African Virtual University

Applied Computer Science: CSI 4105

SOFTWARE ENGINEERING

Dr. Ellen Ambakisye Kalinga

Foreword

The African Virtual University (AVU) is proud to participate in increasing access to education in African countries through the production of quality learning materials. We are also proud to contribute to global knowledge as our Open Educational Resources are mostly accessed from outside the African continent.

This module was developed as part of a diploma and degree program in Applied Computer Science, in collaboration with 18 African partner institutions from 16 countries. A total of 156 modules were developed or translated to ensure availability in English, French and Portuguese. These modules have also been made available as open education resources (OER) on oer.avu.org.

On behalf of the African Virtual University and our patron, our partner institutions, the African Development Bank, I invite you to use this module in your institution, for your own education, to share it as widely as possible and to participate actively in the AVU communities of practice of your interest. We are committed to be on the frontline of developing and sharing Open Educational Resources.

The African Virtual University (AVU) is a Pan African Intergovernmental Organization established by charter with the mandate of significantly increasing access to quality higher education and training through the innovative use of information communication technologies. A Charter, establishing the AVU as an Intergovernmental Organization, has been signed so far by nineteen (19) African Governments - Kenya, Senegal, Mauritania, Mali, Cote d'Ivoire, Tanzania, Mozambique, Democratic Republic of Congo, Benin, Ghana, Republic of Guinea, Burkina Faso, Niger, South Sudan, Sudan, The Gambia, Guinea-Bissau, Ethiopia and Cape Verde.

The following institutions participated in the Applied Computer Science Program: (1) Université d'Abomey Calavi in Benin; (2) Université de Ougagadougou in Burkina Faso; (3) Université Lumière de Bujumbura in Burundi; (4) Université de Douala in Cameroon; (5) Université de Nouakchott in Mauritania; (6) Université Gaston Berger in Senegal; (7) Université des Sciences, des Techniques et Technologies de Bamako in Mali (8) Ghana Institute of Management and Public Administration; (9) Kwame Nkrumah University of Science and Technology in Ghana; (10) Kenyatta University in Kenya; (11) Egerton University in Kenya; (12) Addis Ababa University in Ethiopia (13) University of Rwanda; (14) University of Dar es Salaam in Tanzania; (15) Universite Abdou Moumouni de Niamey in Niger; (16) Université Cheikh Anta Diop in Senegal; (17) Universidade Pedagógica in Mozambique; and (18) The University of the Gambia in The Gambia.

Bakary Diallo

The Rector

African Virtual University.

Production Credits

Author

Dr. Ellen Ambakisye Kalinga

Peer Reviewer

Robert Oboko

AVU - Academic Coordination

Dr. Marilena Cabral

Overall Coordinator Applied Computer Science Program

Prof Tim Mwololo Waema

Module Coordinator

Robert Oboko

Instructional Designers

Elizabeth Mbasu

Diana Tuel

Benta Ochola

Media Team

Sidney McGregor

Barry Savala

Edwin Kiprono

Kelvin Muriithi

Victor Oluoch Otieno

Michal Abigael Koyier

Mercy Tabi Ojwang

Josiah Mutsogu

Kefa Murimi

Gerisson Mulongo

Copyright Notice

Supported By

AVU Multinational Project II funded by the African Development Bank.

Table of Contents

Course Overview

Welcome to Software Engineering

Software engineering is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use (Laurie Williams 2004). To engineering perspective, SE adopts a systematic and organized approach to develop systems.

The course exposes students on how to use several specific practices and techniques used in developing software. It is required to familiarize learners to industrial modeling tools with the aim of exposing them to state-of-the-art practices with respect to software development. At the end of the module , student will have a better understanding of the complexities as well as subtleties of the various software development activities that include working in a team or group. Students will learn process models, software testing techniques and project management skills used in building software effectively, apply good practices, effective design techniques, and development tools. SE is needed to develop all kinds of software projects including complex software projects.

Prerequisites

- Introduction to database systems
- Introduction to structured programming

Material

The materials required to complete this course are:

- Text books
- Lecture notes
- Software tools
- Online Videos
- Wiki pages / Forum discussion

Course Goals

Upon completion of this course the learner should be able to:

- Extract, analyze and specify software requirements through a productive working relationship with various stakeholders and communicate effectively in an ethical manner as software engineering professionals.

- Use the concepts of computer science and engineering principles to develop software projects
- Apply proven and best practices to develop a quality software within time and budgets
- Design, develop, and verify system and application software in industrial, business, and personal applications.

Units

Unit 0: Pre-Assessment

This unit requires you to have an introductory part for this module. It reminds the basic knowledge which will be used in Software Engineering module. This includes the basic knowledge of what is a software system, who are participants in software system development computer programming, the software development cycle and the cycle for data processing with principles of structured programming.

Unit 1: Fundamentals of Software Engineering

The topic intends to introduce and explain the importance of SE in software development. It describes problems faced software development and provides ethical and professional issues needed in software engineering discipline. Software development needs a systematic process. This part also explains a series of steps (a roadmap) that helps to create quality software product. It also describes generic process models.

Unit 2: Planning a Software Project and Software Requirements Analysis and Specification

This unit introduces the bases for obtaining the software requirement specification by employing requirement engineering mechanisms. It explains how to understand what customers want, analyze and validate their requirements and finally produce requirement specification.

The unit also elaborates a set of activities related to software project planning. Software project planning touches the cost estimation of the project and the project scheduling.

Unit 3: Software Design

The purpose of the software design is to produce a model or a representation of an entity that will later be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. This unit describes design objectives and design principles. It elaborates design software processes which include: Architectural Design, Abstract Design, User-Interface Design, Low-Level Design. Function-oriented design, object-oriented design, design notation and specification and verification for design were also presented.

Unit 4: Implementation and Testing

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. It is the assurance of the responds to the specifications from the design phase to implementation. This unit addresses in general all aspects related to the implementation and testing of software. It describes approached techniques used to implement and the mechanisms used to perform software testing developed. It also presents the explored techniques in the whole process.

Unit 5: Maintenance and Project Management

This unit will focus on all aspects related to the management and maintenance of software. The unit explains why software maintenance is needed. It also explains categories of maintenance, an overview of maintenance costs and factors affecting maintenance. Lastly, the unit touches issues of software risk analysis and management.

Assessment

Formative assessments, used to check learner progress, are included in each unit. Summative assessments, such as final tests and assignments, are provided at the end of each module and cover knowledge and skills from the entire module. Summative assessments are administered at the discretion of the institution offering the course. The suggested assessment plan is as follows:

20 marks	Consultation of materials and resources
40 marks	Practical (Hands-on) exercises
40 marks	Formative and summative assessments

Schedule

Estimated time	Activities	Unit
1 Weeks (8 Hrs)	Introduction to Software System	Unit 0
3 Weeks (24 Hrs)	Fundamentals of Software Engineering	Unit I
3 Weeks (24 Hrs)	Planning a software project and Software requirements analysis and specification	Unit II
3 Weeks (24 Hrs)	Software Design	Unit III
3 Weeks (24 Hrs)	Implementation and Testing	Unit IV
3 Weeks (24 Hrs)	Maintenance and Project Management	Unit V
15 Weeks (120 Hours)	Total	

Readings and Other Resources

The readings and other resources in this course are:

Unit 0

Required readings and other resources:

Tran Thi Phien, (2006), "System Analysis and Design", Institute of Information Technology

Vic Broquard, (2006), "C++ for Computer Science and Engineering, A step-by-Step Approach to Learning C++ Programming for Beginners", 4th Edition, Broquard eBooks, ISBN: 0-9705697-2-6

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Unit 1

Required readings and other resources:

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783.

Optional readings and other resources:

Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.

Sommerville I., (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Unit 2

Required readings and other resources:

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition,

McGraw-Hill Higher Education, ISBN 0073655783

Optional readings and other resources: Sommerville I., (2000), "Software Engineering (6th

Edition)". Addison-Wesley, Boston USA

Unit 3

Required readings and other resources:

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Optional readings and other resources:

Sommerville I., (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Unit 4

Required readings and other resources:

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783.

Optional readings and other resources:

Sommerville I., (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Abran A. and Moore J. W., (2004), "IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK)",

David Gustafson (2002), "Theory and Problems of Software Engineering", Schaum's Outline Series, McGraw-Hill, 0-07-140620-4

Unit 5

Required readings and other resources:

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Optional readings and other resources

Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.

Sommerville I., (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Abran A. and Moore J. W., (2004), "IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK)",

Gustafson D., (2002), "Theory and Problems of Software Engineering", Schaum's Outline Series, McGraw-Hill, 0-07-140620-4

Unit 0. Introduction to Software System

Introduction

When studying Software Engineering, there is a number of preliminary basic knowledge a student/learner need to have. This unit reviews some of this basic knowledge. The unit starts by reviewing system analysis and design. It describes the general overview of the computer systems, types of systems, roles of users participating in system development life-cycle. The unit also explains basics in programming, which include: cycle of data processing, steps needed to create a program and structured programming. The purpose of this unit is to determine your grasp of knowledge related to this course.

Objectives

Upon completion of this unit you should be able to:

- Define computer system and Describe types of systems
- Identify major players or participants in the software development
- Practice IPO data processing
- Elaborate steps in program creation
- Use structured programming in solving real problems
- Elaborate steps in program creation
- Use structured programming in solving real problems

KEY TERMS

A System:

A system is a collection of components that work together to realize some objective forms of a system.

IPO:

This is Input, Process and Output (IPO), the most basic design cycle of data processing of a program.

Learning Activities

Activity 1 – System Analysis and Design

Computer Systems

Systems are created to solve problems. One can think of the systems approach as an organized way of dealing with a problem. In this dynamic world, the subject System Analysis and Design (SAD), mainly deals with the software development activities.

What is a System

From Tran Thi Phien, (2006), A system can generally be defined as a collection of components that work together to realize some objective forms of a system. A system may include software, mechanical, electrical and electronic hardware and be operated by people. Basically there are three major components in every system, namely input, processing and output. In a system the different components are connected with each other and they are interdependent, for example:

- Human body represents a complete natural system
- We are also bound by many national systems such as political system, economic system, educational system and so forth
- The objectives of the system demand that some output is produced as a result of processing the suitable inputs.

Types of Systems

As stated by Tran Thi Phien, (2006), there are many types of systems that we come into contact with everyday. The one we are interested with is an automated, computerized information system. Automated systems are the man-made systems that interact with or are controlled by one or more computers. We can distinguish many different kinds of automated systems, but they all tend to have five basic components:

Infrastructure: The physical and hardware system components, e.g. servers, computer hardware: CPUs, disks, terminals, etc

Computer software: The programs and operating software of a system, including operating systems, database systems, utilities, and applications (financial systems)

People: to operate the system, to provide its inputs and consume its outputs, and to provide manual processing activities in a system. E.g. programmers, operators, users of the systems and management

Data: The information captured, used, and supported by a system, including files and databases. The information that the system remembers over a period of time.

Procedures: The programmed and manual guidelines, instructions, and steps involved in operating systems, including information technology (IT) procedures for backup and maintenance. Formal policies and instructions for operating the system.

Business systems use these system components to transform data inputs into information outputs.

Participants to System Development

In a typical systems development project, there are the following major categories of players (Tran Thi Phien, (2006)):

User

The most important player in the systems is the person (or group of people) for whom the system is being built. He or she is the person whom will be interviewed, often in great detail, to learn what features the new system must have to be successful. The user is the "owner" in the sense that he or she receives, or inherits-and thus owns- the system when it is finally built. The user is also the "customer" in at least two important respects:

As in so many other professions, "the customer is always right", regardless of how demanding, unpleasant, or irrational he or she may seem.

The customer is ultimately the person paying for the system and usually has the right and/or the ability to refuse to pay if he or she is unhappy with the product received.

Management

Management is a rather loose term. There are several different kinds of managers:

User managers: managers in charge of several people in the operational area where the new system will be used. These are usually middle-level managers who want systems that will produce a variety of internal reports and short-term trend analyses.

Executive development project (EDP)/MIS managers: the person in charge of the systems development project itself, and the higher-level managers who are concerned with the overall management and allocation of resources of all the technical staff in the systems development organization.

General management: top-level managers who are not directly involved in the EDP organization or in the user organization. This might include the president and/ or chairman of the organization

Systems analysts

The system analyst is a key member of any systems development project. In a boarder sense, the systems analyst plays several roles:

Archaeologist and scribe: As a systems analyst, one of the main jobs is to uncover detail and to document business policy that may exist only as "tribal tradition", passed down from generation to generation of users.

Innovator: The systems analyst must separate the symptoms of the user's problem from the true causes. With his or her knowledge of computer technology, the analyst must help the user explore useful, new applications of computers.

Mediator: The systems analyst who often finds himself in the middle of users, managers, programmers, auditors, and various other players, all of whom frequently disagree with one another.

Project leader: Because the systems analyst is usually more experienced than the programmers on the project, and since he is assigned to the project before the programmers begin working, there is a natural tendency to assign project management responsibilities to the analyst.

Systems designers

The systems designer is the person (or group of people) who will receive the output of the systems analysis work. His or her job is to transform a technology-free statement of user requirements into a high-level architectural design that will provide the framework within which the programmer can work. In many case, the systems analyst and the systems designer are the same person, or member of the same unified group of people. It is important for the systems analyst and systems designer to stay in close touch throughout the project.

Programmers

Particularly on large systems development projects, the systems designers are likely to be a "buffer" between the systems analysts and the programmers. The systems analysts deliver their product to the system designers, and the system designers deliver their product to the programmer. Systems analyst and the programmer may have little or no contact with each other because work is often performed in a strictly serial sequence in many systems development projects. The work of systems analysis takes place first and it has to be completely finished before the work of programming begins.

Operations personnel

The operations personnel are responsible for the computer center, telecommunications network, security of the computer hardware and data, as well as the actual running of computer programs, mounting of disk packs, and handling of output from computer printers. This happens after a new system has not only been analyzed and designed, but has also been programmed and tested.

Management

Management is a rather loose term. There are several different kinds of managers:

User managers: managers in charge of several people in the operational area where the new system will be used. These are usually middle-level managers who want systems that will produce a variety of internal reports and short-term trend analyses.

Executive development project (EDP)/MIS managers: the person in charge of the systems development project itself, and the higher-level managers who are concerned with the overall management and allocation of resources of all the technical staff in the systems development organization.

General management: top-level managers who are not directly involved in the EDP organization or in the user organization. This might include the president and/ or chairman of

the organization

Systems analysts

The system analyst is a key member of any systems development project. In a boarder sense, the systems analyst plays several roles:

Archaeologist and scribe: As a systems analyst, one of the main jobs is to uncover detail and to document business policy that may exist only as "tribal tradition", passed down from generation to generation of users.

Innovator: The systems analyst must separate the symptoms of the user's problem from the true causes. With his or her knowledge of computer technology, the analyst must help the user explore useful, new applications of computers.

Mediator: The systems analyst who often finds himself in the middle of users, managers, programmers, auditors, and various other players, all of whom frequently disagree with one another.

Project leader: Because the systems analyst is usually more experienced than the programmers on the project, and since he is assigned to the project before the programmers begin working, there is a natural tendency to assign project management responsibilities to the analyst.

Systems designers

The systems designer is the person (or group of people) who will receive the output of the systems analysis work. His or her job is to transform a technology-free statement of user requirements into a high-level architectural design that will provide the framework within which the programmer can work. In many case, the systems analyst and the systems designer are the same person, or member of the same unified group of people. It is important for the systems analyst and systems designer to stay in close touch throughout the project.

Programmers

Particularly on large systems development projects, the systems designers are likely to be a "buffer" between the systems analysts and the programmers. The systems analysts deliver their product to the system designers, and the system designers deliver their product to the programmer. Systems analyst and the programmer may have little or no contact with each other because work is often performed in a strictly serial sequence in many systems development projects. The work of systems analysis takes place first and it has to be completely finished before the work of programming begins.

Operations personnel

The operations personnel are responsible for the computer center, telecommunications network, security of the computer hardware and data, as well as the actual running of computer programs, mounting of disk packs, and handling of output from computer printers. This happens after a new system has not only been analyzed and designed, but has also been programmed and tested.

Exercise

What is system, give some definitions of system

How do you distinguish natural systems and man-made systems

List some automated systems and the rules to build them up

Who participate in system development? Explain the role of each of them

Activity 2 - Basics in Programming

The Cycle of Data Processing

The most important aspect of solving a problem on the computer is the initial design phase in which one use paper and pencil to provide the precise steps the computer must take. Nearly every significant program follows the same fundamental design and

it is called the **Cycle of Data Processing**.The Cycle of Data Processing is Input, Process and Output. First the computer must input a set of data on which to work. Once the data has been input into the computer, it can then process that data, often performing some calculations on that information. When the calculations are finished, the computer outputs that set of data and the results. The Cycle of Data Processing is called IPO for short. IPO is the most basic design of a program. Thus, when you are confronting a computer problem to solve, IPO is the starting point! Input a set of information first. Then do the requisite processing steps using that information. Last, output the results.

The Steps Needed to Create a Program

The following steps represent an optimum procedure to follow to solve any problem on the computer (Broquard V., 2006)):

Step 1

Fully understand the problem to be solved. Begin by looking over the output, what the program is supposed to be producing, what are the results? Then look over the input that the program will be receiving. Finally, determine what general processing steps are going to be needed to turn that input into the required output. Part of this step of understanding the problem involves determining the algorithm to be used. An algorithm is a finite series of steps for solving a logical or mathematical problem.

Step 2

Design a solution using paper and pencil. This process involves two distinct activities.

The first action is to design what function(s) would best aid in the solution. Note these are functions that you must write, and not those that are provided by the compiler manufacturer e.g. sqrt.

The second action is crucial. Write out on paper the precise steps needed to solve the problem in the precise sequence. This is often called pseudocode. Pseudocode is done by using English and perhaps some programming language statements.

You are trying at this point to say in English the correct sequence of steps that must be followed to produce the result.

Step 3

Thoroughly desk checks the solution. Desk checks means to play computer and follow slavishly and precisely the steps written down in the solution. You are looking for errors at this point, i.e. the whole purpose of desk checking is to find all errors in the solution.

Step 4

Code the solution into the programming language. With the pseudo coding and memory drawings at hand, it becomes a fairly simple matter to convert the solution into a source program.

Step 5

Compile the program. If there are any errors found by the compiler, these are called syntax errors. Again a syntax error is just incorrect coding. Just fix up the mistyping and recompile. Once you have a clean compile and built the program (and have an executable file), go on to the next step.

Step 6

Test the program with one set of data. Try inputting one set of test data only. Examine the output and verify it is correct. An error here is called a runtime logic error. If the results are not correct, then you have missed something. It is back to Step 1 or 2 to figure out what was missed.

Step 7

Documentation: for the future use. Minimum documentation should include: Title of program, Statement of the problem (abstract), programmer's name, and date.

Structured Programming

A program is usually not limited to a linear sequence of instructions. During its process it may branch off, repeat code or take decisions. There are a number of situations, where the program may have to change the order of execution of statement based on certain conditions, or repeat a group of statement until certain specified conditions are met. This involve a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statement accordingly.

Structured programs have the single-entry, single-exit property. This feature helps in reducing the number of paths for flow of control. If there are arbitrary paths for the flow of control, the program will be difficult to read, understand, debug, and maintain.

A program is one of two types (Agarwal et al, 2010):

Static structure or Dynamic structure.

The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program. The dynamic structure of the program is the sequence of statements executed during the execution of the program.

Both static and dynamic structures are the sequence of statements. The only difference is that the sequence of statements in a static structure is fixed, whereas in a dynamic structure it is not fixed. That means the dynamic sequence of statements can change from execution to execution.

The static structure of a program can be easily understood. The dynamic structure of a program can be easily seen at the time of execution.

Exercise

Clearly describe the cycle of data processing in computer programming

In short explain the steps needed to create a program

Explain the importance of "algorithms" in programming

Describe the possible errors to be encountered when testing programs

Distinguish between "static structure" and "dynamic structure"

UMIT SUMMARY

There are several software development methodologies employed, one of them is the structured approach. Structured approach is the most widely used methodology. Techniques which are invariably present include entity relationship diagrams, dataflow diagrams and data dictionaries. Structured Systems Analysis and Design prescribe analyzing and designing software systems through functional decomposition – i.e. examining an information system in terms of the functions it performs and the data it uses and maintains. The analyst identifies the major functions or processes of a system, then breaks or decomposes each function down into its smaller composite steps.

Unit Assessment

Check your understanding

1. Describe five basic components of automated systems.

   (Answer: section 0.1.3)

2. Explain the role of the following roles in system development:

   (*Answer: section 0.1.4)*

   a) User

   b) System Designer

3. Describe the three basic structured control constructs.

   (Answer: section 0.2.4)

Unit Readings and Other Resources

Tran Thi Phien, (2006), "System Analysis and Design", Institute of Information Technology

Broquard V., (2006), "C++ for Computer Science and Engineering, A step-by-Step Approach to Learning C++ Programming for Beginners", 4th Edition, Broquard eBooks, ISBN: 0-9705697-2-6

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Unit I. Fundamentals of Software Engineering

Unit Introduction

The unit starts by explaining what software is and its types. It describes the crisis and myths of software development. The unit introduces the need of software engineering and Software development process or software development life cycle will be explained. Generic process models, Waterfall and evolutionary are included.

Unit Objectives

Upon completion of this unit you should be able to:

Describe the term software and its types

Define Software Engineering and explain why engineering is being employed in software development

Illustrate phases involved in software development process, i.e. Software development Life Cycle Describe generic process models of the software development

KEY TERMS

Software: Software is a set of instructions with all associated documentation and configuration data used to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software.

Software Engineering: Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches

Software Product:

Software products are software systems delivered to a customer with the documentation which describes how to install and use the system

SDLC: System development life cycle (SDLC) means combination of various activities. In other words we can say that various activities put together are referred as system development life cycle

Process: Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology

Learning Activities

Activity 1 - Software

Introduction

In this activity, the definition of software, types and classes of software, software crisis and software myths are presented.

What is Software

Software is not just the program but also with all associated documentation and configuration data which is needed to make these programs operate correctly. As defined by Agarwal et al,(2010), software is a set of instructions used to acquire inputs and to manipulate them to produce the desired output in terms of functions and performance as determined by the user of the software. It also includes a set of documents, such as the software manual, meant to help users understand the software system. Today's software is comprised of Source Code, Executables, Design Documents, Operations, and System Installation and Implementation Manuals. In other words a software system consists of:

Instructions (a number of separate computer programs) that when executed provide desired functions and performance

Configuration files which are used to set up these programs and data structures that enables the programs to adequately manipulate information

System documentation which describes the structure of the system and

User documentation which explains the use the programs and the system

Software Crisis

Why software projects are not delivering the promised products according to the specifications, on time, within budget and to the quality specified?

Software development was in crisis (disaster) because the methods (if there were any) used were not good enough:

Techniques applicable to small systems could not be scaled up

Major projects were sometimes years late, they cost much more that originally predicted

Software developed were unreliable, performed poorly and were difficult to maintain

During software development, many problems are raised and that set of problems is known as the software crisis. When software is being developed, problems are encountered associated with the development steps.

Problems

Problems encountered include:

- Schedule and cost estimates are often grossly inaccurate.
- The "productivity" of software people hasn't kept pace with the demand for their services.
- The quality of software is sometimes less than adequate.
- With no solid indication of productivity, we can't accurately evaluate the efficiency of new, tools, methods, or standards.
- Communication between the customer and software developer is often poor.
- Software maintenance tasks overwhelm the majority of all software funds.

Causes

Causes of the problems include:

- The quality of the software is not good because most developers use historical data to develop the software.
- If there is delay in any process or stage (i.e., analysis, design, coding & testing) then scheduling does not match with actual timing.
- Communication between managers and customers, software developers, support staff, etc., can break down because the special characteristics of software and the problems associated with its development are misunderstood.
- The software people responsible for tapping the potential often change when it is discussed and resist change when it is introduced.

Software Crisis from the Programmer's Point-of-View

From Programmers' point of view problems include: Problem of compatibility, Problem of portability, Problem in documentation, Problem of piracy of software, Problem in coordination of work of different people, Problem of proper maintenance.

of piracy of software, Problem in coordination of work of different people, Problem of proper maintenance.

Software Crisis from the User's Point-of-View

From the User's Point-of-View problems include: high costs of software, deterioration of Hardware, lack of specialization in development, problem of different versions of software, problem of views, and problem of bugs.

Software Myths

Many causes of a software difficulty can be traced to a mythology - Myths (tradition way) that arose during the early history of software development, some being (Pressman, 2001 and Agarwal et al, 2010):

- We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?
- "A general statement of objectives" is all I need to begin writing programs; we can fill in the details later.
- If we get behind schedule, we can add more programmers and catch up.
- I have got to begin coding because we are already late!
- Project requirement continuously changes, but changes can be easily accommodated because software is flexible.
- It is impossible to assess the quality of a program until after it is written and working.
- Any competent engineer can write programs.
- Let us run a few test cases and then we'll be finished!
- I know what the program does, I don't have time to document it. The only deliverable work product for a successful project is the working program.
- My people do have state-of-art software development tools. After all, we buy them the newest computers
- Working on programs that were written 30 years ago. How can I use modern software engineering techniques?

Conclusion

The solution to problems raised during software development can be done by applying an engineering approach to software development, which includes procedures for Planning, Development, Quality control, Validation and Maintenance. The approach has to be applied consistently, across all types of software, hence the name "Software Engineering".

Assessment

1. Define software.
2. What are the different myths and realities about software?
3. What are the different software components?
4. What is a software crisis? Explain the problems of a software crisis.
5. What are software myths?
6. The "myths" noted are slowly fading as the years pass, but others are taking their place. Attempt to add one or two "new" myths.

Activity 2 – What is Software Engineering

Introduction

This sub-unit provides the concept of applying engineering norms in software development, hence the name "Software Engineering"

Definition

As defined by Pollice, 2005 and IEEE 1993,

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches.

The key criteria for SE are: a well-defined methodology, Predictable milestones, traceability among steps, documentation and control that is maintainability. SE is concerned with the theories, methods and tools which are needed to develop software. A SE is not to produce a working software system only, but also documents such as system design, user manual, etc.

Software engineering deals with both the process of software engineering and the final product. The right process will help produce the right product, but the desired product will also affect the choice of which process to use. A traditional problem in software engineering has been the emphasis on either the process or the product to the exclusion of the other.

Software-Engineering Principles

Software-engineering principles deal with both the process of software engineering and the final product.

The right process will help produce the right product, but the desired product will also affect the choice of which process to use. A traditional problem in software engineering has been the emphasis on either the process or the product to the exclusion of the other. Both are important. To apply principles, the software engineer should be equipped with appropriate methods (general guidelines that govern the execution of some activity; they are accurate, systematic, and disciplined approaches.) and specific techniques that help incorporate the desired properties into processes and products.

Software Product

Software products are software systems delivered to a customer with the documentation which describes how to install and use the system. The critical characteristics of a software product include (Liu, 2001):

Usability: must be useful and usable to improve people's lives.

Software is flexible. A program can be developed to do almost anything. A program can be developed to do almost anything. The characteristic may help to accommodate any kind of change.

Maintainability: should be possible to evolve software to meet the changing needs of customers

Dependability: includes a range of characteristics; reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.

Efficiency: should not make wasteful use of system resources such as memory and processor cycles

Conclusion

To build good systems, we need a well defined development process with clear phases of activities, each of which has an end product, methods and techniques for conducting the phases of activities and for modeling their products and tools for generating the products. The importance of software product characteristics varies from system to system, and optimizing all the characteristics is difficult as some are exclusive.

Assessment

1. What is software engineering?
2. Explain characteristics of software product.

As software becomes more pervasive, risks to the public (due to faulty programs) become an increasingly significant concern. Develop a realistic doomsday scenario (other than Y2K) where the failure of a computer program could do great harm (either economic or human).

Activity 3 – Software Applications

Areas of Software Applications

Software applications are grouped into eight areas for convenience as shown in Figure 1.1 (Agarwal et al, 2010).

System Software

A collection of programs used to run the system as assistance to other software programs. Examples are compliers, editors, utilities, operating system components, drivers, and interfaces. This software resides in the computer system and consumes its resources.

Real-time Software

Deals with a changing environment. First, it collects the input and converts it from analog to a digital, control component that responds to the external environment and performs the action. The software is used to monitor, control, and analyze real-world events as they occur. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

Embedded Software

Software, when written to perform certain functions under control conditions and is further embedded into hardware as a part of large systems, is called embedded

software. The software resides in the Read-Only-Memory (ROM) and is used to control the various functions of the resident products. The products could be a car, washing machine, microwave oven, industrial processing products, gas stations, satellites, and a host of other products, where the need is to acquire input, analyze, identify status, and decide and take action that allows the product to perform in a predetermined manner.

Business Software

Software designed to process business applications is called business software. Business software can be a data- and information processing application. It can drive the business process through transaction processing in on-line or in real-time mode. This software is used for specific operations, such as accounting packages, management information systems, payroll packages, and inventory management.

Personal Computer Software

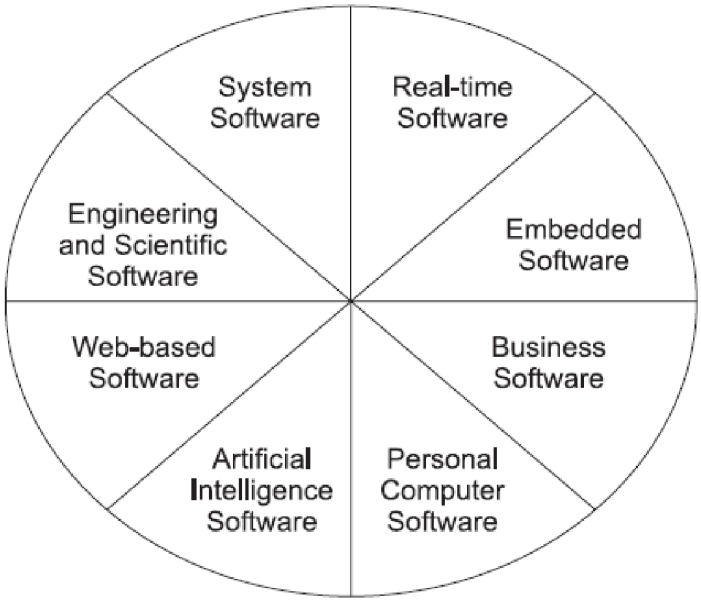
Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external networks, or database access are only a few of hundreds of applications.

Artificial-intelligence Software

Artificial-intelligence software uses non-numerical algorithms, which use the data and information generated in the system to solve complex problems. These problem scenarios are not generally amenable to problem-solving procedures, and require specific analysis and interpretation of the problem to solve it. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

Web-based Software

Web-based software includes the languages by which web pages are processed, i.e., HTML, Java, CGI, Perl, DHTML, etc.



Engineering and Scientific Software

The design and engineering of scientific software deals with processing requirements in their specific fields. They are written for specific applications using the principles and formulae of each field.

Conclusion

Software is important because it affects nearly every aspect of our lives and has become pervasive in our commerce, our culture, and our everyday activities. With Software engineering, any type of software system, including complex systems can be build in timely manner and with high quality

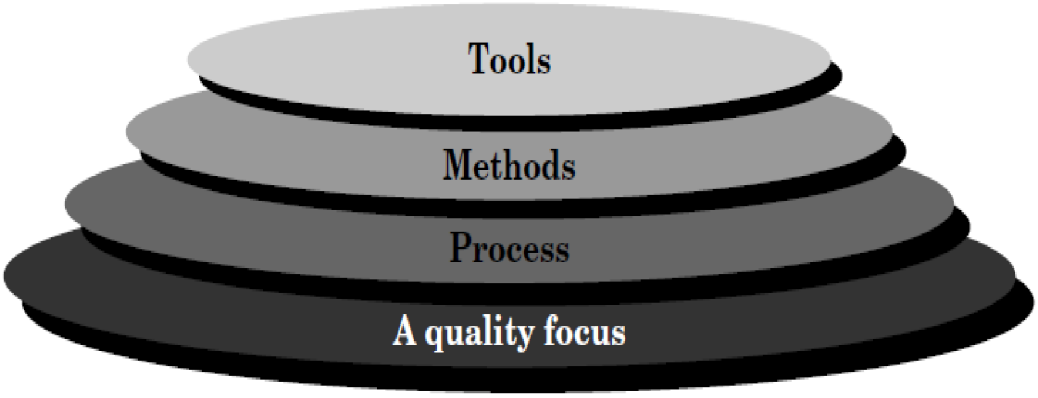
Assessment

1. Give the various application areas of software.
2. Explain "Embedded Software" as one of the software application

Activity 4 – Software Development Process

Software Engineering Layers

Software engineering is a layered technology. Referring to Figure 1.2, any engineering approach (including software engineering) must rest on an organizational commitment to quality. The bedrock that supports software engineering is a quality focus.



To build good systems, we need:

A well defined ***development process*** with clear ***phases of activities,*** each of which has an ***end product***. Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology.

The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Methods and techniques for conducting the phases of activities and for modeling their products

Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Tools for generating the products. Software engineering tools provide automated or semi-automated support for the process and the methods.

The Process

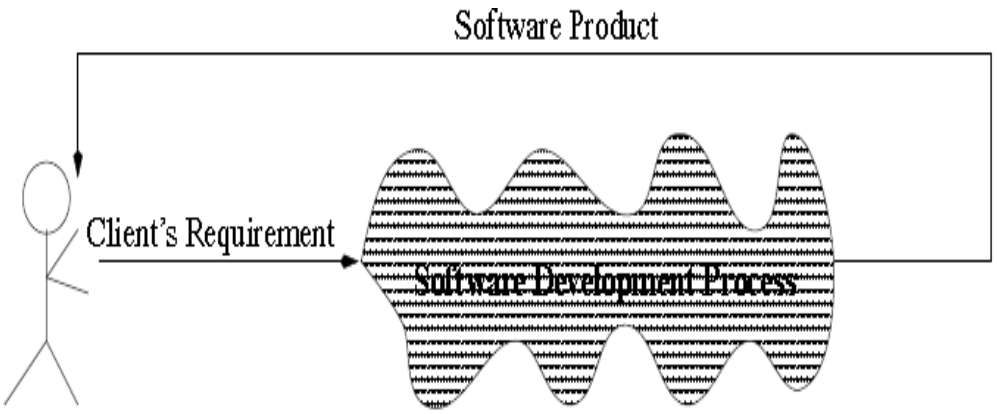
The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. In general, a process is a series of steps involving activities, constraints, and resources that produce an intended output of some kind. Any process has the following characteristics:

- The process prescribes all of the major process activities.
- The process uses resources, subject to a set of constraints (such as a schedule), and produces intermediate and final products.
- The process may be composed of sub-processes that are linked in some way. The process may be defined as a hierarchy of processes, organized so that each sub-process has its own process model.

- Each process activity has entry and exit criteria, so that we know when the activity begins and ends.
- The activities are organized in a sequence, so that it is clear when one activity is performed relative to the other activities.
- Every process has a set of guiding principles that explain the goals of each activity.
- Constraints or controls may apply to an activity, resource, or product

Software Process

All engineering is about how to produce products in a disciplined process. In general, a process defines who is doing what when and how to reach a certain goal. A process to build a software product or to enhance an existing one is called a software development process. A software development process is thus often described in terms of a set of activities needed to transform a user's requirements into a software system (Figure 1.3).



The client's requirements define the goal of the software development. They are prepared by the client (sometime with the help from a software engineer) to set out the services that the system is expected to provide, i.e. functional requirements. The ***functional requirements*** should state ***what*** the system should do rather than ***how it is done.*** Apart from functional requirements, a client may also have non-functional constraints that s/he would like to place on the system, such as the required response time or the use ofa specific language standard.

There are four fundamental process activities, which are common to all software processes. These activities are:

- Software specifications: The functionality of the software and constraints on its operation must be defined.

- Software development: Software that meets the specifications must be produced.
- Software validation: The software must be validated to ensure that it does what the customer wants.
- Software evolution: The software must evolve to meet changing customer needs.

Note that different software processes organize these activities in different ways and are described at different levels of detail. The timing of the activities varies, as does the results of each activity.

Conclusion

Any software development process must start with the activities of capturing and analyzing the client's requirements. These activities and the associated results form the first phase (or sub-process) of the process called requirement analysis (Liu, 2010).

Assessment

1. Why do we need a software development process.
2. Explain in detail the software-engineering process.
3. Describe four fundamental process activities used in software processes.

Activity 5 – Software Development Life Cycle

Introduction

The software-development life-cycle (SDLC) is used to facilitate the development of a large software product in a systematic, well-defined, and cost-effective way. An information system goes through a series of phases from conception to implementation. This process is called the Software-Development Life-Cycle. Various reasons for using a life-cycle model include:

- Helps to understand the entire process
- Enforces a structured approach to development
- Enables planning of resources in advance
- Enables subsequent controls of them
- Aids management to track progress of the system

Phases of SDLC

The SDLC consists of several phases and these phases need to be identified along with defining the entry and exit criteria for every phase. A phase can begin only when the corresponding phase-entry criteria are satisfied. Similarly, a phase can be considered to be complete only when the corresponding exit criteria are satisfied. If there is no clear indication of the entry and exit for every phase, it becomes very difficult to track the progress of the project. The SDLC can be divided into 5-9 phases, i.e., it must have a minimum of five phases and a maximum of nine phases. On average it has seven or eight phases. These are:

- Project initiation and planning/Recognition of need/Preliminary investigation
- Project identification and selection/Feasibility study
- Project analysis
- System design
- Coding
- Testing
- Implementation
- Maintenance

Project initiation and planning/Recognition of need/Preliminary investigation

Recognition of need is nothing but the problem definition. It is the decision about problems in the existing system and the impetus for system change. The first stage of any project or SDLC is called the preliminary investigation. It is a brief investigation of the system under consideration. At this stage the need for changes in the existing system are identified and shortcomings of the existing system are detected.

Feasibility Study

A feasibility study is a preliminary study which investigates the information needs of prospective users and determines the resource requirements, costs, benefits, and feasibility of a proposed project. The goal of feasibility studies is to evaluate alternative systems and to propose the most feasible and desirable systems for development.

System Analysis

Project analysis is a detailed study of the various operations performed by a system and their relationships within and outside the system. Detailed investigation should be conducted with personnel closely involved with the area under investigation, according to the precise terms of reference arising out of the initial study reports.

The tasks to be carried out should be clearly defined such as:

- Examine and document the relevant aspects of the existing system, its shortcomings and problems.
- Analyze the findings and record the results.
- Define and document in an outline the proposed system.
- Test the proposed design against the known facts.
- Produce a detailed report to support the proposals.
- Estimate the resources required to design and implement the system.

The objectives at this stage are to provide solutions to stated problems, usually in the form of specifications to meet the users' requirements and to make recommendations for a new computer-based system. Analysis is an iterative and progressive process, examining information flows and evaluating various alternative design solutions until a preferred solution is available. This is documented as the system proposal.

System Design

System design is the most creative and challenging phase of the SDLC. The term design describes the final system and process by which it is developed. This phase is a very important phase of the life-cycle. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. The design is documented and becomes part of the software configuration.

Coding

The goal of the coding phase is to translate the design of the system into code in a given programming language. In this phase the aim is to implement the design in the best possible manner. This phase affects both testing and maintenance phases.

Well-written code can reduce the testing and maintenance effort. Hence, during coding the focus is on developing programs that are easy to read and understand and not simply on developing programs that are simple to write.

Testing

Testing is the major quality-control measure used during software development. Its basic function is to detect errors in the software. The goal of testing is to uncover requirement, design, and coding errors in the program. Testing is an extremely critical and time-consuming activity. It requires proper planning of the overall testing process.

During the testing of the unit, the specified test cases are executed and the actual results are compared with the expected output. The final output of the testing phase is the test report and the error report, or a set of such reports (one for each unit tested).

Implementation

The implementation phase is less creative than system design. It is mainly concerned with user training, site selection, and preparation and file conversion. Once the system has been designed, it is ready for implementation. Implementation is concerned with those tasks leading immediately to a fully operational system. It involves programmers, users, and operations management, but its planning and timing is a prime function of a systems analyst. It includes the final testing of the complete system to user satisfaction, and supervision of initial operation of the system. Implementation of the system also includes providing security to the system.

Maintenance

Maintenance is an important part of the SDLC. If there is any error to correct or change then it is done in the maintenance phase. Maintenance of software is also a very necessary aspect related to software development. Many times maintenance may consume more time than the time consumed in the development. Also, the cost of maintenance varies from 50% to 80% of the total development cost. Maintenance may be classified as:

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

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

Perfective Maintenance. Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. It involves changes that the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time.As maintenance is very costly and very essential, efforts have been done to reduce its costs. One way to reduce the costs is through maintenance management and software modification audits. Software modification consists of program rewriting and system-level-upgrading.

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

Conclusion

When you build a software system, it's important to go through a series of steps. This is a road map that helps in creating a timely and a high-quality result. This is a systematic approach in which software engineering is emphasizing with a number of phases.

Assessment

1. Discuss the SDLC in brief.
2. Give the basic phases in the software-development life-cycle.
3. What are the different steps in the software-development life-cycle? What are the end products at each step?
4. What are the important activities that are carried out during the feasibility study phase?
5. Explain the different categories of maintenance in the software-development lifecycle.

Activity 6 – Software Development Approaches

Introduction

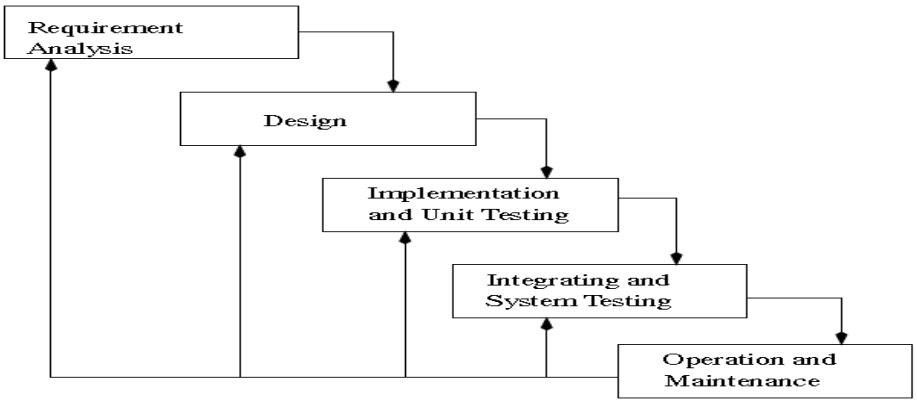
Approaches to systems development, in professional organizations, usually follow one of two basic models: the waterfall model or the evolutionary model.

The Waterfall model

The waterfall model is a very common software development process model. Waterfall is the basis of most of the structured development methods that came into use from the 1970s onwards. It is also known as "The Linear Sequential Model". The linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, implementation, testing, and support or maintenance as shown in Figure 1.4.

The Waterfall model provides a framework for planning top – down systems development. The development flows down a number of successive activity stages. the stages in the waterfall model overlap and feed information to each other.

During design, problems with requirements are identified; during coding, design problems are found and so on. The development process is not a simple linear model but involves a sequence of iterations of the development activities.



Advantages of Waterfall Model

The various advantages of the waterfall model include:

- It is a linear model.
- It is a segmental model.
- It is systematic and sequential.
- It is a simple one.
- It has proper documentation.

Disadvantages of Waterfall Model

The various disadvantages of the waterfall model include:

- It is difficult to define all requirements at the beginning of a project.
- This model is not suitable for accommodating any change.
- A working version of the system is not seen until late in the project's life.
- It does not scale up well to large projects.
- It involves heavy documentation.
- We cannot go backward in the SDLC.
- There is no sample model for clearly realizing the customer's needs.
- There is no risk analysis.
- If there is any mistake or error in any phase then we cannot make good software.
- It is a document-driven process that requires formal documents at the end of each phase.

Problems with the Waterfall model

The waterfall model is the oldest and the most widely used paradigm for software engineering. Among the problems that are sometimes encountered when the waterfall model is applied are:

Real projects rarely follow the sequential flow that the model proposes. Although the waterfall model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed, can be terrible.

Each of these problems is real. However, the waterfall model has a definite and important place in software engineering work. This model is only appropriate when the requirements are well-understood. It provides a template into which methods for analysis, design, implementation, testing, and support can be placed. The waterfall model remains a widely used procedural model for software engineering.

Evolutionary Software Process Models

There is growing recognition that software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight path to an end product unrealistic. The linear sequential model is designed for straight-line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

The techniques used in an evolutionary development include:

Exploratory development where the objective of the process is to work with the client to explore their requirements and deliver a final system. The development starts with the parts of the system which are understood. The system evolves by adding new features as they are proposed by the client.

Prototyping where the objective of the development is to understand the customer's requirements and hence develop a better requirements definition for the system. The prototype concentrates on experimenting with those parts of the client requirements which are poorly understood.

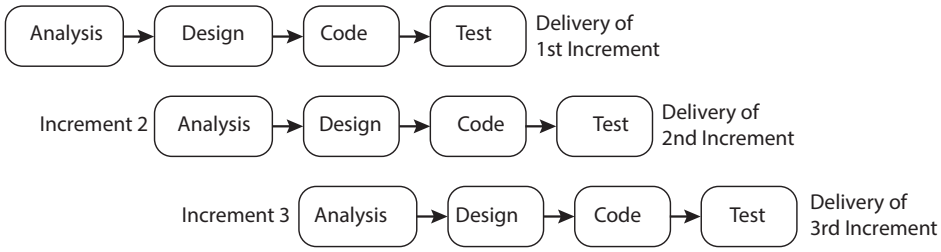
Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refine through many versions until an adequate system has been developed.

Two evolutionary approaches

- The Incremental Model
- The Spiral Model

The Incremental Model

Rather than delivering the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality as shown in Figure 1.5. User requirements are prioritised and the highest priority requirements are included in early increments. Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.



The spiral model

The *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. The project is executed in a series of short lifecycles, each one ending with a release of executable software. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

As per Figure 1.6, a spiral model is divided into a number of framework activities, also called *task regions*. Typically, there are between three and six task regions

Customer communication —tasks required to establish effective communication between developer and customer.

Planning —tasks required to define resources, timelines, and other project related information.

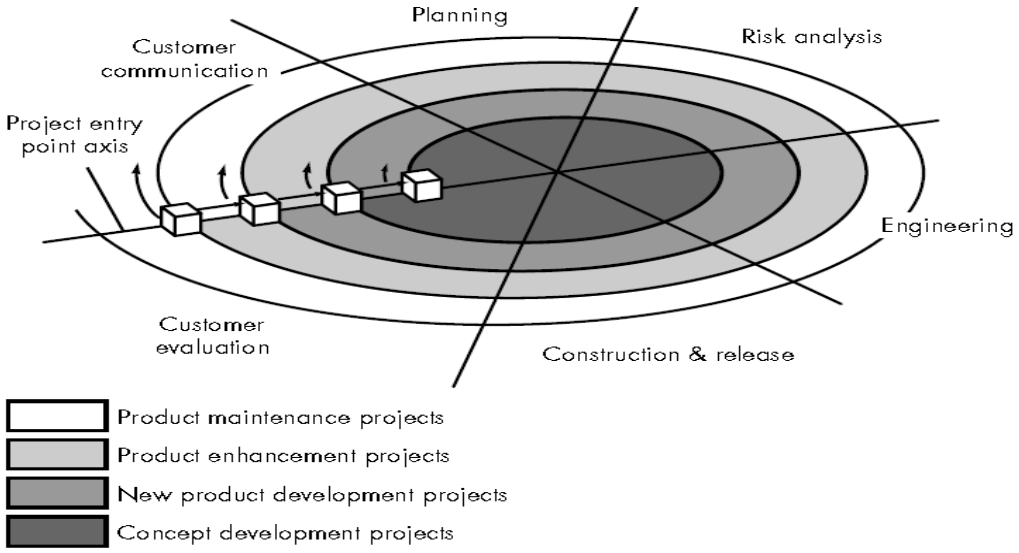
Risk analysis —tasks required to assess both technical and management risks.

Engineering —tasks required to build one or more representations of the application.

Construction and release —tasks required to construct, test, install, and provide user support (e.g., documentation and training).

Customer evaluation —tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

Each of the regions is populated by a set of work tasks, called a task set, that are adapted to the characteristics of the project to be undertaken.



Problems with the Iterations model

Model with the iterations suffers from the following problems

The process is not visible It is difficult and expensive to produce documents which reflect every version of the system.

Systems are poorly structured Continual change tends to corrupt the software structure.

It is not always feasible for large systems, changes in later versions are very much difficult and sometimes impossible. New understanding and new requirements sometimes force the developer to start the whole project all over again. Software evolution is therefore likely to be difficult and costly. Frequent prototyping is also very expensive.

These problems directly lead to the problems that the system is difficult to understand and maintain. It is suggested that this model should be used in the following circumstances:

The development of relatively small systems.

The development of systems with a short lifetime. Here, the system is developed to support some activity with is bounded in time, and hence the maintenance problem is not an important issue.

The development of systems or parts of large systems where it is impossible to express the detailed specification in advance

The ideas, principles and techniques of the evolutionary development process are always useful and should be used in different stages of a wider development process, such as the requirements understanding and validating in the waterfall process.

Conclusion

There are a number of different software development approaches or process models for software engineering which have been proposed, each of which exhibiting strengths and weaknesses, but all having a series of generic phases in common. Software development approach can be chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

Assessment

1. Draw the schematic diagram of the waterfall model of software development. Also discuss its phases in brief.
2. What is a prototype? Draw the schematic diagram of the prototyping model of software development. Also discuss its phases in brief.
3. How do linear and iterative process models differ?
4. Compare the spiral model with the prototyping model by giving their advantages and disadvantages.
5. Is there ever a case when the generic phases of the software engineering process don't apply? If so, describe it.

UNIT SUMMARY

Software has become the key element in the evolution of computer-based systems and products. The problems encountered in terms of software crisis and myths have made the software developers to apply engineering techniques in software development, hence the Software Engineering discipline. Software is composed of programs, data, and documents. Each of these items comprises a configuration that is created as part of the software engineering process. The intent of software engineering is to provide a framework for building software with higher quality.

This unit presented the introduction of software engineering. It explained why engineering discipline was employed in software development after analyzing problems encountered in software development. The unit went further and presented software development process in the form of software development life cycle. Generic process models: the Waterfall and evolutionary models were described.

Unit Assessment

Check your understanding!

1. What do you understand by the term software? *(Answer: section 1.1.2)*
2. Describe three areas of software application. *(Answer: section 1.3)*
3. Distinguish between "engineering and scientific software" and "personal computer software"? (Answer: section 1.3(h) and 1.3(e))
4. What are the different myths and reality about software?
5. (Answer: section 1.1.4)
6. In detail explain the process of software engineering.
7. (Answer: section 1.4.2)
8. Differentiate between "system analysis" and "system design" phases of system development. (Answer: section 1.5.2 (c) and (d))
9. Differentiate between waterfall and evolutionary software process models

(Answer: section 1.6)

Unit Readings and Other Resources

Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1

Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Liu Z., (2001), "*Object-Oriented Software Development Using UML*", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.

Unit II: Planning a Software Project and Software Requirements Analysis and Specification.

Introduction

This unit introduces the bases for obtaining the software requirement specification by employing requirement engineering mechanisms. It explains how to understand what customers want, analyze and validate their requirements and finally produce requirement specification. The unit also elaborates a set of activities related to software project planning. Software project planning touches the cost estimation of the project and the project scheduling.

Objectives

Upon completion of this unit you should be able to:

- Define the following terms: Requirement Engineering, Requirement, software planning
- Describe the processes adopted in requirement engineering
- Demonstrate the process of determining requirements
- Explain the concept of project management and its four important components involved (people, product, process, and project)
- Describe the concept of project management based on finding the cost estimate of the project
- Use the guided steps of software project estimation to obtain the cost of the project

KEY TERMS

Requirement:

A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfill the system's purpose.

Requirement Engineering:

Requirements engineering is the systematic use of proven principles, techniques, and language tools for the cost-effective analysis, documentation, and on-going evaluation of the user's needs and the specifications of the external behavior of a system to satisfy those user needs.

Software-Project Estimation:

Software-project estimation is the process of estimating various resources required for the completion of a project. [Definition]

Learning Activities

Activity 1 – Requirement Engineering

Introduction

In this activity, the definition of "Requirement" and "Requirement Engineering" is presented. It elaborates the process adopted in requirement engineering which will lead to obtaining the system/software requirement specification

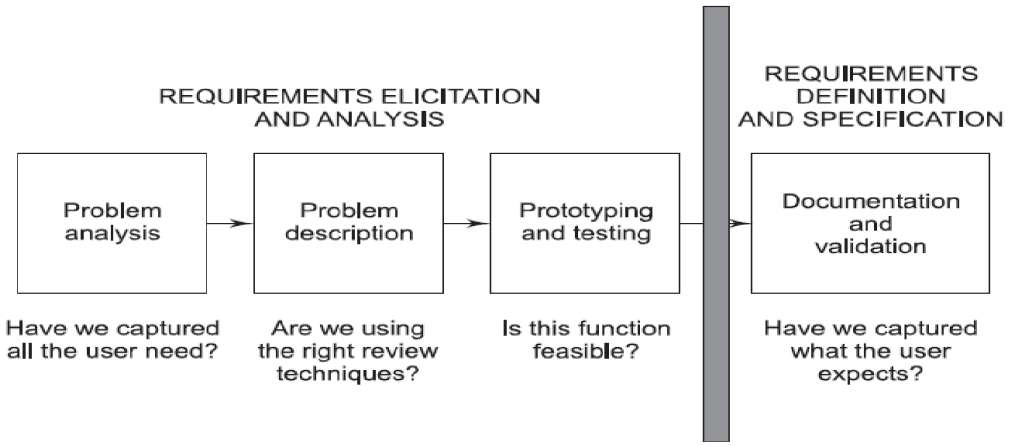
Requirement

A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfill the system's purpose.

Requirements describe the "what" of a system, not the "how." Requirements engineering produces one large document, written in a natural language, and contains a description of what the system will do without describing how it will do it. It provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements

Requirements engineering is the systematic use of proven principles, techniques, and language tools for the cost-effective analysis, documentation, and on-going evaluation of the user's needs and the specifications of the external behavior of a system to satisfy those user needs. It can be defined as a discipline, which addresses requirements of objects all along a system-development process.

As in Figure 2.1, the input to requirements engineering is the problem statement prepared by the customer. The output of the Requirements Engineering (RE) process is a system requirements specification called the Requirement Definition and Description (RDD). The system requirements specification forms the basis for designing software solutions.



Types of Requirements

Requirements are classified into two types

Functional requirements.

They define factors, such as I/O formats, storage structure, computational capabilities, timing, and synchronization.

Are those that relate directly to the functioning of the system.

These are the aspects of the system the client is most likely to recognize.

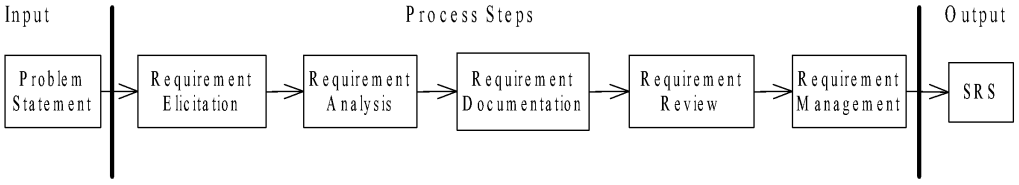
Non-functional requirements.

They define the properties or qualities of a product or they are constraints/restrictions imposed on the system

They include usability, efficiency, performance, space, reliability, portability, etc.

Requirement Engineering Processes

Figure 2.2 illustrates the process steps of requirements engineering. Requirements engineering consists of the following processes:



Requirements gathering (elicitation)

It is a communication process between the parties involved and affected in the problem situation. Sources information includes Customers, End Users, Primary Users, Secondary Users and Stakeholders. Ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. The tools in elicitation are meetings, interviews, video conferencing, e-mails, and existing documents study and facts findings. More than 90% to 95% elicitation should be complete in the initiation stage while the remaining 5% is completed during the development life-cycle.

Requirements elicitation is normally difficult:

Problems of scope

The boundary of the system is not well defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

Problems of understanding

The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.

Problems of volatility

The requirements change over time

Requirements analysis and modelling

Once requirements have been gathered, the work products noted earlier form the basis for requirements analysis. Analysis categorizes requirements and organizes them into related subsets; explores each requirement in relationship to others; examines requirements for validity, consistency, omissions, ambiguity; feasibility and ranks requirements based on the needs of customers/users.

Validity confirms its relevance to goals and objectives and consistently confirms that it does not conflict with other requirements but supports others where necessary.

Feasibility ensures that the necessary inputs are available without bias and error, and technology support is possible to execute the requirement as a solution

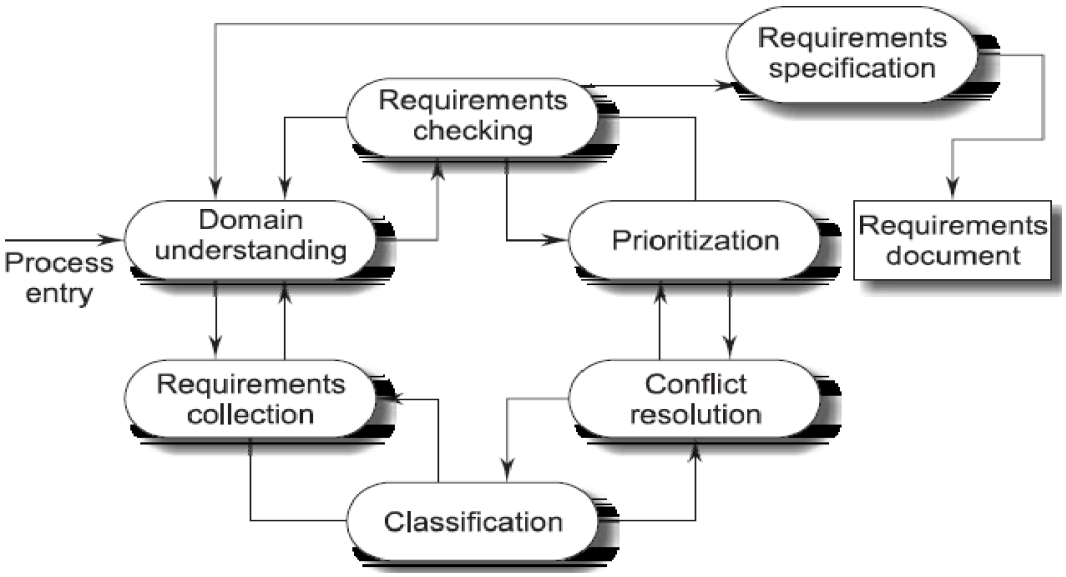
In other point, analysis attempts to find for each requirement, its functionality, features, and facilities and the need for these under different conditions and constraints.

Functionality states "how to achieve the requirement goal."

Features describe the attributes of functionality, and

Facilities provide its delivery, administration, and communication to other systems.

A generic process model of the elicitation and analysis process is shown in Figure 2.3. The process activities are:



Domain Understanding: Analysts must develop their understanding of the application domain.

Requirements Collection: This is the process of interacting with stakeholders in the system to discover their requirements. More domain understanding develops further during this activity.

Classification: This activity takes the unstructured collection of requirements and organizes them into coherent clusters.

Conflict Resolution: When multiple stakeholders are involved, requirements will conflict. This activity is concerned with findings and resolving these conflicts.

Prioritization: In any set of requirements some priorities will be more important than others. This stage involves interaction with stakeholders to discover the most important requirements.

Requirements Checking: The requirements are checked to discover if they are complete, consistent, and in accordance with what stakeholders really want from the system.

Requirements review (Validation)

This is a manual process which involves multiple readers from both client and contractor staff checking the requirements document for anomalies and omissions.

Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

A requirements review can be informal or formal.

Informal reviews simply involve contractors is discussing requirements with as many system stakeholders as possible. Many problems can be detected simply by talking about the system to stakeholders before making a commitment to a formal review.

In a **formal requirements review,** the development team should 'walk' the client through the system requirements, explaining the implications of each requirement. The review team should check each requirement for consistency and should check the requirements as a whole for completeness. The formal technical review is the primary requirements validation mechanism

Requirements management

Requirements define the capability that the software system solution must deliver and the intended results that must result on its application to business problems. In order to generate such requirements, a systematic approach is necessary, through a formal management process called Requirements Management.

Requirements management is defined as a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and project team on the changing requirements of the system. It is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds.

Requirements management begins with identification. Each requirement is assigned a unique identifier that might take the form

<requirement type><requirement #>

Where requirement type takes on values such as F = functional requirement, D = data requirement, B = behavioral requirement, I = interface requirement, and P = output requirement.

Conclusion

The important and difficult part in building a software system is in deciding precisely what to build and establishing the detailed technical requirements. Requirements are identified by eliciting information from the customer. Requirements are analyzed to assess their clarity, completeness, and consistency. Finally, system requirements are managed to ensure that changes are properly controlled.

Assessment

1. What do the term requirements mean?
2. Explain the process of determining the requirements for a software-based system.
3. Discuss the significance and use of requirement engineering. What are the problems in the formulation of requirements?
4. What are the crucial process steps of requirement engineering? Discuss with the help of a diagram.
5. Explain the importance of requirements. How many types of requirements are possible and why?
6. What is requirements elicitation? Discuss any two techniques in detail.

Activity 2 – Planning Software Project

Introduction

This activity briefly explains the concept of "project management" by emphasizing more on project cost estimation. It presents four most need components in project management which are people, product, process, and project. The activity also describes the main steps to be followed in software-project estimation

Project Management

Project management involves the planning, monitoring, and control of the people, process, and events that occur as software evolves from a preliminary concept to an operational implementation. Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed. In project management there is a need to understand the four P's—people, product, process, and project.

People must be organized to perform software work effectively.

Communication with the customer must occur so that product scope and requirements are understood.

Before a project can be planned, product1 objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

A **process** must be selected that is appropriate for the people and the product.

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.

The **project** must be planned by estimating effort and calendar time to accomplish work tasks: defining work products, establishing quality checkpoints. A planned and controlled software project is the way of managing complexity. In order to avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring and controlling the project. Software project management begins with a set of activities that are collectively called project planning. There is a need to estimate the work to be done, the resources that will be required, and the time that will elapse from start to finish before the project begin. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course.

Project Planning Objectives

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. The planning objective is achieved through a process of information discovery that leads to reasonable estimates. Activities associated with software project planning include:

Determination of Software Scope

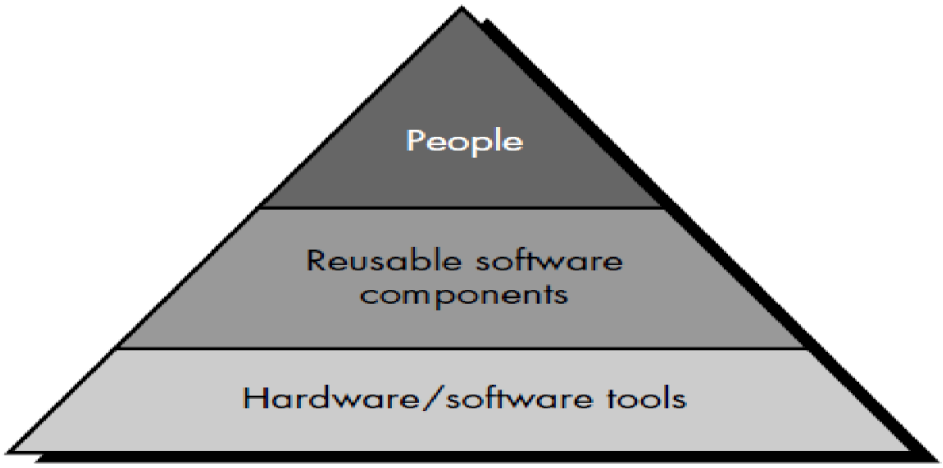
Function and performance allocated to software during requirement engineering should be assessed to establish a project scope that is unambiguous and understandable at the management and technical levels. Software scope describes the data and control to be processed, function, performance, constraints, interfaces, and reliability.

Functions described in the statement of scope are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

Estimation of the Resource

Figure 2.4 illustrates development resources as a pyramid. The development environment—hardware and software tools—sits at the foundation of the resources pyramid and provides the infrastructure to support the development effort. At a higher level, we encounter reusable software components— software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource—people.

Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required; duration of time that resource will be applied. The last two characteristics can be viewed as a time window. Availability of the resource for a specified window must be established at the earliest practical time.



Software Project Estimation

Effective software project estimation is one of the most challenging and important activities in software development. Proper project planning and control is not possible without a sound and reliable estimate. The following sub-unit elaborates more on software project estimation.

Project Cost Estimation

Estimation lays a foundation for all project planning activities and project planning, hence provides the road map for successful software engineering. Underestimating software projects and understaffing it often leads to low-quality deliverables, and the project misses the target deadline leading to customer dissatisfaction and loss of credibility to the company.

On the other hand, overstaffing a project without proper control will increase the cost of the project and reduce the competitiveness of the company

Software-project estimation is the process of estimating various resources required for the completion of a project. Software-project estimation mainly encompasses the following steps:

Estimating the Size of the Project

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

Estimating Efforts Based on Person-months or Person-hours

Person month is an estimate of the personal resources required for the project.

Estimating Schedule in Calendar Days/Month/Year Based on Total Person-months Required and Manpower Allocated to the Project

The duration in calendar month = Total person-months/Total manpower allocated.

Estimating Total Cost of the Project Depending on the Above and Other Resources

In a commercial and competitive environment, software-project estimation is crucial for managerial decision-making.

Table 2.1 gives the relationship between various management functions and software metrics/ indicators. Project estimation and tracking help to plan and predict future projects and provide baseline support for project management and supports decision-making

Activity	Tasks Involved
Planning	Cost estimation, planning for training of manpower, project scheduling, and budgeting the project.
Controlling	Size metrics and schedule metrics help the manager to keep control of the project during execution.
Monitoring/ improving	Metrics are used to monitor progress of the project and wherever possible sufficient resources are allocated to improve it.

Estimating Size

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

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

Estimating Effort

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

Estimating Schedule

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

Schedule estimation in calendar months can be calculated using the following model [McConnell]: Schedule in calendar months = 3.0*(person-months)1/3. Where the parameter 3.0 is variable, used depending on the situation that works best for the organization.

Estimating Cost

Cost estimation is the next step for projects. The cost of a project is derived not only from the estimates of effort and size but from other parameters, such as hardware, travel expenses, telecommunication costs, training costs, etc. Figure 2.6 depicts the cost-estimation process and Figure 2.7 depicts the project-estimation process.

Reasons for Poor/Inaccurate Estimation

The following are some of the reasons for poor and inaccurate estimation:

- Requirements are imprecise. Also, requirements change frequently.
- The project is new and is different from past projects handled.
- Non-availability of enough information about past projects.

- Estimates are forced to be based on available resources.
- Cost and time tradeoffs.

Problems associated with estimates:

- Estimating size is often skipped and a schedule is estimated, which is of more relevance to management.
- Estimating size is perhaps the most difficult step, which has a bearing on all other estimates.
- Good estimates are only projections and subject to various risks.
- Organizations often give less importance to collection and analysis of historical data of past development projects. Historical data is the best input to estimate a new project.
- Project managers often underestimate the schedule because management and customers often hesitate to accept a prudent realistic schedule.

Project-Estimation Guidelines

Some guidelines for project estimation are as follows:

- Preserve and document data pertaining to past projects.
- Allow sufficient time for project estimation especially for bigger projects.
- Prepare realistic developer-based estimates. Associate people who will work on the project to reach a realistic and more accurate estimate.
- Use software-estimation tools.
- Re-estimate the project during the life-cycle of the development process.
- Analyze past mistakes in the estimation of projects.

Conclusion

One of the problems affiliated in software development is to under-estimating resources required for a project. Developing a practical project plan is essential to gain an understanding of the resources required, and how these should be applied.

Assessment

Why does cost estimation play an important role in the software-development process?

Clearly describe processes for project-estimation

Explain the term "Estimate Effort" in view of the software cost estimation

What are the various reasons for poor/inaccurate estimation?

UNIT SUMMARY

The software project planner must estimate three things before a project begins: how long it will take, how much effort will be required, and how many people will be involved. The unit presented requirement engineering and its process to obtain the requirement specification. It also presented software project planning and specification in project cost estimation.

Unit Assessment

Check your understanding!

Discuss the significance and use of requirement engineering. (Answer: section 2.1.2)

What are the problems encountered during requirements gathering? (Answer: section 2.1.4.1)

Describe processes of requirement engineering. (Answer: section 2.1.4)

What does the term "Software Requirement Specification" mean? (Answer: section 2.1.4.3)

Why does cost estimation play an important role in the software-development process? (Answer: section 2.2.4)

List and explain steps to be followed during software project estimation. (Answer: section 2.2.4)

List six guidelines used for software project estimation. (Answer: section 2.2.7)

Unit Readings and Other Resources

- Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1
- Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Unit III: Software Design

Unit Introduction

The purpose of the software design is to produce a model or a representation of an entity that will later be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. This unit defines what a software design is. It describes design objectives and design principles. It further elaborates design software processes which include: Architectural Design, Abstract Design, User-Interface Design, Low-Level Design. Function-oriented design, object-oriented design, design notation and specification and verification for design were presented.

Unit Objectives

Upon completion of this unit you should be able to:

- Describe what is software design
- Explain software design objectives and principles
- Distinguish and describe phases/processes of software design
- Describe importance of each phase/process of software design
- Distinguish between "Functional-Oriented" and "Object-Oriented " approaches
- Develop design specifications and verify design

KEY TERMS

Software Design:

Software design is the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details.

Design Principles

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.

Design Specifications

Design specifications address different aspects of the design model and are completed as the designer refines his representation of the software.

Learning Activities

Activity 1 – Software Design

Introduction

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

Definition of Software Design

Agarwal et al., (2010) is referring the definition of software design as stated by Coad and Yourdon to be the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details.

The input to software design includes an understanding of requirements, environmental constraints and design criteria, while the output of the design effort is composed of the following:

- Architecture design which shows how pieces are interrelated
- Specifications for any new pieces
- Definitions for any new data

During software design, software requirements specification feed the design task. Using one of a number of design methods the design task produces a data design, an architectural design, an interface design, and a component design.

The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software.

The architectural design defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied [SHA96]. The architectural design representation—the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.

The interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior.

The component-level design transforms structural elements of the software architecture into a procedural description of software components.

During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. The importance of software design can be stated with a single word—quality. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support steps that follow. Without design, we risk building an unstable system:

- One that will fail when small changes are made;
- One that may be difficult to test;
- One whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

Design Objectives/Properties

Below are some of the properties or objectives of the software design:

Correctness

The design of a system is correct if the system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. The goal should focus in finding the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

Verifiability

Design should be correct and it should be verified for correctness. Verifiability is concerned with how easily the correctness of the design can be checked. Various verification techniques should be easily applied to design.

Completeness

Completeness requires that all the different components of the design should be verified, i.e., all the relevant data structures, modules, external interfaces, and module interconnections are specified.

Traceability

Traceability is an important property that can get design verification. It requires that the entire design element be traceable to the requirements.

Efficiency

Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system consumes less processor time and memory.

Simplicity

Simplicity is the most important quality criteria for software systems. Maintenance of a software system is usually quite expensive. The design of the system is one of the most important factors affecting the maintainability of the system.

Design Principles

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, that creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design.

Like an architect's plans for a house, the design model begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software.

Basic design principles enable the software engineer to navigate the design process. Principles for software design as stipulated by Agarwal et al., 2010, include:

The design process should not suffer from "tunnel vision."

A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts.

The design should be traceable to the analysis model

Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

The design should not reinvent the wheel

Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.

That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.

The design should exhibit uniformity and integration.

A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

The design should be structured to accommodate change

The design concepts discussed in the next section enable a design to achieve this principle.

The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered

Well designed software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

Design is not coding, coding is not design

Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

The design should be assessed for quality as it is being created, not after the fact

A variety of design concepts (Section 13.4) and design measures (Chapters 19 and 24) are available to assist the designer in assessing quality.

The design should be reviewed to minimize conceptual (semantic) errors

There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors [MEY88].

1 .External quality factors are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability).

2. Internal quality factors are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.

As per Pressman, (2001), there are three design principles:

- Problem partitioning
- Abstraction and
- Top-down and Bottom-up design

Problem Partitioning

When solving a small problem, the entire problem can be tackled at once. But solving larger problems, the basic principle is the time tested principle of "divide and conquer." This principle suggests dividing into smaller pieces, so that each piece can be conquered separately. For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately.

Abstraction

An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Abstraction is an crucial part of the design process and it is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other, but interact with other components. In order to allow the designer to concentrate on one component at a time, abstraction of other components is used.

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.

During the design process, abstractions are used in a reverse manner not in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions.

Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

Top-down and Bottom-up Design

A system consists of components, which have components of their own, hence a system is a hierarchy of components. The highest-level components correspond to the total system. To design such hierarchies there are two possible approaches: top-down and bottom-up.

The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast,

A bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

Top-Down Approach

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

Top-down design methods often result in some form of stepwise refinement. Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

Bottom-Up Approach

A bottom-up design approach as shown in Figure 3.2 starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with layers of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and still a higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

Note:

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. However,

Bottom-up approach is used if a system to be built is from an existing system.

This is because it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used).

Conclusion

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product .Design principles establish an overriding philosophy that guides the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the constn1ction activity that follows Pressman, 2001).

Exercise

What is system design?

Explain, in detail, the three design principles in system design.

What is abstraction? What are the verification metrics for system design?

Define:

Problem partitioning

Abstraction

Top-down and bottom-up design

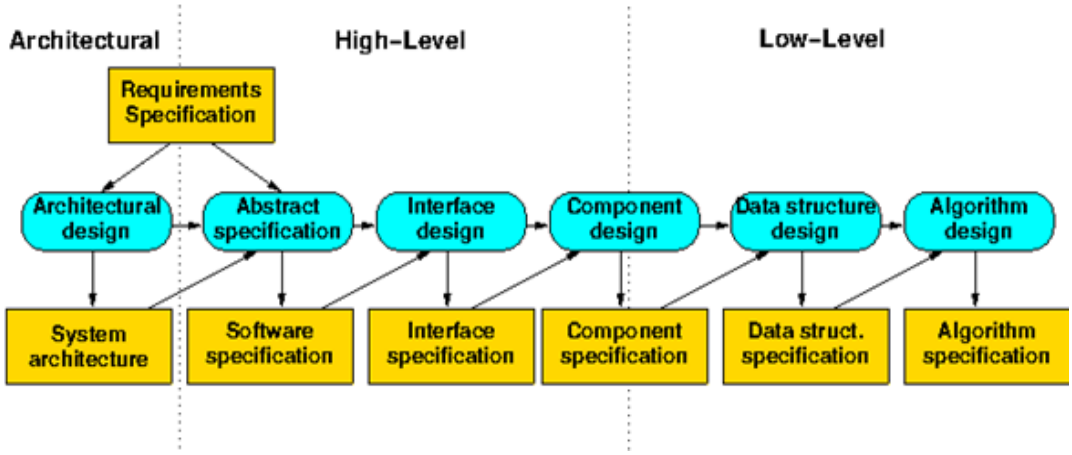
Activity 2 – The Design Process

Introduction

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.

As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. The means that the pieces of a problem are solvable separately; the cost of solving the entire problem is more than the sum of the cost of solving all the pieces. However, the different pieces cannot be entirely independent of each other as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem.

Figure 3.3 shows the general model of the software design process.



Architectural Design

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

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

- Analyze the effectiveness of the design in meeting its stated requirements,
- Consider architectural alternatives at a stage when making design changes is still relatively easy, and Reducing the risks associated with the construction of the software.

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

Data-centric architecture

Data-centric architecture involves the use of a central database operation of inserting and updating it in the form of a table. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.

Figure 3.4 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software.

Data-flow architecture

Data-flow architecture is central around the pipe and filter mechanism. This architecture is applied when input data takes the form of output after passing through various phases of transformations. These transformations can be via manipulations or various computations done on the data.

Object-oriented architecture

In object-oriented architecture the software design moves around the clauses and objects of the system. The class encapsulates the data and methods. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

Layered architecture

Layered architecture defines a number of layers and each layer performs tasks. The outer-most layer handles the functionality of the user interface and the innermost layer mainly handles interaction with the hardware.

The basic structure of a layered architecture is illustrated in Figure 3.5. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing (Core layer). Intermediate layers provide utility services (Utility layer) and application software (Application layer) functions.

Objectives of Architectural Design

To develop a model of software architecture, which gives an overall organization of the program module in the software product. Software architecture encompasses two aspects of structures of the data and hierarchical structures of the software components. Architectural design defines the organization of program components. It does not provide the details of each component and its implementation.

To control the relationship between modules. One module may control another module or may be controlled by another module. The organization of a module can be represented by a tree-like structure.

Why Is Software Architecture Important?

Software architecture is important because:

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

Abstract Specification

For each sub-system, an abstract specification of the services it provides and the constraints under which it must operate is produced. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

There are two common abstraction mechanisms for software systems:

Functional abstraction and Data abstraction.

In functional abstraction, a module is specified by the function it performs. For example, 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 system is partitioned into smaller functions that comprise the system function.

The second unit for abstraction is data abstraction. There are certain operations required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. 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 internals of the object are hidden; only the operations on the object are visible.

User-Interface Design

Interface design focuses on three areas of concern:

The design of interfaces between software components

The design of interfaces between the software and other nonhuman producers and consumers of information (i.e. other external entities), and

The design of the interface between a human (i.e., the user) and the computer.

This process of software design is concerned with the third interface design category—user interface design.

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

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 layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

User interface design has as much to do with the study of people. Some few questions that must be asked and answered as part of user interface design include:

- Who is the user?
- How does the user learn to interact with a new computer-based system?
- How does the user interpret information produced by the system?
- What will the user expect of the system?

Golden Rules in User Interface Design

There are three golden rules which form the basis for a set of user interface design principles

Place the user in control.

Reduce the user's memory load.

Make the interface consistent.

Place the User in Control

The followings are the design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

Reduce the user's memory load

The more a user has to remember, the more error-prone will be the interaction with the system. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall.

Make the interface consistent.

The interface should present and acquire information in a consistent fashion. This implies that

All visual information is organized according to a design standard that is maintained throughout all screen displays,

Input mechanisms are constrained to a limited set that are used consistently throughout the application, and Mechanisms for navigating from task to task are consistently defined and implemented.

Process of Designing User Interface

The overall process for designing a user interface:

Begins with the creation of different models of system function (as perceived from the outside).

The human- and computer-oriented tasks that are required to achieve system function are then outlined; Design issues that apply to all interface designs are considered; tools are used to prototype and Ultimately implement the design model; and the result is evaluated for quality.

Interface Design Models

A design model of the entire system incorporates data, architectural, interface, and procedural representations of the software. The requirements specification may establish certain constraints that help to define the user of the system, but the interface design is often only incidental to the design model. The user model establishes the profile of end-users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality"

The User Interface Design Process

The design process for user interfaces is iterative and can be represented using a spiral model as shown in Figure 3.6. The spiral implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the implementation activity involves prototyping—the only practical way to validate what has been designed. The user interface design process encompasses four distinct framework activities:

User, task, and environment analysis and modelling

The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

Interface design

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences. The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

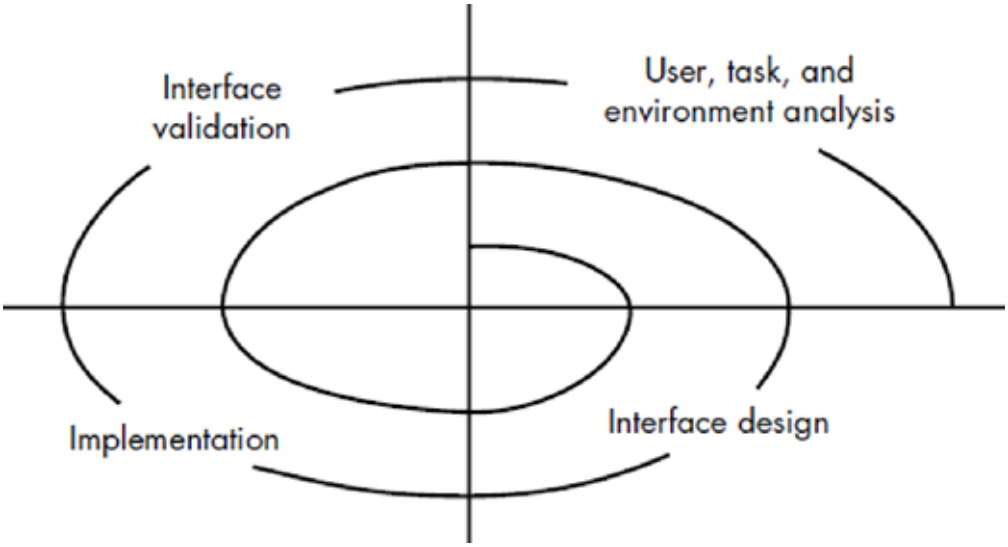
Interface validation

Validation focuses on

The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements;

The degree to which the interface is easy to use and easy to learn; and

The users' acceptance of the interface as a useful tool in their work.



Component Design

Component-level design, also called procedural design, occurs after data, architectural, and interface designs have been established. That is Data, architectural, and interface design must be translated into operational software. To accomplish this, the design must be represented at a level of abstraction that is close to code. Component-level design establishes the algorithmic detail required to manipulate data structures, effect communication between software components via their interfaces, and implement the processing algorithms allocated to each component.

Component-level design is represented using a programming language. In essence, the program is created using the design model as a guide. An alternative approach is to represent the procedural design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code.

Structured Programming

Pressman, 2001 state that in the early 1960s a set of constrained logical constructs was proposed from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure, was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily. The constructs are sequence, condition, and repetition.

Sequence implements processing steps that are essential in the specification of any algorithm.

Condition provides the facility for selected processing based on some logical occurrence, and

Repetition allows for looping.

These three constructs are fundamental to structured programming—an important component-level design technique.

Low-Level Design

Modularization

A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

Modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved.

Modular system consists of well-defined, manageable units with well-defined interfaces among the units. Desirable properties of a modular system include:

Each function in each abstraction has a single, well-defined purpose.

Each function manipulates no more than one major data structure.

Functions share global data selectively. It is easy to identify all routines that share a major data structure. Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

Structure Charts

The structure chart is one of the most commonly used methods for system design. Structure charts are used during architectural design to document hierarchical structures, parameters, and interconnections in a system.

It partitions a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to a black box and appropriate outputs are generated by the black box. This concept reduces complexity because details are hidden from those who have no need or desire to know. Thus, systems are easy to construct and easy to maintain.

Pseudo-Code

"Pseudo" means imitation or false and "code" refers to the instructions written in a programming language. Pseudo-code notation can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise English language phrases that are structured by keywords, such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Pseudo-code is also known as program-design language or structured English.

Flowcharts

A flowchart is a convenient technique to represent the flow of control in a program. A flowchart is a pictorial representation of an algorithm that uses symbols to show the operations and decisions to be followed by a computer in solving a problem. The actual instructions are written within symbols/boxes using clear statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation in a sequence.

Flowcharts are the plan to be followed when the program is written. Expert programmers may write programs without drawing the flowcharts. But for a beginner it is recommended that a flowchart should be drawn before writing a program, which in turn will reduce the number of errors and omissions in the program. Flowcharts also help during testing and modifications in the programs.

Difference between Flowcharts and Structure Charts

A structure chart differs from a flowchart in the following ways:

It is usually difficult to identify different modules of the software from its flowchart representation.

Data interchange among different modules is not represented in a flowchart.

Sequential ordering of tasks inherent in a flowchart is suppressed in a structure chart.

A structure chart has no decision boxes.

Unlike flowcharts, structure charts show how different modules within a program interact and the data that is passed between them.

Conclusion

Design is the place where stakeholder requirements, business needs and technical consideration all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interlaces, and components that are necessary to implement the system.

Exercises

Define architectural design.

What are the objectives of architectural design?

Explain the various design techniques that come under the category of low-level design.

Define:

- Modularization
- Structure charts
- Pseudo-code
- Flowcharts

Distinguish between "Structure Charts" and ""Flowcharts" giving an example to each

Give the hierarchical format of a structure chart. Also, give the basic building blocks of a structure chart.

Develop two additional design principles that "place the user in control."

Develop two additional design principles that "reduce the user's memory load."

Develop two additional design principles that "make the interface consistent."

All modern programming languages implement the structured programming constructs. Provide examples from three programming languages.

Activity 3 – Other Aspects of the Software Design

Functional-Oriented Versus the Object-Oriented Approaches

Table 3.1 provides the difference between the Functional-Oriented Versus the Object-Oriented Approach in software design

S/N	Functional-oriented Approach	Object-oriented Approach
	The basic abstractions, which are given to the user, are real-world functions, such as sort, merge, track, display, etc.	The basic abstractions are not the real-world functions, but are the data abstraction where the real-world entities are represented, such as picture, machine, radar system, customer, student, employee, etc.
	Functions are grouped together by which a higher-level function is obtained. An example of this technique is software analysis/Software Design (SA/SD).	The functions are grouped together on the basis of the data they operate on, such as in class person, function displays are made member functions to operate on its data members such as the person name, age, etc.
	The state information is often represented in a centralized shared memory.	The state information is not represented in a centralized shared memory but is implemented/distributed among the objects of the system.

Design Specifications

Design specifications address different aspects of the design model and are completed as the designer refines his representation of the software.

First, the overall scope of the design effort is described, which is derived from system specification and the analysis model (software requirements specification).

Then, data design is specified, which includes data structures, any external file structures, internal data structures, and a cross-reference that connects data objects to specific files.

Then architectural design indicates how the program architecture has been derived from the analysis model. Structure charts are used to represent the module hierarchy.

Interface design indicates the design of external and internal program interfaces along with a detailed design of the human/machine interface. A detailed prototype of a GUI may also be represented. Procedural design specifies components—separately addressable elements of software—such as subroutines, functions, or procedures in the form of English language processing narratives. This narrative explains the procedural function of a component (module).

Design specification contains a requirements cross-reference. The purpose of this cross-reference is:

- To establish that all requirements are satisfied by the software design.
- To indicate which components are critical to the implementation of specific requirements.

The final section of the design specification contains supplementary data, such as algorithm descriptions, alternative procedures, and tabular data.

Verification for Design

Like in other phases in the development process, the output of the system design phase should be verified before proceeding with the activities of the next phase.

If the design is expressed in some formal notation for which analysis tools are available, then through tools it can be checked for internal consistency (e.g., those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.).

If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used.

There are two fundamental approaches to verification.

- The first consists of experimenting with the behavior of a product to see whether the product performs as expected (i.e., testing the product).
- The other consists of analyzing the product—or any design documentation related to it—to deduce its correct operation as a logical consequence of the design decisions. The two categories of verification

Conclusion

A number of design methods or approaches are being applied throughout t he industry. Each soft ware design approach introduces unique heuristics and notation view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: a mechanism for the translation of the requirements model into a design representation, a notation for representing functional components and their interfaces, heuristics for refinement and partitioning, and guidelines for quality assessment.

Regardless of t he design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design.

Verification in the design phase is aimed at assessing the correctness, consistency, and adequacy of the design with respect to the requirements and analysis models.

Exercise

Give any two important differences between the function-oriented and object-oriented design approaches.

Discuss the major advantages of the object-oriented design approach over the function-oriented design approach.

Explain the term design specification.

What is abstraction? What are the verification metrics for system design?

Do you design software when you "write" a program? What makes software design different from coding?

UNIT SUMMARY

Design is the most important part of software engineering. During design, progressive refinements of data structure, architecture, interfaces, and procedural detail of software components are developed, reviewed, and documented. Design results in representations of software that can be assessed for quality.

A number of fundamental software design principles and concepts have been presented. Design principles guide the software engineer as the design process proceeds. Design concepts provide basic criteria for design quality. Modularity (in both program and data) and the concept of abstraction enable the designer to simplify and reuse software components.

Refinement provides a mechanism for representing successive layers of functional detail. Program and data structure contribute to an overall view of software architecture, while procedure provides the detail necessary for algorithm implementation. Information hiding and functional independence provide heuristics for achieving effective modularity.

Unit Assessment

Check your understanding!

List five software design principles and describe any two of them. (Answer: section 3.1.4)

Describe the essence of "problem partitioning" in software design. (Answer: section 3.1.4.1)

Describe the statement "Design is not coding, coding is not design" as one of the software design principle. (Answer: section 3.1.4(h))

With the aid of diagrams, differentiate between Top-down and bottom-up software design approaches. (Answer: section 3.1.4.3)

Briefly explain the user-interface design of the software process. (Answer: section 3.2.4)

Differentiate between a flowchart and a pseudo code. (Answer: sections 3.2.6.3 and 3.2.6.4)

Discuss the term verification in reference to system design. (Answer: section 3.3.3)

Unit Readings and Other Resources

- Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1
- Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Unit IV: Implementation and Testing

Unit Introduction

During implementation phase, each of the components from the design is realized as a program unit. Each unit then must be either verified or tested against its specification obtained in the design stage. Testing the software is how to ensure that it responds the specifications from the design phase to implementation. This unit will address in general all aspects related to the implementation and testing of software. The unit will describe the techniques used to implement software and the mechanisms used to perform software testing.

Unit Objectives

Upon completion of this unit you should be able to:

- Define what is implementation and testing
- Describe basic principles of testing and testing objectives
- Determine whether the observed behavior conforms to the expected behavior
- Elaborate different levels of testing
- Distinguish and perform white-box and black-box testing
- Define the different testing techniques and distinguish their differences
- Perform the test plans

KEY TERMS

Software Implementation:

Software implementation is a process of realizing the design specification as a program unit

Software Testing

Testing is a set of activities used to test the source code in order to uncover (and correct) errors before delivery of software to customer.

Test Case

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

Structural Testing

Structural testing is an approach to testing where the tests are derived from knowledge of the software's structure and implementation.

Functional testing

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

Learning Activities

Activity 1 – Software Coding

Introduction

During implementation stage, each of the components from the design is realized as a program unit. Each unit then must be either verified or tested against its specification obtained in the design stage. This process is as depicted in Figure 4.1. Then the individual program units representing the components of the system are combined and tested as a whole to ensure that the software requirements have been met. When the developers are satisfied with the product, it is then tested by the client (acceptance testing). This phase ends when the product is accepted by the client.

Coding

During coding the focus is on developing programs that are easy to read and understand and not simply on developing programs that are simple to write. Coding can be subject to company-wide standards that may define the entire layout of programs, such as headers for comments in every unit, naming conventions for variables, classes and functions, the maximum number of lines in each component, and other aspects of standardization.

Structured programming helps the understandability of programs. The goal of structured programming is to linearize the control flow in the program. Single entry-single exit constructs should be used. The constructs include selection (if-then-else) and iteration (while, repeat-unit).

Structured Programming

Structured programming refers to a general methodology of writing good programs. A good program is one that has the following properties:

- It should perform all the desired actions.
- It should be reliable, i.e., perform the required actions within acceptable margins of error.
- It should be clear, i.e., easy to read and understand.
- It should be easy to modify.
- It should be implemented within the specified schedule and budget.

Structured programs have the single-entry, single-exit property. This feature helps in reducing the number of paths for flow of control. If there are arbitrary paths for the flow of control, the program will be difficult to read, understand, debug, and maintain.

A program is one of two types: static structure or dynamic structure.

The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program.

The dynamic structure of the program is the sequence of statements executed during the execution of the program.

Both static and dynamic structures are the sequence of statements. The only difference is that the sequence of statements in a static structure is fixed, whereas in a dynamic structure it is not fixed. That means the dynamic sequence of statements can change from execution to execution. The static structure of a program can be easily understood. The dynamic structure of a program can be easily seen at the time of execution.

Objectives of Structured Programming

The objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program.

As the statements in a program text are linearly organized, the objective of structured programming is to develop programs whose control flow during execution is linearized and follows the linear organization of the program text.

Since program cannot be written as a sequence of simple statements without any branching or repetition, structured constructs are used. In structured programming, a statement is not a simple assignment statement, it is a structured statement.

Principles of Structured Programming

All structured program design methods are based upon the two fundamental principles stepwise refinement and three structured control constructs. The objective of program design is to transform the required function of the program, as stated in the program specification, into a set of instructions, which can easily be translated into a chosen programming language. The process of stepwise refinement is an approach that the stated program function is broken down into subsidiary functions in progressively increasing levels of detail until the lowest level functions are achievable in the programming language.

The second principle of structured program design is that any program can be constructed using only three structured control constructs. The constructs selection, iterations, and sequence are shown in Figure 4.2 (a, b, c, d). Any program independent of the technology platform can be written using these constructs, i.e., selection, repetition, sequence. These structures are the basis of structured programming.

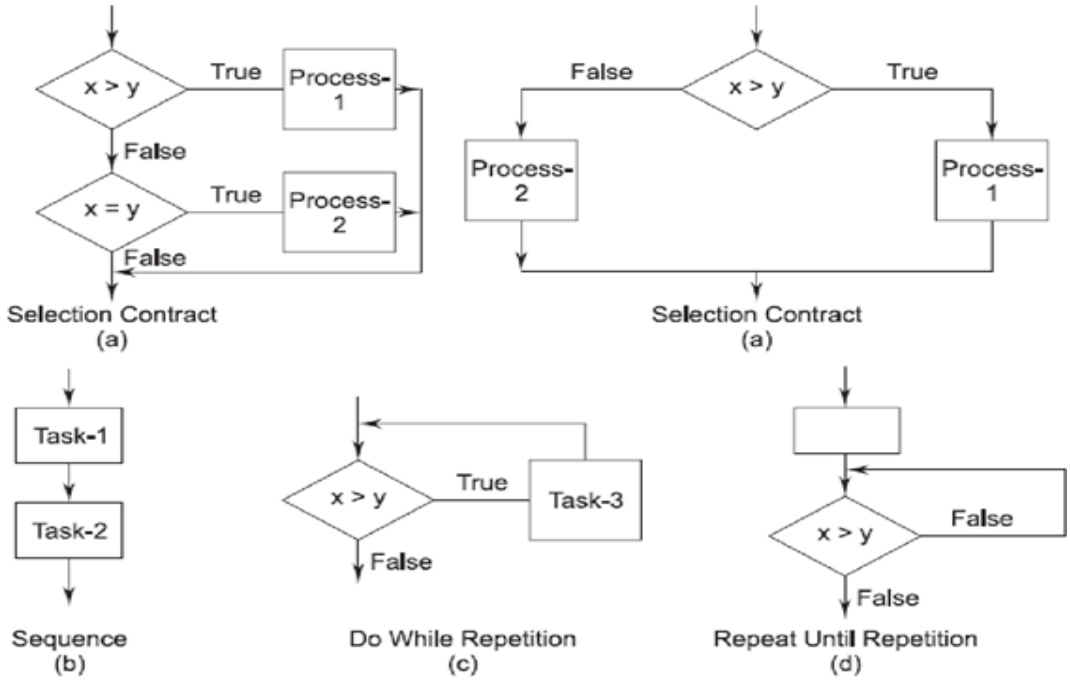


Figure 4.2: Basics of Structured Programming: Selection, Iterations, and Sequence

Advantages of Structured Programming

The advantages of structured programming are:

Is that it is very convenient to put logic systematically into the program. Due to the ease of handling complex logic, the user, reader, and programmer understand the program easily.

It is easy to verify, conduct reviews, and test the structured programs in an orderly manner. If errors are found, they are easy to locate and correct.

Conclusion

The goal of the coding phase is to translate the design of the system into code in a given programming language. In this phase the aim is to implement the design in the best possible manner. This phase affects both testing and maintenance phases. Well-written code can reduce the testing and maintenance effort.

Exercise

Explain the term "implementation" of the SDLC.

What is "structured programming"? How do modern programming languages, facilitate writing structured programs? (Use any programming language you are familiar with.

What are the advantages of writing structured programs versus unstructured programs?

State two advantages of structured programming.

Activity 2 – Software Testing Fundamentals

Testing Principles

Testing is a set of activities that can be planned in advance and conducted systematically. A number of software-testing strategies have been proposed in the literature. All provide the software developer with a template for testing. Before a software engineer make use of testing techniques, he/she should know the basic principles that script the testing process. A template for testing has the following basic principle (Agarwal et al., 2010):

All tests should be determined according to customer requirements

The purpose is to discover possible defects or flaws that cause the system to not function according to customer requirements.

Tests should be planned before you even start:

After finishing the requirements analysis process the test planning can start. Detailed Test cases can start as soon as the design model ends.

The Pareto principal is applied to software testing

Simple test using Pareto principle argues that 80 percent of all faults discovered during the testing phase can affect 20 percent of all program components. The problem at this stage is to isolate these components suspected by testing the same.

Testing should begin "in the small" and progress toward testing "in the large."

The first tests planned and executed generally focuses on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Exhaustive testing is not possible

The number of path permutations for even a moderately-sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

To be more efficient test should be conducted by an independent third party.

The software engineer who creates a system is not the best person to perform all program tests. Especially for large projects an independent test group is required.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source-code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy must provide guidance for the practitioner and a set of milestones for the manager.

Test Oracle

To test any program, we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this we need a test oracle.

A test oracle is a mechanism, different from the program itself, which can be used to check the correctness of the output of the program for the test cases.

Conceptually, we can consider testing a process in which the test cases are given to the test oracle and the program under testing. The output of the two is then are compared to determine if the program behaved correctly for the test cases, as shown in Figure 4.3.

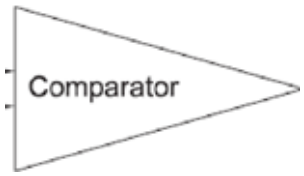


Figure 4.3: Test Oracles (Source: Agarwal et al., 2010)

Test oracles are considered as human, they can conduct tests when there is a discrepancy between the oracles and program results. First you need to check the results produced by the oracles before declaring that there is a flaw in the program. In the above illustrated picture, the test case data is according to the oracle and test the program on that will be tested. The result of each of the end elements is compared to determine if the Program behaved properly in accordance with the test case.

The human oracles generally use the specifications of the program to decide what the "correct" behavior of the program shouldn't be. To help the oracle to determine the correct behavior, it is important that the behavior of the system be unambiguously specified and the specification itself shouldn't be error-free.

Conclusion

Software testing means finding errors. While designing and implementing a computer-based system or a product, one should consider "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Exercise

What is testing? Explain the different types of testing performed during software development.

Define the various principles of testing.

What are test oracles?

Activity 3 – Levels of Testing

Introduction

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

Unit Testing

Integration Testing

System Testing

Test Unit

In unit testing individual components are tested to ensure they operate correctly. Focus on the verification effort. . On the smallest unit of software design, each component is tested independently without the other system components. There are some reasons to perform test drives instead of a testing the whole product:

The size of a simple module is small enough that you can allocate an error fairly easily.

The module is small enough that you can attempt to test it in some demonstrably some exhaustive fashion.

Shuffling the interaction of multiple errors in widely different parts of the software are eliminated.

In this case the module interface is tested to ensure that the information properly flows inside and outside the program unit that is being tested. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps of the execution algorithm. Conditions boundary tested are to ensure that modules enquiries operate at the boundaries established to limit or restrict processing. All independent paths through the control of structure are exercised to ensure that all the statements have Been module executed at least once. And finally, all errors handling paths are tested computation. According to Agarwal et al, 2010 there are some common errors in computation as follows:

- Codes Mixing operation
- incorrect Initialization
- Arithmetic incorrectly precedence
- Precision inaccuracy
- incorrect representation of representation of expressions

Test cases in unit testing should uncover errors like:

- Comparison of different data types
- incorrect Logical operators or precende
- incorrect Comparison of variables
- Improper loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables

Integration Testing

Another test level is the integration test. Integration test is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing .In such a test, unit-tested modules are combined into subsystems, which are then tested. The goal is to see if the modules can be integrated properly.

The following are the various approaches used to perform integration testing:

Incremental Approach

The incremental approach means to first combine only two components together and test them. Remove the errors if they are there, otherwise combine another component to it and then test again, and so on until the whole system is developed.

Top-Down Integration Testing

Top-down integration testing is an incremental approach to construction of program structures. Modules are integrated by moving downward through the control hierarchy beginning with the main control module.

Bottom-up integration

Bottom-up integration testing, as its name implies, begins construction and testing with the components at the lowest level in the program structure. A bottom-up integration strategy may be implemented with the following steps:

- Low-level components are combined into clusters (sometimes called builds) that perform specific software sub-functions.
- A driver (a control program for testing) is written to coordinate test case input and output.

The cluster is tested.

Drivers are removed and clusters are combined moving upward in the program structure.

Regression Test

Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors. The regression test suite contains three different classes of test cases:

- Additional tests that focus on software functions.
- A representative sample of tests that will exercise all software functions.
- Tests that focus on the software components that have been changed.

Smoke Test

Smoke testing is an integration testing approach that is commonly used when "shrink-wrapped" software products are developed. Smoke testing is characterized as a rolling integration approach because the software is rebuilt with new components and testing.

Smoke testing encompasses the following activities:

Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

A series of tests is designed to expose errors that will keep the build from properly performing its functions.

The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.

Smoke testing provides a number of benefits when it is applied on complex software engineering projects:

- Integration risk is minimized.
- Quality of end product is improved.
- Error diagnosis and correction are simplified.
- Progress is easier to assess.

Integration Sandwich

Sandwich integration testing is the combination of both the top-down and bottom-up approach. So, it is also called mixed integration testing. In it, the whole system is divided into three layers, just like a sandwich: the target is in the middle and one layer is above the target and one is below the target. The top-down approach is used in the layer that is above the target and the bottom-up approach is used in the layer that is below the target.

System Test

In the system test, subsystems are integrated to make the whole system. The testing process is concerned with finding errors that result from unanticipated interactions between subsystems and system components. It is also concerned with validating that the system meets its functional and non functional requirements. There are three essentially main kinds of system testing:

Alpha Testing

Alpha testing refers to the system testing carried out by the test team within the development organization. The alpha test is conducted at the developer's site by the customer under the project team's guidance. In this test, users test the software on the development platform and point out errors for correction. However, the alpha test, because a few users on the development platform conduct it, has limited ability to expose errors and correct them. Alpha tests are conducted in a controlled environment. It is a simulation of real-life usage. Once the alpha test is complete, the software product is ready for transition to the customer site for implementation and development.

Beast test

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

Beta tests are conducted at the customer site in an environment where the software is exposed to a number of users. The developer may or may not be present while the software is in use. So, beta testing is a real-life software experience without actual implementation. In this test, end users record their observations, mistakes, errors, and so on and report them periodically.

In a beta test, the user may suggest a modification, a major change, or a deviation. The development has to examine the proposed change and put it into the change management system for a smooth change from just developed software to a revised, better software. It is standard practice to put all such changes in subsequent version releases.

Acceptance Test

Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system. When customer software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than the software engineers, an acceptance test can range from an informal 'test drive' to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

Conclusion

Each level in software testing has its importance which provides a means of eliminating errors from early stages. In unit testing individual components are tested to ensure that they operate correctly, In integration testing, many unit-tested modules are combined into subsystems which are then tested. In system testing, subsystems are integrated to make up the whole system and then tested.

Exercise

What are the different levels of testing? Explain.

Suppose developed software has successfully passed all the three levels of testing, i.e., unit testing, integration testing, and system testing. Can we claim that the software is defect-free? Justify your answer.

What is unit testing?

What is integration testing? Which types of defects are uncovered during integration testing?

What is regression testing? When is regression testing done? How is regression testing performed?

Define sandwich testing.

Activity 4 – White-Box and Black-Box Testing

White-Box Testing/Structural Testing

A complementary approach to functional or black-box testing is called structural or white-box testing. In this approach, test groups must have complete knowledge of the internal structure of the software. Structural testing is an approach to testing where the tests are derived from knowledge of the software's structure and implementation. Structural testing is usually applied to relatively small program units, such as subroutines, or the operations associated with an object. As the name implies, the tester can analyze the code and use knowledge about the structure of a component to derive test data as shown in Figure 4.5. The analysis of the code can be used to find out how many test cases are needed to guarantee that all of the statements in the program are executed at least once during the testing process.

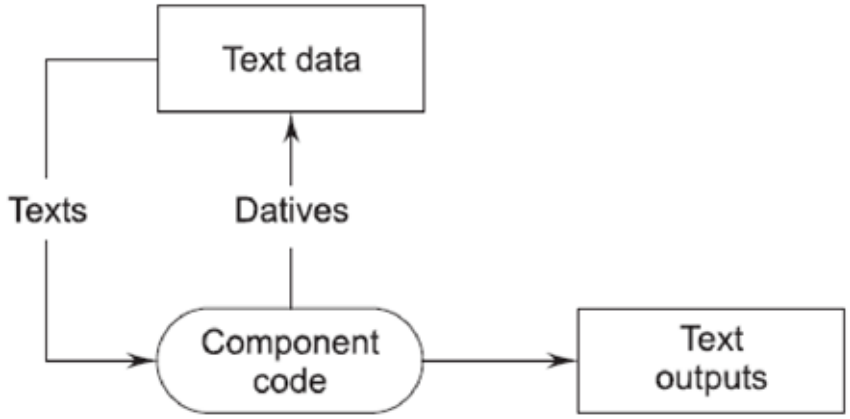


Figure 4.5: Structural Testing (Source: Agarwal et al., 2010)

In white-box testing, test cases are selected on the basis of examination of the code, rather than the specifications. White-box testing is illustrated in Figure 4.6.

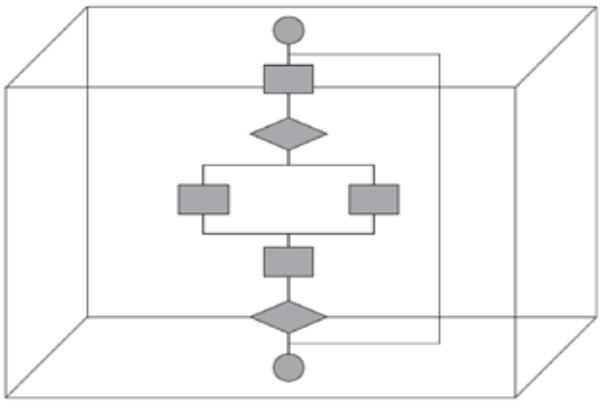


Figure 4.6: White-Box Testing (Source: Agarwal et al., 2010)

Using white-box testing methods the software engineer can test cases that:

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decision on their true and false sides.
- Exercise all loops at their boundaries.
- Exercise internal data structures to ensure their validity.

The nature of software defects are:

- Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
- We often believe that a logical path is not to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.

Reasons White-box Testing is Performed

White-box testing is carried out to test whether:

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

Advantages of Structural/White-box Testing

The various advantages of white-box testing include:

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

Functional/Black-Box Testing

In functional testing the structure of the program is not considered. Test cases are decided on the basis of the requirements or specifications of the program or module and the internals of the module or the program are not considered for selection of test cases.

Functional testing refers to testing that involves only observation of the output for certain input values, and there is no attempt to analyze the code, which produces the output. The internal structure of the program is ignored. For this reason, functional testing is sometimes referred to as black-box testing (also called behavioral testing) in which the content of a black-box is not known and the function of black box is understood completely in terms of its inputs and outputs.

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. Black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Other names for black-box testing (BBT) include specifications testing, behavioral testing, data-driven testing, functional testing, and input/output driven testing. In black-box testing, the tester only knows the inputs that can be given to the system and what output the system should give. In other words, the basis for deciding test cases in functional testing is the requirements or specifications of the system or module. This form of testing is also called functional or behavioral testing.

Black-box testing is not an alternative to white-box techniques; rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. Black-box testing identifies the following kinds of errors:

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

When these errors are controlled then:

- Function(s) are valid.
- A class of inputs is validated.
- Validity is sensitive to certain input values.
- The software is valid and dependable for a certain volume of data or transactions.
- Rare specific combinations are taken care of.

Black-box testing tries to answer the following questions:

- How is functional validity tested?
- How are system behavior and performance tested?
- How are the boundaries of a data class isolated?
- How will the specific combinations of data affect system operation?
- What data rates and data volume can the system tolerate?

- Is the system particularly sensitive to certain input values?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria:

- Test cases that reduce by a count that is greater than one.
- Test cases that tell us something about the presence or absence of classes of errors.

Advantages of Black-box Testing

The advantages of this type of testing include:

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

Test Plan

The test plan is a document that contains different test cases designed to test different test objects and different testing attributes. The plan puts the test of a logical and sequential order form according to the chosen strategy, top-down or bottom-up. Usually the test plan is a test matrix and list of test cases in accordance with the execution order of each task. Table 4.1 illustrates the matrix of test and test cases within the test. Test ID, test name, and test cases are designed well before the development phase and have been designed for those who conduct the tests. A test plan states:

- The items to be tested.
- At what level they will be tested at.
- The sequence they are to be tested in.
- How the test strategy will be applied to the testing of each item and the test environment.

										Test
Planned Date		N	…	4	3	2	1	Test	Test ID	
Successful	Completed							Name	Tester	ID

Table 4.1: Test Plan (Source: Agarwal et al., 2010)**Tests-Case Design**

A test case is a set of instructions designed to discover a particular type of error or defect in the software system by inducing a failure. The goal of selected test cases is to ensure that there is no error in the program and if there is it then should be immediately depicted. Ideal test casement should contain all inputs to the program. This is often called exhaustive testing.

There are two criteria for the selection of test cases:

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

Each test case needs proper documentation, preferably in a fixed format. There are many formats; one format is suggested in Table 4.2:

<table>
<tr><th>Test Case Name</th><th>Test Case ID</th></tr>
<tr><td>Purpose of Test</td><td>Testing Object (Unit, Application, Module, etc)</td></tr>
<tr><td colspan="2">Test Attribute</td></tr>
<tr><td colspan="2">Tests focus (function, feature, process, interface, validation, verification, etc.)</td></tr>
<tr><td colspan="2">Test type (alpha, beta, unit, integration, system)</td></tr>
<tr><td rowspan="2">Test Process</td><td>A set of instructions for conducting the test-initial stating</td></tr>
<tr><td>condition-inputs-specifications-output expected</td></tr>
<tr><td>Test Results</td><td>Expected and actual and comparison, error description, post-process state</td></tr>
<tr><td>Action</td><td>Correction, authorization, and feedback through retest</td></tr>
<tr><td colspan="2">Action to initialize the pre-test status</td></tr>
</table>

Table 4.2: Test Case Documentation Format (Source: Agarwal et al., 2010)

Conclusion

For conventional applications, software is tested from two different perspectives: internal program logic is exercised using "white box" test-case design techniques and software requirements are exercised using "black box" test-case design techniques. Use cases assist in the design of tests to uncover errors at the software validation level. In every case, the intent is to find the maximum number of errors with the minimum amount of effort and time. A set of test cases designed to exercise internal logic, interfaces, component collaborations, and external requirements is designed and documented, expected results a redefined, and actual results are recorded (Pressman, 2001).

Exercise

What is a test case? What is test-case design?

What is the difference between

Black-box testing and white-box testing

Top-down and bottom-up testing approaches

Alpha and beta testing

What are test plans and test cases? Illustrate each with an example.

Why does software testing need extensive planning? Explain.

What is smoke testing?

Differentiate between integration testing and system testing.

Define structural testing. Give the various reasons structural testing is performed.

UNIT SUMMARY

In this unit aspects related to the implementation/coding phase and testing were discussed. In coding phase, a structured programming has been explained more. The primary objective for test case design is to derive a set of tests that have the highest likelihood for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage.

The difference between the various testing techniques such as unit testing, integration and system were explained in detail. The need to implement a test plan, or a life cycle of software testing and how the techniques and tools are integrated into this was discussed in this chapter.

Unit Assessment

Check your understanding!

With the aid of sketches, explain three basic control structures of the structured programming. (Answer: section 4.1.5)

Describe two principles of structured programming. (Answer: section 4.1.5)

Why is regression testing important? When is it used? (Answer: section 4.3.3)

List and describe four software testing principles. (Answer: section 4.2.1)

List levels of software testing and briefly describe one level. (Answer: section 4.3)

Define the term "test plan" as applied in software testing. (Answer: section 4.4.3)

Unit Readings and Other Resources

- Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1
- Pressman R. S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783
- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.

Unit V: Maintenance and Project Management

Unit Introduction

Software will definitely undergo change after it is delivered to the customer. This maintenance phase starts with the system being installed for practical use, after the product is delivered to the client. It lasts till the beginning of system's retirement phase. Maintenance does not normally involve major changes to the system's architecture. Changes are implemented by modifying existing components and adding new components to the system. Maintenance includes all changes to the product once the client has agreed that it satisfied the specification document.

Software is a difficult activity. Lots of things can go wrong. Understanding the risks and taking proactive measures to avoid or manage them—is a key element of good software

Unit Objectives

Upon completion of this unit you should be able to:

- Explain why software maintenance is needed
- Describe categories of maintenance
- Describe factors affecting maintenance
- Define what is risk analysis and management
- Describe categories of risks and risk management
- Identify risks and produce risk item checklists
- Perform the test plans

KEY TERMS

Software Maintenance:

Software maintenance is the activity associated with keeping an operational computer system continuously in tune with the requirements of users and data processing operations.

Software Risk Analysis and Management

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty.

Risk

Risk is defined as an exposure to the chance of injury or loss

Risk Management

Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal

Learning Activities

Activity 1 – Software Maintenance Phase

Introduction

Software maintenance is the activity associated with keeping an operational computer system continuously in tune with the requirements of users and data processing operations. Software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities and optimization. The software maintenance process is expensive and risky and is very challenging. There is a need for software maintenance due to the following reasons:

- Changes in user requirements with time. That is the customer requires functional or performance enhancements
- Program/System problems, errors have been encountered
- The software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), - Changing hardware/Software environment
- To improve system efficiency and throughout
- To modify the components
- To test the resulting product to verify the correctness of changes
- To eliminate any unwanted side effects resulting from modifications
- To augment or fine-tune the software
- To optimize the code to run faster
- To review standards and efficiency
- To make the code easier to understand and work with
- To eliminate any deviations from specifications

Categories of Maintenance

Maintenance may be classified into the four categories as follows:

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

Corrective Maintenance

Corrective maintenance means repairing processing or performance failures or making changes because of previously uncorrected problems. involves correcting errors which were not discovered in earlier stages of the development process while leaving the specification unchanged.

Adaptive Maintenance

Adaptive maintenance means changing the program functions. This is done to adapt to external environment changes in which the product operates such as new government regulations. This type is known as enhancement maintenance.

Perfective Maintenance

Perfective maintenance means enhancing the performance or modifying the programs to respond to the user's additional or changing needs. Involves changes that the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time. This type is also a kind of enhancement maintenance

Preventive Maintenance

Preventive maintenance is the process of preventing systems from being obsolete. Preventive maintenance involves the concept of re-engineering and reverse engineering in which an old system with an old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

Studies have indicated that, on average, maintainers spend approximately 17 % of their time on corrective maintenance, 65% on perfective maintenance, and 18% on adaptive maintenance as shown in Figure 5.1(Sommeville, 2000). It is estimated that between 40% and 70% of the overall software development lifecycle costs are spent on maintenance.

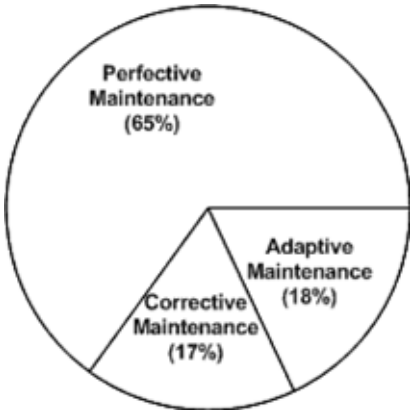


Figure 5.1: Distribution of maintenance effort (Source: Sommeville, 2000)

Maintenance Costs

Maintenance cost is usually greater than development costs (2* to 100* depending on the application), it is affected by both technical and non-technical factors. The cost increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.

It is advisable to invest more effort in early phases of the software life-cycle to reduce maintenance costs. The defect repair ratio increases heavily from the analysis phase to the implementation phase as shown in Table 5.1. Therefore, more effort during development will certainly reduce the cost of maintenance.

Table 5.1: Defect Repair Ratio

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Factors Affecting Maintenance

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

New Application

As users gain experience of new application, they will begin to see potential improvement and features

Staff Mobility

It is always easier for original programmers to update the code than someone else. When staff move on, it becomes harder to maintain code unless it is very well documented.

Too Many Versions

It can be difficult to trace changes in code if there have been a number of releases.

Insufficient Documentation

If the design documentation or internal commentary is poor or missing, then maintenance will be affected. It is good practice to use internal commentary and descriptive variable names.

External Hardware and Software Changes

Changes to hardware platforms, or upgrades to operating systems can affect maintenance requirements.

Conclusion

Technology is changing rapidly and business requirements or functions and information technology that support organizations are changing very fast. This rapid change causes the continuous maintenance of software on businesses. For computer software, change occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when the application is reengineered to provide benefit in a modern context.

Exercise

What is software maintenance? Describe various categories of maintenance. Which category consumes maximum effort and why?

Some people feel that "maintenance is manageable." What is your opinion about this issue?

Explain the different types of maintenance

Why is maintenance needed?

What are the different types of maintenance that a software product might need? Why is such maintenance required?

Activity 2 – Software Risk Analysis and Management

Introduction

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

Risk is defined as an exposure to the chance of injury or loss. That is, risk implies that there is a possibility that something negative may happen. In the context of software projects, negative implies that there is an adverse effect on cost, quality, or schedule.

Risk management is the area that tries to ensure that the impact of risks on cost, quality, and schedule is minimal

Software Risks

Risk always involves two characteristics

Uncertainty—the risk may or may not happen; that is, there are no 100% probable risks.1

Loss—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project Risks

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.

Technical Risks

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than we thought it would be.

Business Risks

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are:

Building a excellent product or system that no one really wants (market risk),

Building a product that no longer fits into the overall business strategy for the company (strategic risk),

Building a product that the sales force doesn't understand how to sell,

Losing the support of senior management due to a change in focus or a change in people (management risk), and Losing budgetary or personnel commitment (budget risks).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). A first step toward avoiding risks when possible and controlling them when necessary is by identifying known and predictable risks. There are two distinct types of risks for each of the categories that have been presented above: generic risks and product-specific risks. Generic risks are a potential threat to every software project.

Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand.

To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed:

"What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

Product size—risks associated with the overall size of the software to be built or modified.

Business impact—risks associated with constraints imposed by management or the marketplace.

Customer characteristics—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization.

Development environment—risks associated with the availability and quality of the tools to be used to build the product.

Technology to be built—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" are listed along with their probability of occurrence.

Although generic risks are important to consider, usually the product-specific risks cause the most headaches. Be certain to spend the time to identify as many product-specific risks as possible.

Risk Management Categories

Risk management plays an important role in ensuring that the software product is error-free. Firstly, risk management takes care that the risk is avoided, and if it is not avoidable, then the risk is detected, controlled, and finally recovered.

Riisk management can be categorized as follows:

Risk Avoidance: involving risk anticipation and risk tools

The first phase is to avoid risk by anticipating and using tools from previous project histories. In the case where there is no risk, the risk manager stops.

Risk Detection: involving risk analysis, risk category and risk prioritization

In the case of risk, detection is done using various risk analysis techniques and further prioritizing risks.

Risk Control: involving risk pending, risk resolution and risk not solvable

Risk is then being controlled by pending risks, resolving risks, and in the worst case (if the risk is not solved), lowering the priority.

Risk Recovery: involving full, partial and extra/alternate feature

Lastly, risk recovery is done fully, partially, or an alternate solution is found.

The African Virtual University Headquarters

Cape Office Park

Ring Road Kilimani

PO Box 25405-00603

Nairobi, Kenya

Tel: +254 20 25283333

contact@avu.org

oer@avu.org

The African Virtual University Regional Office in Dakar

Université Virtuelle Africaine

Bureau Régional de l'Afrique de l'Ouest

Sicap Liberté VI Extension

Villa No.8 VDN

B.P. 50609 Dakar, Sénégal

Tel: +221 338670324

bureauregional@avu.org