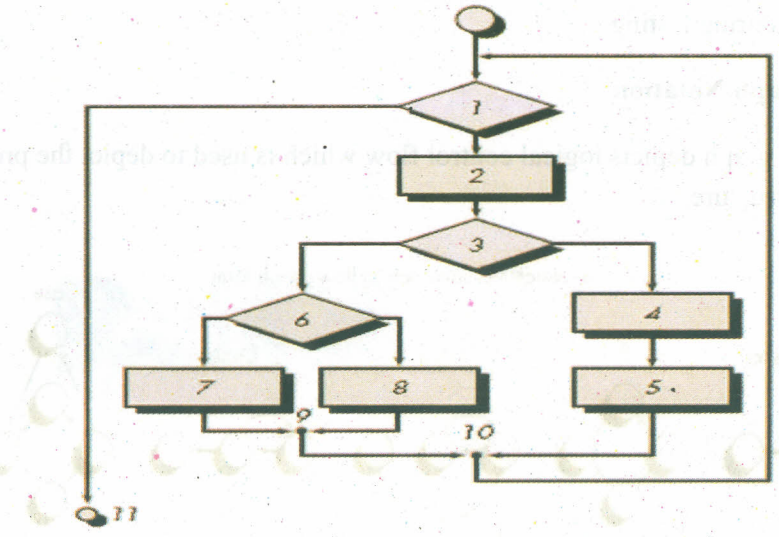


Fig 7

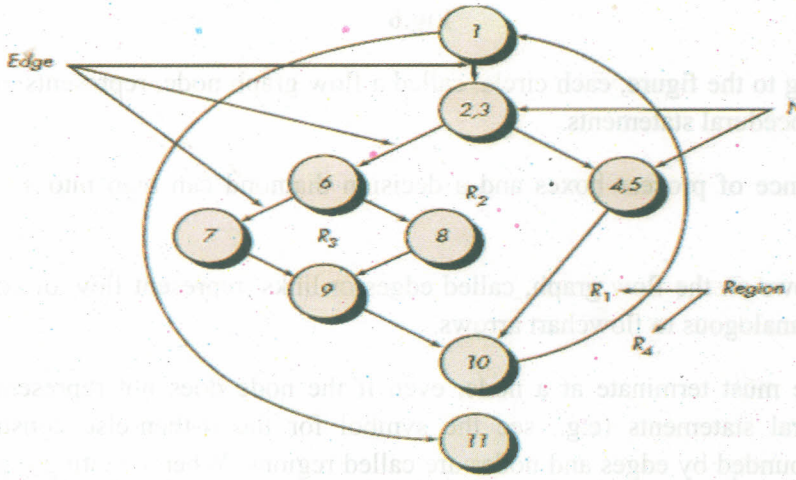


Fig 8

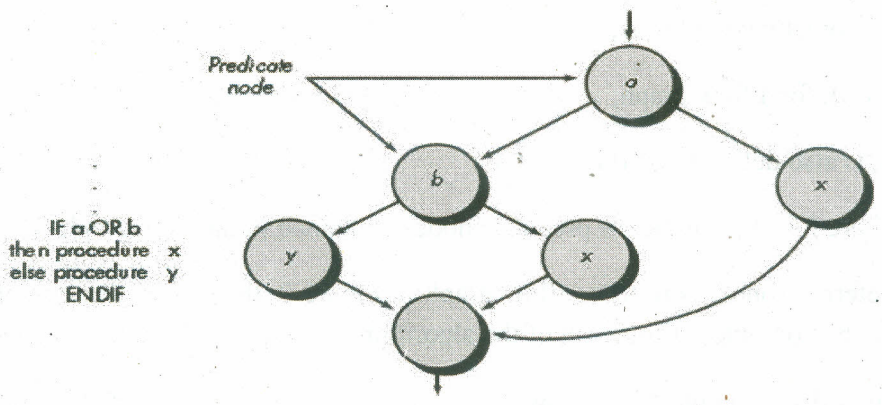


Fig 9

Example:

For example, a set of independent paths for the flow graph illustrated in Figure 9 is:

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 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Figure 9.

That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer. Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

The number of regions of the flow graph correspond to the cyclomatic complexity.

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, N is the number of flow graph nodes.

Cyclomatic complexity,

V(G), for a flow graph,

G, is also defined as $V(G) \doteq P + 1$

where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 9, the cyclomatic complexity can be computed using each of the algorithms just noted:

The flow graph has four regions.

V(G) = 11 edges - 9 nodes + 2 = 4.

V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 9 is 4.

More important, the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases

```
PROCEDURE average;
    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.
    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
      minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;                          2
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100         3
      increment total.input by 1;
      IF value[i] >= minimum AND value[i] <= maximum        5
        THEN increment total.valid by 1;
             sum = sum + value[i]                           7
        ELSE skip
      ENDIF                                                 8
      increment i by 1;
    ENDDO                                                   9
    IF total.valid > 0                                      10
      THEN average = sum / total.valid;                     11
      ELSE average = -999;                                  12
    ENDIF                                                   13
END average
```

Fig 10

1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules. Referring to the PDL for average in Figure 8, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 9.

2. Determine the cyclomatic complexity of the resultant flow graph. The cyclomatic complexity, V(G), is determined. It should be noted that V(G) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure average, compound conditions count as two) and adding 1. Referring to Figure 9,

   V(G) = 6 regions

   V(G) = 17 edges _ 13 nodes + 2 = 6

   V(G) = 5 predicate nodes + 1 = 6

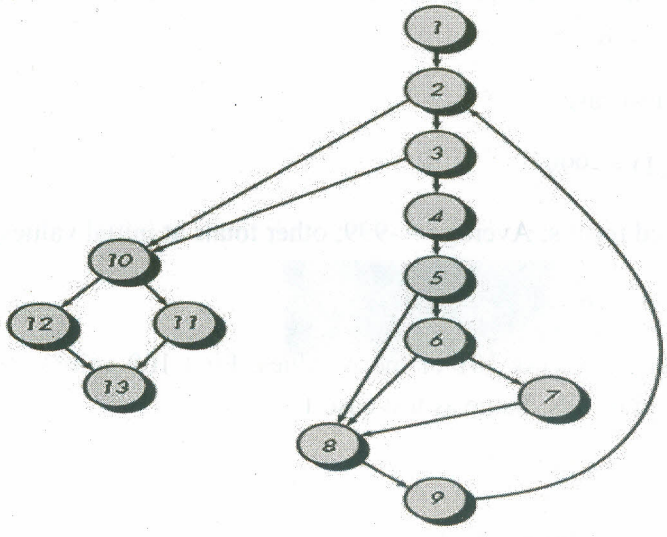


Fig 11

3. Determine a basis set of linearly independent paths. The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six paths:

   path 1: 1-2-10-11-13

   path 2: 1-2-10-12-13

   path 3: 1-2-3-10-11-13

   path 4: 1-2-3-4-5-8-9-2-. . .

   path 5: 1-2-3-4-5-6-8-9-2-. . .

   path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set just described are:

   Path 1 test case:

   Value (k) = valid input, where k < i for 2 = i = 100

   Value (i) = -999 where 2 = i = 100

   Expected results: Correct average based on k values and proper totals.

   Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

   Path 2 test case:

   Value (1) = -999

   Expected results: Average = -999; other totals at initial values.

   Path 3 test case:

   Attempt to process 101 or more values. First 100 values should be valid. Expected results: Same as test case 1.

   Path 4 test case:

   Value (i) = valid input where i < 100

   Value (k) < minimum where k < i

   Expected results: Correct average based on k values and proper totals.

   Path 5 test case:

   Value (i) = valid input where i < 100

   Value (k) > maximum where k <= i

   Expected results: Correct average based on n values and proper totals.

   Path 6 test case:

   value(i) = valid input where i < 100

   Expected results: Correct average based on n values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

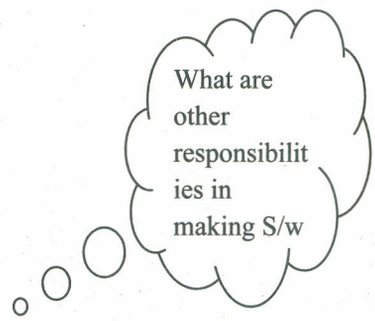
It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

Note: Errors are much more common in the neighbourhood of logical conditions than they are in the locus of the sequential processing.

Cyclomatic complexity provides the upper bound on the number of test cases that must be executed to guarantee that every statement in a component has been executed at least once.

3.6 MANAGEMENT OF APPLICATON TEST

3.6.1 Test Organization

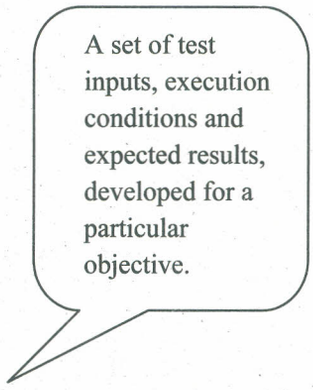


For managing the testing, tester makes one document which is known as test plans or test cases. This document also available for past references. The test plan is a mandatory document. You can't test without one. For simple, straight-forward projects the plan doesn't have to be elaborate but it must address certain items. As identified by the "American National Standards Institute and Institute for Electrical and Electronic Engineers Standard 829/1983 for Software Test Documentation", the following components should be covered in a software test plan

Table.1: Items Covered by a Test Plan

Items Covered by a Test Plan

Component	Description	Purpose
Responsibilities	Specific people who are and their assignments	Assigns responsibilities and keeps everyone on track and focused
Assumptions	Code and systems status and availability	Avoids misunderstandings about schedules
Test	Testing scope, schedule, duration, and prioritization	Outlines the entire process and maps specific tests
Communication	Communications plan—who, what, when, how	Everyone knows what they need to know when they need to know it
Risk Analysis	Critical items that will be tested	Provides focus by identifying areas that are critical for success
Defect Reporting	How defects will be logged and documented	Tells how to document a defect so that it can be reproduced, fixed, and retested
Environment	The technical environment, data, work area, and interfaces used in testing	Reduces or eliminates misunderstandings and sources of potential delay



There are levels in which each test case will fall in order to avoid duplication efforts.

Level 1: In this level you will write the basic test cases from the available specification and user documentation.

Level 2: This is the practical stage in which writing test cases depend on actual functional and system flow of the application.

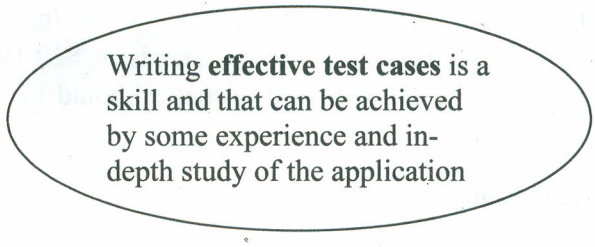
Level 3: This is the stage in which you will group some test cases and write a test procedure. Test procedure is nothing but a group of small test cases maximum of 10.

Level 4: Automation of the project. This will minimize human interaction with system and thus QA can focus on current updated functionalities to test rather than remaining busy with regression testing.

Why we write test cases?

The basic objective of writing test cases is to validate the testing coverage of the application. If you are working in any CMM company then you will strictly follow test cases standards. So writing test cases brings some sort of standardization and minimizes the ad-hoc approach in testing.

3.6.2 Tips of Creating Effective Test Cases



There are some tips to write effective test cases which are as follows.

1. Template – need one and complete template for creating test cases. We can create it in a text editor, spreadsheet or buy or customize the tool.

2. Descriptive and specific – you need a short, concise title and description. Should clearly define the purpose and scope of their operations.

3. Reusable – one and the same test case can be used in many scenarios.For ex. You write a test case for checking number is negative or not. This test case must run in different program.

4. Atomic – the greatest tester nightmare struggled with the manual testing is a test case, which is too long and doing too much dependency checks. Tester may have to perform 100 test cases and do them in one day, but it can also have their 10 and do it for a week. Let us create test case, which can be

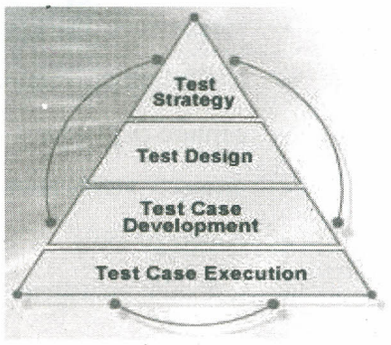
performed up to several minutes, and gain the favor of the other fellow testers, among the satisfied, we can also include those who write automation scripts.

5. Positive and negative – the next important thing is the order of creation. What to do when the project manager requests a set of test cases on the day following approval of an implementation plan? Usually we do not say that he wanted only the beginning of positive tests, those which check whether the functionality exists and acts in accordance with the requirement. Give him as such set as soon as possible, then we can deal with the preparation of a negative description situation and the relationship between other requirements. To sum up: Positive -> Negative -> Relationships

6. Refactor – if you already done your job we have to remember that applications are changing and the changes in the test cases should follow this process. We can version it, using clone and change method for a given version number. Head of the tests should determine how many past versions we support and we should care enough about our current collection of test cases.

7. Test data – there are moments in which along the correct execution of the test we have to provide some data, which usually (if they are too complex to be described in the text) are attached as binary files. We need to analyze whether it is easier to provide test data or may be easier to add the appropriate test steps.

8. Setup and tear down – if the test case requires the initial configuration, call a particular component, we need to take this into account in the zero step. It is also important to cleanup in the last step like in unit testing or automation, but I believe that during the test manual is worth to preserve this good practice.



3.6.3 Characteristics of Good Test Cases

There are some characteristics for good test cases

- Realistic quality levels for goals
- Includes time for planning
- Can be monitored and updated
- Includes user responsibilities
- Based on past experience
- Recognizes learning curves

In last testing is like a teacher in the class…which find out progress, feasibility and defects in the application…

Now how can you define which test case is better…???? And this test case gives you proper result..??? We have some characteristics for test cases also.

It is capable for finding errors. So first one is it test plan should find defect in the program.

Find Defects

- Have high probability of finding a new defect.
- Unambiguous tangible result that can be inspected.
- Repeatable and predictable.

The test plan also check the feasibility of the project

- Traceable to requirements or design documents
- Push systems to its limits
- Execution and tracking can be automated
- Do not mislead
- Feasible

3.6.4 Testing Tools

These tools are available for testing:-

- automatic test generation
- source code analysis
- configuration/version control
- capture & playback
- property verification
- coverage tools
- test suite management

3.7 PROCESS OF CREATING EFFECTIVE TEST CASES

Writing effective test cases is a skill and that can be achieved by some experience and in-depth study of the application on which test cases are being written.

A test case

"A test case has components that describes an input, action or event and an expected response, to determine if a feature of an application is working correctly."

There are levels in which each test case will fall in order to avoid duplication efforts.

Level 1: In this level you will write the basic test cases from the available specification and user documentation.

Level 2: This is the practical stage in which writing test cases depend on actual functional and system flow of the application.

Level 3: This is the stage in which you will group some test cases and write a test procedure. Test procedure is nothing but a group of small test cases maximum of 10.

Level 4: Automation of the project. This will minimize human interaction with system and thus QA can focus on current updated functionalities to test rather than remaining busy with regression testing.

So you can observe a systematic growth from no testable item to a Automation suit.

The format of the Test Cases may be as illustrated below:

Test Case ID

Test Case Description:

What to Test?

How to Test?

Input Data

Expected Result

Actual Result

Sample Test Case Format:

Test Case ID	Test Case Description	What To Test?	How to Test?	Input Data	Expected Result	Actual Result	Pass/Fail

Additionally the following information may also be captured:

a) Test Suite Name

b) Tested By

c) Date

d) Test Iteration (The Test Cases may be executed one or more times)

Objective behind test cases

The basic objective of writing test cases is to validate the testing coverage of the application. If you are working in any CMMi company then you will strictly follow test cases standards. So writing test cases brings some sort of standardization and minimizes the ad-hoc approach in testing.

TIPS FOR WRITING TEST CASES

One of the most frequent and major activity of a Software Tester (SQA/SQC person) is to write Test Cases. First of all, kindly keep in mind that all this discussion is about 'Writing Test Cases' not about designing/defining/identifying TCs.

There are some important and critical factors related to this major activity.Let us have a bird's eye view of those factors first.

a. Test Cases are prone to regular revision and update:

We live in a continuously changing world, software are also not immune to changes. Same holds good for requirements and this directly impacts the test cases. Whenever, requirements are altered, TCs need to be updated. Yet, it is not only the change in requirement that may cause revision and update to TCs.

During the execution of TCs, many ideas arise in the mind, many sub-conditions of a single TC cause update and even addition of TCs. Moreover, during regression testing several fixes and/or ripples demand revised or new TCs.

b. Test Cases are prone to distribution among the testers who will execute these:

Of course there is hardly the case that a single tester executes all the TCs. Normally there are several testers who test different modules of a single application. So the TCs are divided among them according to their owned areas of application under test. Some TCs related to integration of application, may be executed by multiple testers while some may be executed only by a single tester.

c. Test Cases are prone to clustering and batching:

It is normal and common that TCs belonging to a single test scenario usually demand their execution in some specific sequence or in the form of group. There may be some TCs pre-requisite of other TCs. Similarly, according to the business logic of AUT, a single TC may contribute in several test conditions and a single test condition may consist of multiple TCs.

d. Test Cases have tendency of inter-dependence:

This is also an interesting and important behavior of TCs that those may be interdependent on each other. In medium to large applications with complex business logic, this tendency is more visible.

The clearest area of any application where this behavior can definitely be observed is the interoperability between different modules of same or even different applications. Simply speaking, wherever the different modules or applications are interdependent, the same behavior is reflected in the TCs.

e. Test Cases are prone to distribution among developers (especially in TC driven development environment):

An important fact about TCs is that these are not only to be utilized by the testers. In normal case, when a bug is under fix by the developers, they are indirectly using the TC to fix the issue. Similarly, where the TCD development is followed, TCs are directly used by the developers to build their logic and cover all scenarios, addressed by TCs, in their code.

SO, KEEPING THE ABOVE 5 FACTORS IN MIND, HERE ARE SOME TIPS TO WRITE TEST CASES:

1. Keep it simple but not too simple; make it complex but not too complex:

This statement seems a paradox, but I promise it is not so. Keep all the steps of TCs atomic, precise with correct sequence and with correct mapping to expected results, this is what I mean to make it simple.

Now, making it complex in fact means to make it integrated with the Test Plan and other TCs. Refer to other TCs, relevant artifacts, GUIs etc. where and when required. But do this in balanced way, do not make tester to move to and fro in the pile of documents for completing single test scenario. On the other hand do not even let the tester wish you had documented these TCs in some compact manner. While writing TCs, always remember that you or someone else will have to revise and update these.

2. After documenting Test cases, review once as Tester:

Never think that the job is done once you have written the last TC of the test scenario. Go to the start and review all the TCs once, but not with the mind of

TC writer or Testing Planner. Review all TCs with the mind of a tester. Think rationally and try to dry run your TCs. Evaluate that all the Steps you have mentioned are clearly understandable, and expected results are in harmony with those steps.

The test data specified in TCs is feasible not only for actual testers but is according to real time environment too. Ensure that there is no dependency conflict among TCs and also verify that all references to other TCs/artifacts/GUIs are accurate because, testers may be in great trouble otherwise.

3. Bound as well as ease the testers:

Do not leave test data on testers, give them range of inputs especially where calculations are to be performed or application's behavior is dependent on inputs. You may divide the test item values among them, but never give them liberty to choose the test data items themselves. Because, intentionally or unintentionally, they may use same test data and some important test data may be ignored during the execution of TCs.

Keep the testers eased by organizing TCs according to the testing categories and related areas of application. Clearly instruct and mention which TCs are inter-dependent and/or batched. Similarly, explicitly indicate which TCs are independent and isolated so that tester may manage his overall activity at his or her own will.

4. Be a Contributor:

Never accept the FS or Design Document as it is. Your job is not just to go through the FS and identifying the Test Scenarios. Being a quality related resource, never hesitate to contribute. Suggest to developers too, especially in TC-driven development environment. Suggest the drop-down-lists, calendar controls, selection-list, group radio buttons, more meaningful messages, cautions, prompts, improvements related to usability etc.

5. Never Forget the End User

The most important stakeholder is the 'End User' who will actually use the AUT. So, never forget him at any stage of TCs writing. In fact, End User should not be ignored at any stage throughout the SDLC, yet my emphasis so far is just related to my topic. So, during the identification of test scenarios, never overlook those cases which will be mostly used by the user or are business critical even of less frequent use. Imagine yourself as End User, once go through all the TCs and judge the practical value of executing all your documented TCs.

Check Your Progress 1

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) What is unit testing how it is different from module testing?

.....

.....

.....

.....

2) Which one is better Black box testing or white box testing?

.....

.....

.....

.....

3) How to calculate cylomatric complexity?

.....

.....

.....

.....

4) Explain V model of Testing?

.....

.....

.....

.....

5) What is Test case and write characteristics of good test cases?

.....

.....

.....

.....

6) Why application testing is so important? Give one example.

.....

.....

.....

.....

7) What is test cases? How it is important?

.....

.....

.....

.....

8) Is black box testing gives good result? Comment.

.....

.....

.....

.....

9) Bank A/c can be 500 to 1000 or 0 to 499 0r 2000(the field type is integer). What are equilance classes?

.....

.....

.....

.....

10) Make one program for addition, subtraction and multiplication of two numbers. Find out in which cases your program will give errors?

.....

.....

.....

.....

3.8 LET US SUM UP

Testing is a major component of software development, and is a major science in itself. Software development involves developing software against a set of requirements (whether they be a specific client's requirements for a project, or generic requirements for a shrink-wrapped product).

Software testing is needed to verify and validate that the software that has been built has been built to meet these specifications. Testing ensures that what you get in the end is what you wanted to build. Testing enhances the integrity of a system by detecting deviations in design and errors in the system. Testing aims at detecting error-prone areas. This helps in the prevention of errors in a system. Testing also adds value to the product by conforming to the user requirements.

Another way to say this is: The desired result of testing is a level of confidence in the software so that the organization is confident that the software has an acceptable defect rate. What constitutes an acceptable defect rate depends on the nature of the software.

Another way of saying this in an even more business like manner is: Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors.

And what are some the common things that necessitate the need for a testing plan:

1. Communication gaps between the requirement specifier and the developer
2. Over commitment by the developer
3. Not proper understanding of all the complexities of the system
4. Inadequate requirements gathering

3.9 CHECK YOUR PROGRESS: THE KEY

1. **Unit testing and how it is different from module testing.**

   A series of stand-alone tests are conducted during Unit Testing. Each test examines an individual component that is new or has been modified. Some time Unit testing and module testing are one and the same thing, some time it may be different. Unit testing checks each function of program but module testing check the group of functions. For example In a calculator program if tester will check program for addition using integer, etc. data type then it will called unit testing. But if tester checks the whole module for addition then it is called module testing.

2) There are some pros and cons for both the testing techniques , which are as follows

   Pros of Black box testing

   - **The focus is on the goals of the software** with a requirements-validation approach to testing. Thanks Roger for pointing that out on the previous post. These tests are most commonly used for functional testing.
   - **Easier to staff a team**. We don't need software developers or other experts to perform these tests (note: expertise *is* required to identify which tests to run, etc). Manual testers are also easier to find at lower

rates than developers – presenting an opportunity to save money, or test more, or both.

Cons of Black box testing

- **Higher maintenance cost with automated testing**. Application changes tend to break black-box tests, because of their reliance on the constancy of the interface.
- **Redundancy of tests**. Without insight into the implementation, the same code paths can get tested repeatedly, while others are not tested at all.

Pros of White box testing

- **More efficient automated testing**. Unit tests can be defined that isolate particular areas of the code, and they can be tested independently. This enables faster test suite processing
- **More efficient debugging of problems.** When a regression error is introduced during development, the source of the error can be more efficiently found – the tests that identify an error are closely related (or directly tied) to the troublesome code. This reduces the effort required to find the bug.

Cons of White box testing

- **Harder to use to validate requirements**. White box tests incorporate (and often focus on) how something is implemented, not why it is implemented. Since product requirements express "full system" outputs, black box tests are better suited to validating requirements. Carefull white box tests can be designed to test requirements.
- **Hard to catch misinterpretation of requirements**. Developers read the requirements. They also design the tests. If they implement the wrong idea in the code because the requirement is ambiguous, the white box test will also check for the wrong thing. Specifically, the developers risk testing that the wrong requirement is properly implemented.
- **Hard to test unpredictable behavior**. Users will do the strangest things. If they aren't anticipated, a white box test won't catch them. I recently saw this with a client, where a bug only showed up if the user visited all of the pages in an application (effectively caching them) before going back to the first screen to enter values in the controls.

- **Requires more expertise and training**. Before someone can run tests that utilize knowledge of the implementation, that person needs to learn about how the software is implemented.

It depends upon the team size, nature of project which testing is suitable. Developers utilize white box tests to prevent submission of bugs to a testing team that uses black box tests to validate that requirements have been met (and to perform system level testing). This approach also allows for a mixture of manual and automated testing. Any continuous integration strategy should utilize both forms of testing.

3) **Calculation of cyclomatic complexity**.

Make graphs for calculating cyclomatic complexity. The equation for calculating the cyclomatic complexity number comes from the theory of graphs where it refers to the number of paths from any point in a topological space to any other point. It is expressed as: CCN = E - N + P

4) V model of Testing

V-model is for testing the software.

Business Case<----------->Release Testing

Requirement<----------->Acceptance Testing

System Spec<----------->System Testing

System Design<----------->Interface Testing

Component Design<--->Component Testing

Construct

Component

5) **Test case and characteristics of good test cases**

For managing the testing, tester makes one document which is known as test plans or test cases .This document also available for past references. The test plan is a mandatory document. You can't test without one. For simple, straight-forward projects the plan doesn't have to be elaborate but it must address certain items. Characteristics of good test plan

- Realistic quality levels for goals
- Includes time for planning
- Can be monitored and updated
- Includes user responsibilities

- Based on past experience
- Recognizes learning curves

6) **Application testing is so important.**

Application testing deals with tests for the entire application. This is driven by the scenarios from the analysis team. Application limits and features are tested here.

The application must successfully execute all scenarios before it is ready for general customer availability. After all, the scenarios are a part of the requirement document and measure success. Application testing represents the bulk of the testing done by industry.

7) **Test cases**

A **test case** in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. The mechanism for determining whether a software program or system has passed or failed such a test is known as a **test oracle**. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is considered sufficiently scrutinized to be released. Test cases are often referred to as **test scripts**, particularly when written. Written test cases are usually collected into test suites

8) Depending upon the nature of software it is decidable that black box testing is good or give suitable result.

9) Bank A/c can be 500 to 1000 or 0 to 499 0r 2000(the field type is integer). Equilance classes:

- valid class: 0 <= account <= 499
- valid class: 500 <= account <= 1000
- valid class: 2000 <= account <= 2000
- invalid class: account < 0
- invalid class: 1000 < account < 2000
- invalid class: account > 2000

In equivalence class, you need to take one value from each class and test whether the value causes the system to act as the class' definition. It means that in this example, you need to create at least 6 test cases – one for each valid class and one for each invalid class.

10) One program for addition, subtraction and multiplication of two numbers and cases in which program will give errors:

I am making program for addition in C

```
#include<stdio.h>
#include<conio.h>
Void main()
{Int num1,num2,mul;
Printf("Enter first number");
Printf("Enter second number");
Scanf("%d",&num1);
Scanf("%d",&num2);
Mul=num1*num2;
Printf("your multiplication is,%d",mul); }
```

Now in this case test case will be

1) if user will enter double,float values
2) If user will enter zero.
3) If user will enter large value

3.10 SUGGESTED READINGS

- Dustin. *Effective software testing.*
- Sriniwasan. *Software Testing Principal and Practice*
- www.softwaretestinggenius.com

UNIT 4 APPLICATION PRODUCTION AND MAINTENANCE

Structure

4.0 INTRODUCTION

Many software development organizations, including many product and online services groups within Microsoft, use agile software development and management methods to build their applications. Historically, security has not been given the attention it needs when developing software with agile methods. Since agile methods focus on rapidly creating features that satisfy customers' direct needs, and security is a customer need, it's important that it not be overlooked. In today's highly interconnected world, where there are strong regulatory and privacy requirements to protect private data, security must be treated as a high priority.

There is a perception today that agile methods do not create secure code, and, on further analysis, the perception is reality. There is very little "secure Agile" expertise available in the market today. This needs to change. But the only way the perception and reality can change is by actively taking steps to integrate security requirements into agile development methods.

Microsoft has embarked on a set of software development process improvements called the Security Development Lifecycle (SDLC). The SDLC has been shown to reduce the number of vulnerabilities in shipping software by more than 50 percent. However, from an Agile viewpoint, the SDLC is heavyweight because it was designed primarily to help secure very large products, such as Windows® and Microsoft Office, both of which have long development cycles.

If agile practitioners are to adopt the SDLC, two changes must be made. First, SDLC additions to agile processes must be lean. This means that for each feature, the team does just enough SDLC work for that feature before working on the next one. Second, the development phases (design, implementation, verification, and release) associated with the classic waterfall-style SDLC do not apply to Agile and must be reorganized into a more Agile-friendly format. To this end, the SDLC team at Microsoft developed and put into practice a streamlined approach that melds agile methods and security—the Security Development

4.1 OBJECTIVES

After studying this unit, you should be able to:

- know about application development life cycle;
- describe phases of application development life cycle; and
- explain production methods.

4.2 LIFE CYCLE

A letter from national University of Educational Planning & Administration Operations and Maintenance is the fourth phase of the SDLC. In this phase, systems are in place and operating, enhancements and/or modifications to the system are developed and tested, and hardware and/or software is added or replaced. The system is monitored for continued performance in accordance with security requirements and needed system modifications are incorporated. The operational system is periodically assessed to determine how the system can be made more effective, secure, and efficient. Operations continue as long as the system can be effectively adapted to respond to an organization's needs while maintaining an agreed-upon risk level. When necessary modifications or changes are identified, the system may reenter a previous phase of the SDLC.

Key security activities for this phase include:

- Conduct an operational readiness review;
- Manage the configuration of the system;
- Institute processes and procedures for assured operations and continuous monitoring of the information system's security controls; and
- Perform reauthorization as required.

4.2.1 Application Development Life Cycle: An Overview

An application is a collection of programs that satisfies certain specific requirements (resolves certain problems). The solution could reside on any platform or combination of platforms, from a hardware or operating system point of view.

As with other operating systems, application development on z/OS® is usually composed of the following phases:

- Design phase
- Gather requirements.
  - User, hardware and software requirements
  - Perform analysis.
  - Develop the design in its various iterations:
    - High-level design
    - Detailed design
  - Hand over the design to application programmers.
- Code and test application.
- Perform user tests.

User tests application for functionality and usability.

- Perform system tests.
  - Perform integration test (test application with other programs to verify that all programs continue to function as expected).
  - Perform performance (volume) test using production data.
- Go into production–hand off to operations.
- Ensure that all documentation is in place (user training, operation procedures).
- Maintenance phase--ongoing day-to-day changes and enhancements to application.

Figure.1 shows the process flow during the various phases of the application development life cycle.

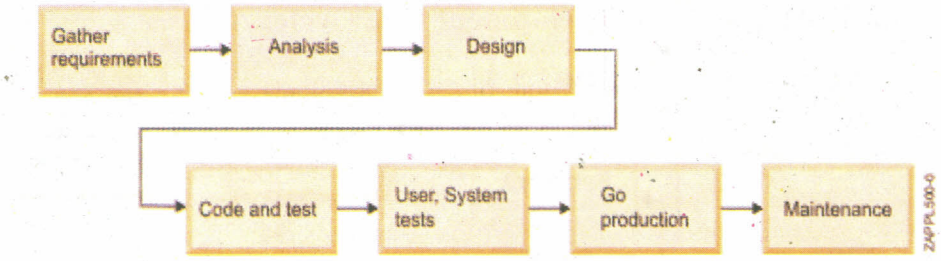


Fig. 1: Application development life cycle

Figure. 2 depicts the design phase up to the point of starting development. Once all of the requirements have been gathered, analyzed, verified, and a design has been produced, we are ready to pass on the programming requirements to the application programmers.

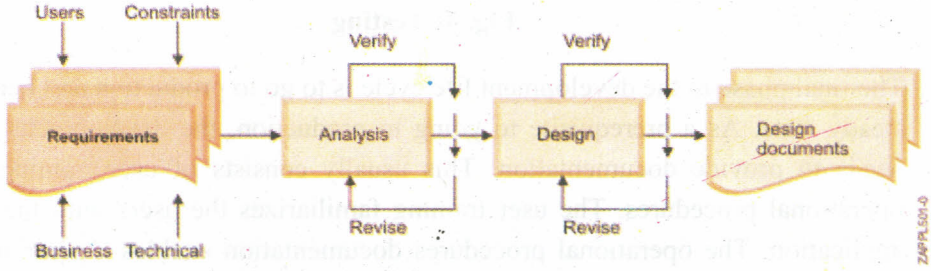


Fig. 2: Design phase

The programmers take the design documents (programming requirements) and then proceed with the iterative process of coding, testing, revising, and testing again, as we see in Figure. 3.

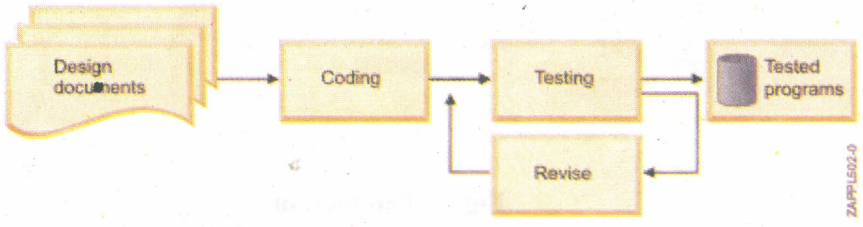


Fig. 3: Development phase

After the programs have been tested by the programmers, they will be part of a series of formal user and system tests. These are used to verify usability and functionality from a user point of view, as well as to verify the functions of the application within a larger framework (Figure. 4).

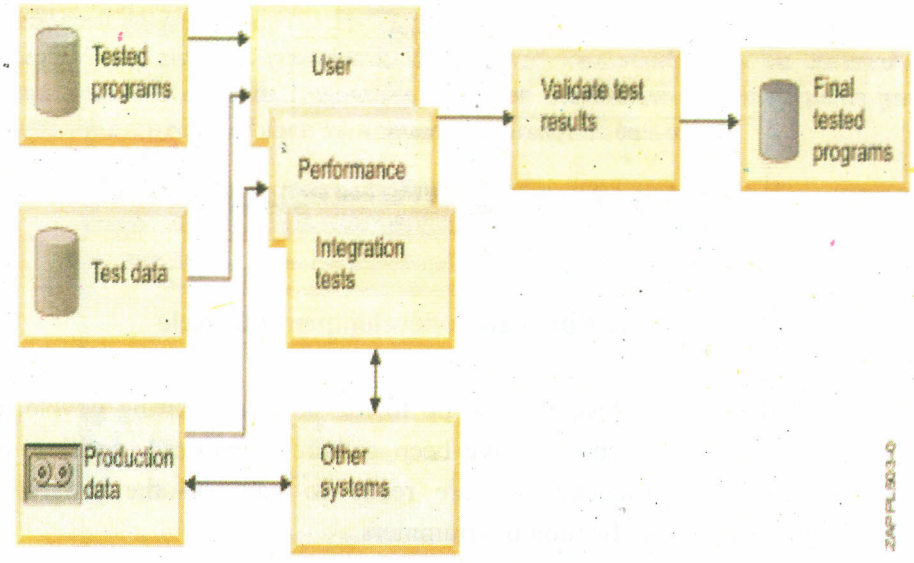


Fig. 4: Testing

The final phase in the development life cycle is to go to production and become steady state. As a prerequisite to going to production, the development team needs to provide documentation. This usually consists of user training and operational procedures. The user training familiarizes the users with the new application. The operational procedures documentation enables Operations to take over responsibility for running the application on an ongoing basis.

In production, the changes and enhancements are handled by a group (possibly the same programming group) that performs the maintenance. At this point in the life cycle of the application, changes are tightly controlled and must be rigorously tested before being implemented into production (Figure. 5).

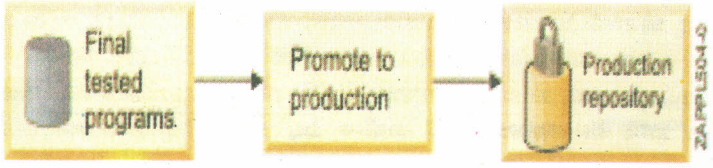


Fig. 5: Production

As mentioned before, to meet user requirements or solve problems, an application solution might be designed to reside on any platform or a combination of platforms. As shown in Figure. 6, our specific application can be located in any of the three environments: Internet, enterprise network, or central site. The operating system must provide access to any of these environments.

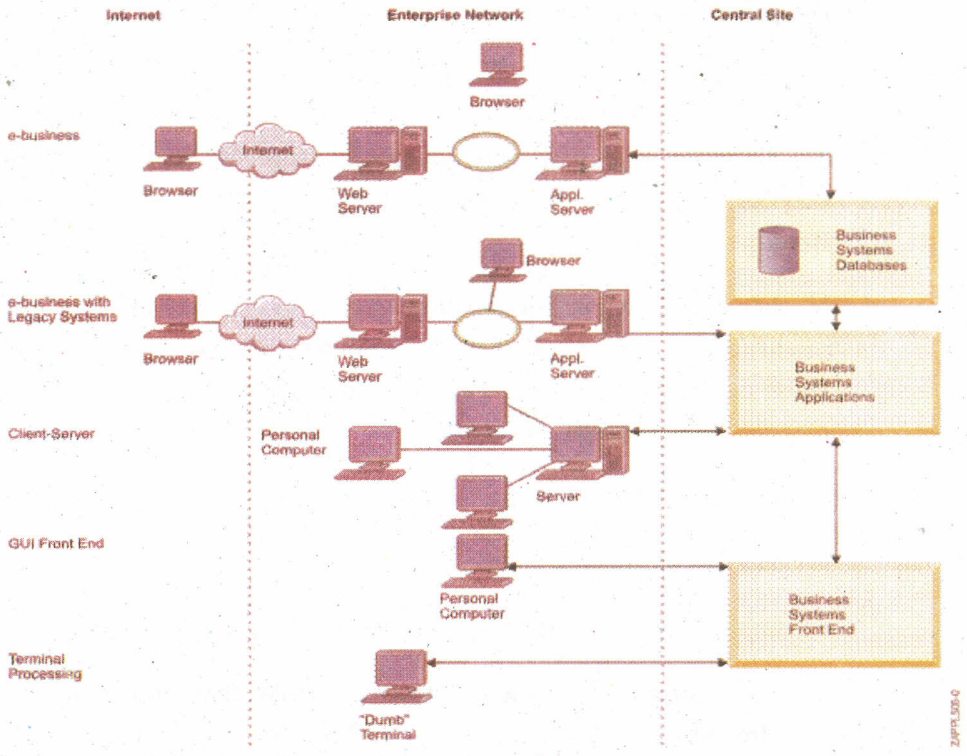


Fig. 6: Growing infrastructure complexity

To begin the design process, we must first assess what we need to accomplish. Based on the constraints of the project, we determine how and with what we will accomplish the goals of the project. To do so, we conduct interviews with the users (those requesting the solution to a problem) as well as the other stakeholders.

The results of these interviews should inform every subsequent stage of the life cycle of the application project. At certain stages of the project, we again call upon the users to verify that we have understood their requirements and that our solution meets their requirements. At these milestones of the project, we also ask the users to sign off on what we have done, so that we can proceed to the next step of the project.

4.2.2 Software Maintenance

Software maintenance is a controlled process that ensures that the software continues to meet the user needs. The steps are this process is:

- **Change** the software according to the change process for code:

  1. The users report problems and omissions in the software in Application Problem Reports.

     The URD may be modified in a later stage when the Software Review Board approves a new user requirement.

2. **Diagnose** the problems.
3. **Review** the resulting Software Change Requests. The Software Review Board decides, but for non-critical problems the project manager may decide. In the case of very urgent critical problems the project manager may take a decision that has to be confirmed by the Software Review Board.
4. **Modify** the software. The developers have to evaluate the effect of the modification on performance, resource consumption, cohesion, coupling, complexity, consistency, portability, reliability, maintainability, safety, security, ...
5. **Verify** the software:
   - detailed design and code,
   - unit review /integration/system test each change,
   - Regression test (rerun old tests to verify that there are no side effects).

- **Release** the software:

1. **Define** the release (contents and type):

Release type	Adaptive changes	Perfective changes	Corrective changes	Configuration Items included	Users that will receive this release
major	yes	yes	yes	all	all
minor	small	yes	yes	all	all
emergency	no	no	yes	selected	selected

2. **Document** the release. The release is documented in Application Release Notes that list the changes in this release.
3. **Audit** the release (physical and functional).
4. **Deliver** the release to the users. The responsibility for retaining a copy of every release lies with the maintenance team.
5. **Install** the release.
6. **Validate** the release: users run acceptance tests.

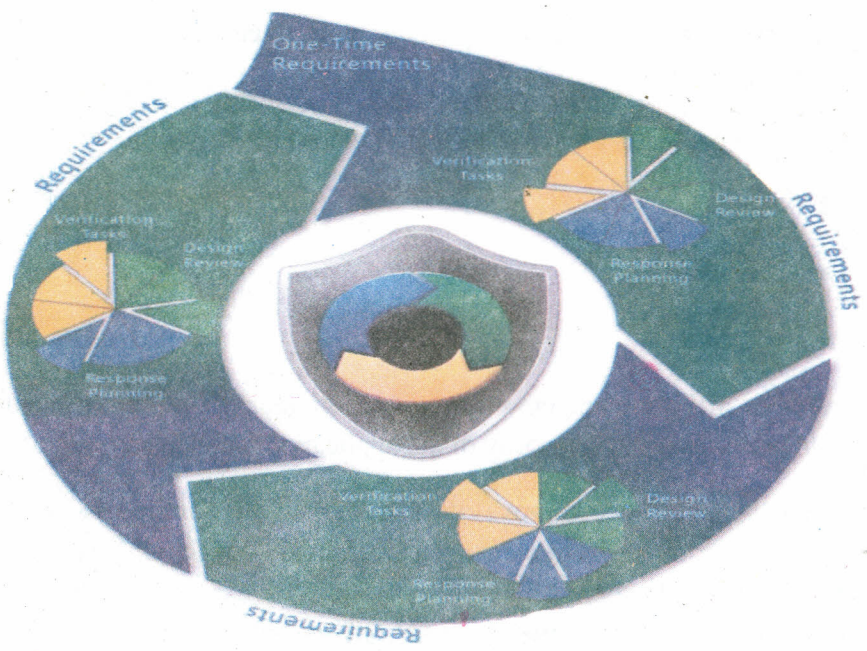


Fig. 7

4.2.3 Two Phases of Application Development Life Cycle

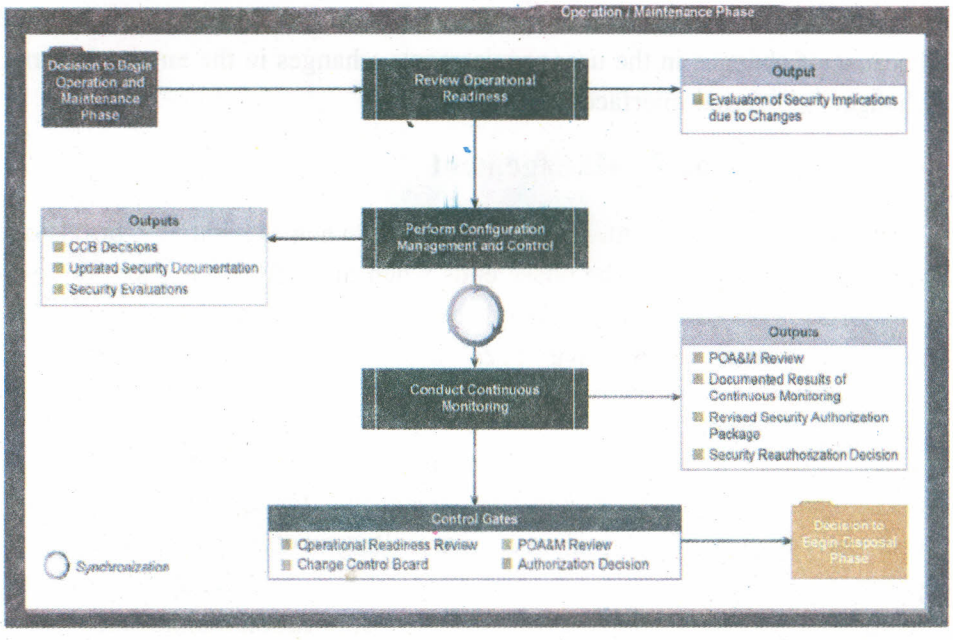


Fig. 8: Synchronization in Application Development Life Cycle

There are mainly two Phases for Application:

Software maintenance is 'the process of modifying software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment'. Maintenance is always necessary to keep software usable and useful. Often there are very tight constraints on changing software and optimum solutions can be difficult to find. Such constraints make maintenance a challenge; contrast the development

phase, when designers have considerably more freedom in the type of solution they can adopt.

Application maintenance activities can be classified as:

- corrective;
- perfective;
- adaptive.

Corrective maintenance removes application faults. Corrective maintenance should be the overriding priority of the application maintenance team.

Perfective maintenance improves the system without changing its functionality. The objective of perfective maintenance should be to prevent failures and optimize the application. This might be done, for example, by Modifying the components that have the highest failure rate, or components whose performance can be cost-effectively improved.

Adaptive maintenance modifies the application to keep it up to date with its environment. Users, hardware platforms and other systems all makeup the environment of an application system. Adaptive maintenance may be needed because of changes in the user requirements, changes in the target platform, or changes in external interfaces.

4.2.4 Configuration Management

The operations and maintenance phase starts when the initiator provisionally accepts the application. The phase ends when the software is taken out of use. The phase is divided into two periods by the final acceptance milestone. All the acceptance tests must have been successfully completed before final acceptance.

There must be a maintenance organization for every application Product in operational use. The developers are responsible for application maintenance and user support until final acceptance. Responsibility for these activities passes to a maintenance team upon final acceptance. The application project manager leads the developers. Similarly, the 'application maintenance manager' leads the maintenance team. Application project managers and application maintenance managers are collectively called 'application managers'.

The mandatory inputs to the operations and maintenance phase are the provisionally accepted application system, the statement of provisional acceptance and the application Transfer Document (STD). Some of the plans from the development phases may also be input.

The developer writes the Project History Document (PHD) during the warranty period. This gives an overview of the whole project and a summary account of the problems and performance of the application during the warranty period.

This document should be input to the application Review Board (SRB) that recommends about final acceptance. The PHD is delivered to the initiator at final acceptance.

Before final acceptance, the activities of the development team are controlled by the SPMP/TR. The development team will normally continue to use the SCMP, SVVP and SQAP from the earlier phases. After final acceptance, the maintenance team works according to its own plans. The new plans may reuse plans that the development team made, if appropriate. The maintenance team may have a different configuration management system, for example, but continue to employ the test tools and test application used by the developers.

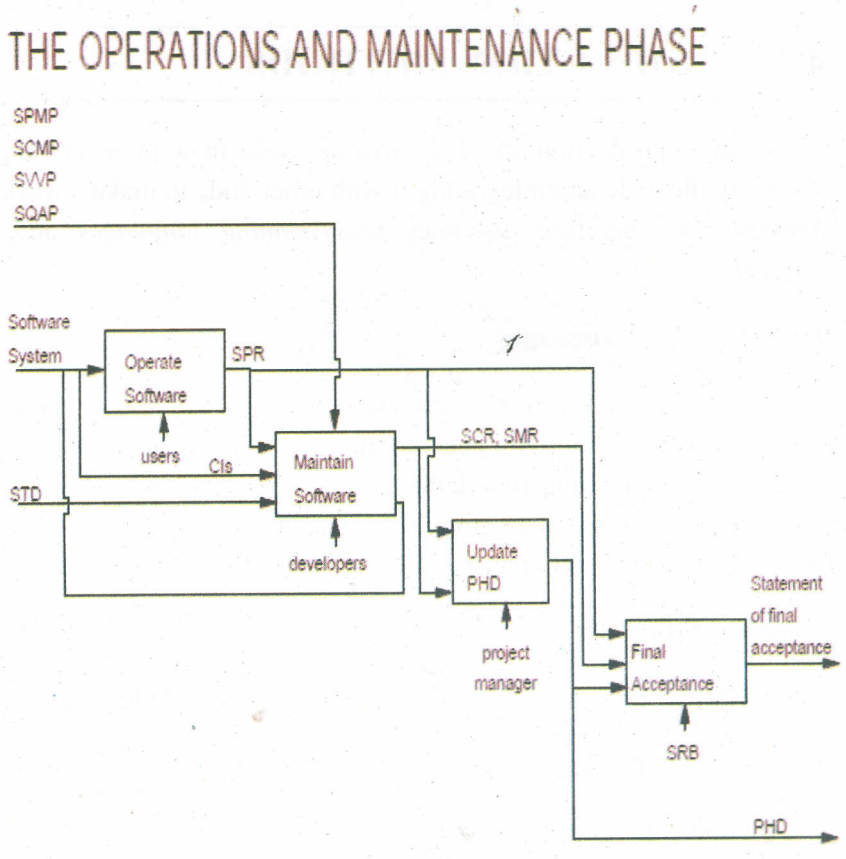


Fig. 9: OM Phase Activities before Final Acceptance

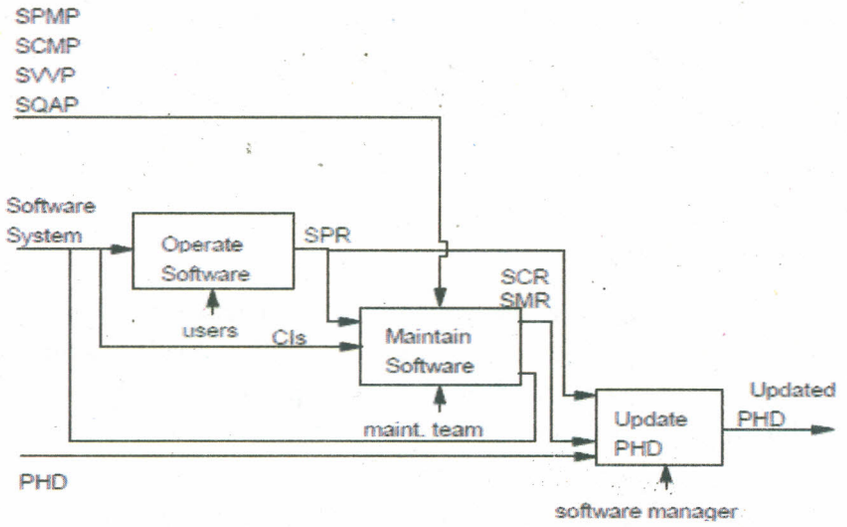


Fig. 10: OM Phase Activities before Final Acceptance

e

4.3 PRODUCTION METHODS

Application production involves writing code in a programming language, verifying the code and integrating it with other code to make a working system. This section therefore discusses *programming languages* and integration methods.

Programming Languages

Programming languages are best classified by their features and application domains. Classification by 'generation' (e.g. 3GL, 4GL) can be very misleading because the generation of a language can be completely unrelated to its age (e.g. Ada, LISP). Even so, study of the history of programming languages can give useful insights into the applicability and features of particular languages.

Classification

The following classes of programming languages are widely recognized:

- procedural languages;
- object-oriented languages;
- functional languages;
- logic programming languages.

Application-specific languages based on database management systems are not discussed here because of their lack of generality. Control languages, such as those used to command operating systems, are also not discussed for similar reasons.

Procedural languages are sometimes called 'imperative languages' or 'algorithmic languages'. Functional and logic programming languages are often collectively called 'declarative languages' because they allow programmers to declare 'what' is to be done rather than 'how'.

Procedural Languages

A procedural language should support the following features:

- sequence (composition);
- selection (alternation);
- iteration;
- division into modules.

The traditional procedural languages such as COBOL and FORTRAN support these features. The sequence constructs, also known as the composition construct, allows programmers to specify the order of execution. This is trivially done by placing one statement after another, but can imply the ability to branch (e.g. GOTO).

The sequence construct is used to express the dependencies between operations. Statements that come later in the sequence depend on the results of previous statements. The sequence construct is the most important feature of procedural languages, because the program logic is embedded in the sequence of operations, instead of in a data model (e.g. the trees of Prolog, the lists of LISP and the tables of RDBMS languages).

The selection constructs, also known as the condition or alternation Construct, allows programmers to evaluate a condition and take appropriate action (e.g. IF... THEN and CASE statements).

The iteration construct allows programmers to construct loops (e.g. DO...). This saves repetition of instructions.

The module construct allows programmers to identify a group of Instructions and utilize them elsewhere (e.g. CALL...). It saves repetition of Instructions and permits hierarchical decomposition.

Some procedural languages also support:

- block structuring;
- strong typing;
- recursion.

Block structuring enforces the structured programming principle that modules should have only one entry point and one exit point. Pascal, Ada and C support block structuring.

Strong typing requires the data type of each data object to be declared. This stops operators being applied to inappropriate data Objects and the interaction of data objects of incompatible data types (e.g. when the data type of a calling argument does not match the data type of a called argument). Ada and Pascal are strongly typed languages. Strong typing helps a compiler to find errors and to compile efficiently.

Recursion allows a module to call itself (e.g. module a calls module A), permitting greater economy in programming. Pascal, Ada and C support recursion.

4.4 MAINTENANCE

Software Maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes.

A common perception of maintenance is that it merely involves fixing defects. However, one study indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

Software maintenance and evolution of systems was first addressed by Meir M. Lehman in 1969. Over a period of twenty years, his research led to the formulation of Lehman's Laws (Lehman 1997). Key findings of his research include that maintenance is really evolutionary development and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Lehman demonstrated that systems continue to evolve over time. As they evolve, they grow more complex unless some action such as code refactoring is taken to reduce the complexity.

The key software maintenance issues are both managerial and technical. Key management issues are: alignment with customer priorities, staffing, which organization does maintenance, estimating costs. Key technical issues are: limited understanding, impact analysis, testing, maintainability measurement.

In the late 1970s, a famous and widely cited survey study by Lientz and Swanson, exposed the very high fraction of life-cycle costs that were being expended on maintenance. They categorized maintenance activities into four classes:

Adaptive – dealing with changes and adapting in the software environment

Perfective – accommodating with new or changed user requirements which concern functional enhancements to the software

Corrective – dealing with errors found and fixing it

Preventive – concerns activities aiming on increasing software maintainability and prevent problems in the future

The survey showed that around 75% of the maintenance effort was on the first two types, and error correction consumed about 21%. Many subsequent studies suggest a similar magnitude of the problem. Studies show that contribution of end user is crucial during the new requirement data gathering and analysis. And this is the main cause of any problem during software evolution and maintenance. So software maintenance is important because it consumes a large part of the overall lifecycle costs and also the inability to change software quickly and reliably means that business opportunities are lost.

Maintaining Software during Development

Software maintenance planning

The integral part of software is the maintenance part which requires accurate maintenance plan to be prepared during software development and should specify how users will request modifications or report problems and the estimation of resources such as cost should be included in the budget and a new decision should address to develop a new system and its quality objectives .The software maintenance which can last for 5–6 years after the development calls for an effective planning which addresses the scope of software maintenance, the tailoring of the post delivery process, the designation of who will provide maintenance, an estimate of the life-cycle costs.

Software maintenance processes

The six software maintenance processes are as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.

2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and

propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.

3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

There are a number of processes, activities and practices that are unique to maintainers, for example:

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;
- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;
- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

Categories of maintenance in ISO/IEC 14764

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective. These have since been updated and ISO/IEC 14764 presents:

Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems.

Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.

Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability.

Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

There is also a notion of pre-delivery/pre-release maintenance which is all the good things you do to lower the total cost of ownership of the software. Things like compliance with coding standards that includes software maintainability goals. The management of coupling and cohesion of the software. The attainment of software supportability goals (SAE JA1004, JA1005 and JA1006 for example). Note also that some academic institutions are carrying out research to quantify the cost to ongoing software maintenance due to the lack of resources such as design documents and system/software comprehension training and resources (multiply costs by approx. 1.5-2.0 where there is no design data available.).

Configuration Management

Configuration management was first developed by the United States Air Force for the Department of Defense in the 1950s as a technical management discipline of hardware. The concepts of this discipline have been widely adopted by numerous technical management functions, including systems engineering (SE), integrated logistics support (ILS), Capability Maturity Model Integration (CMMI), ISO 9000, Prince2 project management methodology, COBIT, Information Technology Infrastructure Library (ITIL), product lifecycle management, and application lifecycle management. Many of these functions and models have redefined configuration management from its traditional holistic approach to technical management. Some treat configuration management as being similar to a librarian activity, and break out change control or change management as a separate or stand alone discipline. However the bottomline is and always shall be Traceability.

Software configuration management

The traditional software configuration management (SCM) process is looked upon by practitioners as the best solution to handling changes in software projects. It identifies the functional and physical attributes of software at various points in time, and performs systematic control of changes to the identified attributes for the purpose of maintaining software integrity and traceability throughout the software development life cycle.

The SCM process further defines the need to trace changes, and the ability to verify that the final delivered software has all of the planned enhancements that are supposed to be included in the release. It identifies four procedures that

must be defined for each software project to ensure that a sound SCM process is implemented. They are:

- Configuration identification
- Configuration control
- Configuration status accounting
- Configuration audits

These terms and definitions change from standard to standard, but are essentially the same.

Configuration identification is the process of identifying the attributes that define every aspect of a configuration item. A configuration item is a product (hardware and/or software) that has an end-user purpose. These attributes are recorded in configuration documentation and baselined. Baselining an attribute forces formal configuration change control processes to be effected in the event that these attributes are changed.

Configuration change control is a set of processes and approval stages required to change a configuration item's attributes and to re-baseline them.

Configuration status accounting is the ability to record and report on the configuration baselines associated with each configuration item at any moment of time.

Configuration audits are broken into functional and physical configuration audits. They occur either at delivery or at the moment of effecting the change. A functional configuration audit ensures that functional and performance attributes of a configuration item are achieved, while a physical configuration audit ensures that a configuration item is installed in accordance with the requirements of its detailed design documentation.

Configuration management is widely used by many military organizations to manage the technical aspects of any complex systems, such as weapon systems, vehicles, and information systems. The discipline combines the capability aspects that these systems provide an organization with the issues of management of change to these systems over time.

Outside of the military, CM is appropriate to a wide range of fields and industry and commercial sectors.

Computer hardware configuration management

Computer hardware configuration management is the process of creating and maintaining an up-to-date record of all the components of the infrastructure, including related documentation. Its purpose is to show what makes up the

infrastructure and illustrate the physical locations and links between each item, which are known as configuration items.

Computer hardware configuration goes beyond the recording of computer hardware for the purpose of asset management, although it can be used to maintain asset information. The extra value provided is the rich source of support information that it provides to all interested parties. This information is typically stored together in a configuration management database (CMDB). This concept was introduced by ITIL.

The scope of configuration management is assumed to include, at a minimum, all configuration items used in the provision of live, operational services.

Computer hardware configuration management provides direct control over information technology (IT) assets and improves the ability of the service provider to deliver quality IT services in an economical and effective manner. Configuration management should work closely with change management.

All components of the IT infrastructure should be registered in the CMDB. The responsibilities of configuration management with regard to the CMDB are:

- identification
- control
- status accounting
- verification

The scope of configuration management is assumed to include:

- physical client and server hardware products and versions
- operating system software products and versions
- application development software products and versions
- technical architecture product sets and versions as they are defined and introduced
- live documentation
- networking products and versions
- live application products and versions
- definitions of packages of software releases
- definitions of hardware base configurations
- configuration item standards and definitions

The benefits of computer hardware configuration management are:

- helps to minimize the impact of changes
- provides accurate information on CIs (Configuration Items)
- improves security by controlling the versions of CIs (Configuration Items) in use
- facilitates adherence to legal obligations
- helps in financial and expenditure planning

Check Your Progress 1

Note: a) Space is given below for writing your answer.

b) Compare your answer with the one given at the end of the Unit.

1) What are Application maintenance activities?

.....

.....

.....

.....

2) What are features of procedural languages?

.....

.....

.....

.....

4.5 LET US SUM UP

As products age it becomes more difficult to keep them updated with new user requirements. Maintenance costs developers time, effort, and money. This requires that the maintenance phase be as efficient as possible. There are several steps in the software maintenance phase. The first is to try to understand the design that already exists. The next step of maintenance is reverse engineering in which the design of the product is reexamined and restructured. The final step is to test and debug the product to make the new changes work properly.

This unit will discuss what maintenance is, its role in the software development process, how it is carried out, and its role in iterative development, agile development, component-based development and open source development.

The modification of a software product after delivery is required to correct faults, to improve performance or other attributes or to adapt the product to a modified environment.

4.6 CHECK YOUR PROGRESS: THE KEY

Check Your Progress 1

1) Application maintenance activities can be classified as:

- corrective;
- perfective;
- adaptive.

2) A procedural language should support the following features:

- sequence (composition);
- selection (alternation);
- iteration;
- division into modules.

4.7 SUGGESTED READINGS

- Bijlsma, B.J.; Heeren, E.E. and Roubtsova, S. Stuurman (2011). *Software Architecture.*
- http://mail.svce.ac.in/~uvarajan/softeng/outline25.html
- J. Pérez López and L. Ribas i Xirgo (2010*). Introduction to Software Development.*
- Jindal,Gaurav and Bakshi,Arun (2010). *Object Oriented Software Engineering,* Haranand Publications, Delhi.
- Kim Walden. *Seamless Object-Oriented Software Architecture - Analysis And Design of Reliable Systems*
- Pressman, Roger S. (2005). *Software Engineering: A Practitioner's Approach,* McGraw-Hill.

Student Satisfaction Survey

Student Satisfaction Survey of IGNOU Students

ignou THE PEOPLE'S UNIVERSITY

Enrollment No.	
Mobile No.	
Name	
Programme of Study	
Year of Enrolment	
Age Group	☐ Below 30 ☐ 31-40 ☐ 41-50 ☐ 51 and above
Gender	☐ Male ☐ Female
Regional Centre	
States	
Study Center Code	

Please indicate how much you are satisfied or dissatisfied with the following statements

Sl. No.	Questions	Very Satisfied	Satisfied	Average	Dissatisfied	Very Dissatisfied
1.	Concepts are clearly explained in the printed learning material	☐	☐	☐	☐	☐
2.	The learning materials were received in time	☐	☐	☐	☐	☐
3.	Supplementary study materials (like video/audio) available	☐	☐	☐	☐	☐
4.	Academic counselors explain the concepts clearly	☐	☐	☐	☐	☐
5.	The counseling sessions were interactive	☐	☐	☐	☐	☐
6.	Changes in the counseling schedule were communicated to you on time	☐	☐	☐	☐	☐
7.	Examination procedures were clearly given to you	☐	☐	☐	☐	☐
8.	Personnel in the study centers are helpful	☐	☐	☐	☐	☐
9.	Academic counseling sessions are well organized	☐	☐	☐	☐	☐
10.	Studying the programme/course provide the knowledge of the subject	☐	☐	☐	☐	☐
11.	Assignments are returned in time	☐	☐	☐	☐	☐
12.	Feedbacks on the assignments helped in clarifying the concepts	☐	☐	☐	☐	☐
13.	Project proposals are clearly marked and discussed	☐	☐	☐	☐	☐
14.	Results and grade card of the examination were provided on time	☐	☐	☐	☐	☐
15.	Overall, I am satisfied with the programme	☐	☐	☐	☐	☐
16.	Guidance from the programme coordinator and teachers from the school	☐	☐	☐	☐	☐

After filling this questionnaire send it to:
Programme Coordinator, School of Vocational Education and Training,
Room no. 19, Block no. 1, IGNOU, Maidangarhi, New Delhi- 110068

MPDD-IGNOU/P.O.1T/Feb,2012

ISBN-978-81-266-5889-3